Universidad de las Ciencias Informáticas

Facultad 4

Arquitectura de software para videojuegos desarrollados sobre el motor de juego Unity 3D

Trabajo de diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autor: Alejandro Andrés Pi Cruz

Tutor (es): Ing. Javier Alejandro Domínguez Falcón

Ing. Andy Hernández Paez

"Año 59 de la Revolución"
La Habana, Cuba
Julio 2017

“El éxito es que cada meta sea un paso y que cada paso sea una meta”

Johann Peter Eckerman

Declaración de autoría

Declaro ser el único autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.
Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

_____

Alejandro Andrés Pi Cruz
Autor

_____

Ing. Javier Alejandro Domínguez Falcón
Tutor

_____

Ing. Andy Hernández Paez
Tutor

Datos de contacto

Tutor: Ing. Javier Alejandro Domínguez Falcón
Edad: 26.
Ciudadanía: cubana.
Institución: Universidad de las Ciencias Informáticas (UCI).
Título: Ingeniería en Ciencias Informáticas.
Síntesis del tutor:

E-mail: jafalcon@uci.cu

Tutor: Ing. Andy Hernández Páez.
Edad: 28.
Ciudadanía: cubana.
Institución: Universidad de las Ciencias Informáticas (UCI).
Título: Ingeniería en Ciencias Informáticas.
Síntesis del tutor:

E-mail: andyhp@uci.cu

Dedicatoria

Para mi madre, por darme su amor incondicional y por creer siempre en mí. Gracias por ser la mejor madre del mundo. Te quiero mucho.

Para mi abuelo, donde sea que este que sé que estará orgulloso.

Para mi abuela, por consentirme demasiado y aguantarme.

A mi tía, hermana, cuñado, en general para mi familia.

Para todos mis amigos.

Agradecimientos

A mi familia, en especial a mi madre por haberme apoyado en todo momento y haber creído que podría cumplir este sueño.

A mi tutor Andy que sin su ayuda y sus consejos me hubiera sido imposible hacer esta tesis, por su paciencia a la hora de revisarme el documento, por todo lo que aprendí gracias a él y por los días que tuvo que apartar su trabajo para atenderme.

A mi tutor Javier que me tuvo a su lado molestándolo, haciéndole preguntas e interrumpiendo su trabajo y tuvo la paciencia para enseñarme, guiarme y ayudarme incondicionalmente cada vez que necesité de él.

A mi oponente Roberto por haberme ayudado desde el principio (incluso cuando no era mi oponente), por su aporte en la confección de esta tesis y sus consejos.

Al jurado por haberme guiado a hacer una mejor tesis.

A todas las personas que han colaborado de una forma u otra a la realización de este trabajo y a mi formación, especialmente a los profesores, ingenieros, personal del centro y alumnos ayudantes que tanto me han aportado no solo como profesional sino como persona.

A todos los amigos de la universidad, del apartamento, del aula, del año, de la facultad, de los laboratorios, del café, de los festivales, del grupo, de los videojuegos, los que se han quedado atrás pero nos alcanzarán y los que no han podido continuar.

A la UCI, en especial a la facultad 5, de la que siempre seré un grundy.

A todos, muchas gracias.

Resumen

En los últimos años la arquitectura de software se ha consolidado como una disciplina que intenta contrarrestar los efectos negativos que pueden surgir durante el desarrollo de un producto informático, ocupando un rol significativo en la estrategia de negocio de una organización que basa sus operaciones en el software, volviéndose necesaria para todo tipo de desarrollos, incluyendo los videojuegos.

La presente investigación tiene como objetivo principal diseñar una arquitectura de software para videojuegos desarrollados sobre el motor de juego Unity 3D, que permita organizar y estructurar sus características funcionales básicas. Mediante el estudio de las arquitecturas usadas en videojuegos tanto a nivel internacional como nacional fueron agrupadas las clases candidatas a pertenecer a la arquitectura del sistema, identificándose los paquetes principales, las dependencias entre ellos, los patrones de diseño y las buenas prácticas empleadas, concretando una arquitectura de software basada en la integración de los tipos de arquitectura: en capas y basada en componentes.

Para la validación de la propuesta fue usado un prototipo funcional de un videojuego del género plataformas, empleando para describirlo elementos de diseño de videojuegos, especificación de mecanismos y las vistas propuestas por Robert Nord: conceptual, de módulos, de código y de implementación. Se obtuvo como resultado un videojuego de plataformas que usa la propuesta de solución como arquitectura base.

La arquitectura propuesta fue validada a través de las técnicas de evaluación basadas en prototipos, en escenarios, en conjunto con la aplicación del método de Análisis de Acuerdos de Arquitectura de Software (ATAM). Mediante la técnica enunciada se pudo identificar los riesgos presentes en la propuesta realizada y determinar que la arquitectura satisface los atributos de calidad definidos para la presente investigación según el modelo de calidad ISO/IEC 25010.

Palabras clave: Arquitectura de software, ISO/IEC 25010, Unity 3D, videojuego.

Índice de contenidos

Índice de figuras

Índice de tablas

Introducción

El uso de las Tecnologías de Información y las Comunicaciones (TIC) en los últimos años ha permitido extender la informática a muchos sectores, ocupando un lugar creciente en la vida humana y en el funcionamiento de las sociedades. Uno de los procesos que mayor impacto ha tenido en la sociedad es el de creación de videojuegos, que se vuelve más complejo con el tiempo producto del aumento de los requerimientos de los usuarios.

Los videojuegos son una vía de entretenimiento interactivo en el que uno o varios usuarios, mediante un dispositivo de entrada, se comunican con un sistema que posea imágenes de video. La plataforma en la que se desarrolle (o sistema) puede ser una computadora, consola, o incluso un celular. La interactividad usuario–sistema está dada por la capacidad del equipo de trabajo de planificar un sistema en el que el usuario se sienta cómodo y controle la situación [1].

Estos productos de software han evolucionado con increíble rapidez en los últimos años, convirtiéndose en una industria que cuenta con equipos de trabajo multidisciplinarios, al punto que para desarrollar un videojuego es necesario, desde el concepto inicial hasta su versión final, el conocimiento de varias disciplinas y habilidades por parte del equipo de trabajo, tales como programación, arquitectura de software, diseño y *marketing*. El desarrollo de productos de este tipo pasa por la creatividad y recorre varias ciencias formales y sociales, convirtiéndose en más que un típico desarrollo de software.

A lo largo de la historia de los videojuegos, sus creadores han ido dando lugar a una variedad creciente de géneros en las distintas plataformas disponibles. Estos géneros se han ido conformando en torno a factores como: la representación gráfica, el tipo de interacción entre el jugador y la máquina, la ambientación, y su sistema de juego, siendo este último el criterio más habitual a tener en cuenta. Los géneros o categorías de videojuegos son una forma de clasificar los videojuegos en función fundamentalmente de su mecánica de juego. Otros factores, como la estética o la temática, pueden también influir a la hora de definir ciertos géneros [2].

Entre las categorías de videojuegos existentes, una de las que más popularidad tiene actualmente es plataformas. Los videojuegos pertenecientes a ese género se caracterizan por tener un personaje que debe caminar, correr, saltar o escalar sobre una serie de plataformas y acantilados, con enemigos, mientras se recogen objetos para poder completar el juego [2].

Al igual que ocurre en otras disciplinas en el campo de la informática, el desarrollo de videojuegos se ha beneficiado de la aparición de herramientas que facilitan dicho desarrollo, automatizando determinadas tareas y ocultando la complejidad inherente a muchos procesos de bajo nivel, como es el caso de los motores de videojuegos.

Actualmente, se pueden encontrar en los mercados diferentes opciones de videojuegos tanto basados en licencias libres como comerciales que usan diferentes motores, los que se encargan de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, *scripting*, animación, inteligencia artificial, redes, *streaming*, administración de memoria y un escenario gráfico [3]. La variedad y cantidad de motores disponibles en el mercado amerita un arduo trabajo para los desarrolladores al momento de decidir o portar sus aplicaciones de un motor a otro, pues cada uno posee su propio conjunto de características, métodos de inicialización e interfaces.

Entre los motores de juego más usados en la actualidad se encuentra Unity 3D, que es usado para realizar proyectos tanto 2D como 3D dado que posee un entorno de desarrollo sencillo de manejar para los principiantes y suficientemente potente para los expertos, permitiendo crear fácilmente videojuegos y aplicaciones para multitud de plataformas. Esto, sumado a la características de los lenguajes de programación que permite (C#, JavaScript) hace que sea el que se emplee en los proyectos de desarrollo de videojuegos en la Universidad de Ciencias Informáticas (UCI).

El diseño y desarrollo de un videojuego es demasiado complejo para que pueda ser abordado por completo sin usar una estructura lógica que describa los componentes que lo forman. Otro aspecto esencial para estructurar y desarrollar un videojuego con la calidad deseada es la Arquitectura de Software (AS), la que se encarga de definir, de forma abstracta, los componentes de un sistema, sus interfaces y la comunicación entre ellos. Una AS, consiste en un conjunto de patrones y abstracciones coherentes que proporcionan soluciones a problemas de manera eficiente [4]. Su necesidad de empleo viene dada porque esta rama de la informática selecciona y diseña con base en objetivos (requerimientos) y restricciones, y en los videojuegos, una AS consistente satisface objetivos de todo desarrollo de este tipo como mantenibilidad, extensibilidad, flexibilidad e interacción con otros sistemas de información.

En la Universidad de Ciencias Informáticas (UCI) en colaboración con los estudios de animación del Instituto Cubano de Arte e Industria Cinematográfico (ICAIC) se realizan videojuegos de diferentes géneros. Para ello la universidad cuenta con el Centro de Entornos Interactivos 3D Vertex, en el cual se tiene como objetivo generar soluciones integrales, tecnologías, productos y servicios informáticos que cumplan con las necesidades y expectativas del usuario; potenciando la formación especializada y las investigaciones afines que garanticen un alto valor agregado. También tiene como fin desarrollar productos y servicios informáticos asociados a los Entornos Virtuales Interactivos 3D, con un alto valor agregado resultado de un ciclo completo de I+D+I: Investigación-Desarrollo e Innovación.

Este centro ha desarrollado 5 productos usando el motor de juego Unity 3D, que han sido liberados satisfactoriamente desde la dirección de calidad de la universidad: Súper Claria, La Chivichana, Aventuras en la Manigua, Especies invasoras y La Neurona. Después de la entrega de los productos

antes mencionados se ha continuado el desarrollo de nuevos videojuegos, pero se ha detectado, mediante consultas con fuentes de información (desarrolladores del centro) y observación de los productos desplegados, que existe la siguiente **situación problemática** en el centro Vertex:

- Falta de organización en las funcionalidades de un videojuego sobre la herramienta de desarrollo Unity 3D lo que provoca poca representatividad funcional de las características del producto.
- La inexistencia de una estructura lógica basada en paquetes de la implementación dificulta el entendimiento de esta por parte del equipo de desarrollo y de los nuevos desarrolladores.
- Poca representación y documentación que describa el funcionamiento de un videojuego que se desarrolle sobre el motor de juego Unity 3D en el centro, lo que obstaculiza el proceso de su aprendizaje para los desarrolladores.
- No se tiene referencia sobre el código fuente que se necesita para el desarrollo de nuevos videojuegos, lo que provoca desorganización en la implementación y diversidad en el modo que se implementa en los proyectos de este tipo.
- La no reutilización de funcionalidades implementadas de juegos terminados en los nuevos desarrollos trae como consecuencia la pérdida de tiempo del equipo de trabajo en volver a implementar componentes que fueron utilizados con anterioridad.

Por la **situación problemática** antes expuesta se propone realizar la investigación a partir del siguiente **problema a resolver**: ¿Cómo establecer una organización y estructuración funcional de las características básicas de un videojuego que se desarrolle sobre el motor de juego Unity 3D?

Se define como **objeto de estudio**: Arquitectura de Software.

A raíz del problema de investigación se especifica como **objetivo general:** Diseñar una arquitectura de software para videojuegos desarrollados sobre el motor de juego Unity 3D, que permita organizar y estructurar sus características funcionales básicas.

Partiendo del objetivo general antes propuesto, el **campo de acción** de esta investigación es: Arquitectura de software para videojuegos desarrollados sobre el motor de juego Unity 3D.

Para darle cumplimiento al objetivo planteado se proponen las siguientes **tareas:**

1. Elaboración del marco teórico a partir del estado del arte actual referente al tema.
2. Análisis de la arquitectura de software de productos establecidos en el dominio de aplicación de videojuegos, para identificar elementos a reutilizar.
3. Análisis de especificaciones estándares que posibilitan el desarrollo de software basado en componentes.
4. Diseño de la arquitectura de software para videojuegos que se desarrollen sobre el motor Unity 3D.

5. Implementación de un prototipo funcional de un videojuego de plataformas que permita validar la arquitectura propuesta.
6. Validación de la propuesta arquitectónica.

Como **métodos científicos de investigación,** usados para consultar información novedosa sobre el tema y concretar su relevancia se emplearon los siguientes:

Teóricos:

Histórico – Lógico: Método teórico que se utilizará para analizar la evolución y las tendencias actuales de las arquitecturas de software.

Analítico – Sintético: Método teórico que se utilizará para extraer y analizar la información sobre las principales arquitecturas de software usadas en el Centro de Entornos Interactivos 3D Vertex.

Empíricos:

Consultas de fuente de información: Método empírico que se utilizará para la consulta de fuentes bibliográficas durante la investigación.

Observación: Método empírico que se utilizará para observar los resultados obtenidos en la caracterización e identificación de los principales tipos de arquitecturas de software y decidir luego cuál o cuáles serán más adecuadas.

Estructura de la tesis:

- **Capítulo 1.** Fundamentación teórica: En este capítulo se hace un estudio del estado del arte de la arquitectura de software para videojuegos, definiendo conceptos de importancia como AS, vistas arquitectónicas, videojuegos y abordando temas referentes a los diferentes modelos arquitectónicos existentes. Se realiza un estudio del arte de diferentes AS que han sido propuestas para videojuegos por diversos autores en el ámbito internacional y las empleadas en el desarrollo de videojuegos en el centro Vertex.
- **Capítulo 2.** Propuesta de solución: En este capítulo se definen las herramientas y metodologías usadas para dar una solución al problema de investigación planteado. Se plantean las diferentes vistas del modelo arquitectónico sobre el cual se implementó la propuesta de solución, usando el enfoque de Robert Nord para describir la AS.
- **Capítulo 3.** Evaluación de la arquitectura propuesta: Este capítulo expone algunos conceptos asociados a la evaluación de las arquitecturas de software. Se evalúa la arquitectura propuesta mediante el Método de Análisis de Acuerdos de Arquitectura de Software (*Architecture Trade-off Analysis Method*, ATAM). Además, se aplicaron técnicas basadas en escenarios y prototipo sobre el videojuego de plataformas desarrollado.

Capítulo #1. Fundamentación teórica

1.1 Introducción

En este capítulo se introduce el concepto de arquitectura de software tomando como referencia diferentes autores. Se aclaran las definiciones de vista arquitectónica que describen la solución propuesta. Se analizan los estilos arquitectónicos, patrones de arquitectura, tipos de videojuegos, con una aproximación hacia el género plataformas. Es explicada la importancia de usar una AS para el desarrollo de cualquier software y en específico, los videojuegos. Se hace un estudio del arte en el mundo, particularmente en el centro Vertex para encontrar elementos comunes que puedan usarse en la propuesta de solución.

1.2 Arquitectura de software

El término arquitectura de software ha sido definido dicho término por un gran número de autores, quienes aportan sus puntos de vista para facilitar el entendimiento de dicho concepto.

Es conveniente definir el concepto, ya que hoy en día el término de arquitectura se usa para referirse a varios aspectos relacionados con las Tecnologías de la Información (TI). De acuerdo al *Software Engineering Institute* (SEI), la Arquitectura de Software se refiere a *"las estructuras de un sistema, compuestas de elementos con propiedades visibles de forma externa y las relaciones que existen entre ellos"* [5].

El Instituto de Ingeniería Eléctrica y Electrónica IEEE propone: "La Arquitectura del Software es la organización fundamental de un sistema formada por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán, y los principios que orientan su diseño y evolución" [6].

Garlan y otros por su parte refieren: "...es el nivel del diseño del software donde se definen la estructura y propiedades globales del sistema". "...se centra en aquellos aspectos del diseño y desarrollo que no pueden tratarse de forma adecuada dentro de los módulos que forman el sistema" [7].

"Más allá de los algoritmos y estructuras de datos de la computación; el diseño y la especificación de la estructura general del sistema emergen como una clase nueva de problema. Los aspectos estructurales incluyen la estructura global de control y la organización general; protocolos de comunicación, sincronización y acceso de datos; asignación de funciones para diseñar elementos; distribución física, composición de elementos de diseño; ajuste y rendimiento; y selección entre otras alternativas de diseño" [8].

Por otro lado, Philippe Kruchten en el *Rational Unified Process* (RUP) plantea: "La arquitectura de software representa la estructura o las estructuras del sistema, que consta de componentes de software, las propiedades visibles externamente y las relaciones entre ellas" [9].

Robert Nord plantea [10]:

“Arquitectura de software es sobre tomar decisiones estructurales fundamentales que son costosos de cambiar una vez implementado. Opciones de arquitectura de software, también llamados decisiones arquitectónicas, incluyen opciones estructurales específicas de posibilidades en el diseño de software. Por ejemplo, los sistemas que controlaban la lanzadera de espacio vehículo de lanzamiento tenía el requisito de ser muy rápido y muy confiable. Por lo tanto, una adecuada Computación en tiempo real lengua tendría que ser elegido. Además, para satisfacer la necesidad de fiabilidad la elección podría realizarse para tener múltiples redundantes e independientemente producido copias del programa y ejecutar estas copias en independiente *hardware* y contrastar resultados.”

La AS tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, así como requerimientos no funcionales. Es una solución a las dificultades que con el tiempo se han ido descubriendo, desarrollando formas y guías generales, con base a las cuales se puedan resolver esos problemas.

1.2.1 Principales dificultades causadas por deficiencias en una AS

Por la inexistencia de una estructura lógica que se encargue del funcionamiento y acoplamiento de diversos elementos dentro de un sistema o deficiencias en dicha estructura pueden ocasionarse problemas, tales como [10]:

- Falla en cumplir con la calidad necesaria.
- Falla en intentar soportar las necesidades de negocio.
- Falta de compromiso con el usuario para que el proyecto termine (problemas en la comunicación y exceso de tiempo de producción).

1.2.2 Áreas de investigación de las AS

Hay pocas caracterizaciones en torno de las áreas que componen el territorio de investigaciones sobre AS. David Garlan y Dewayne Perry, en su introducción al volumen de abril de 1995 de IEEE *Transactions on Software Engineering* dedicado a la AS, en el cual se delinean las áreas de investigación más promisorias, enumeran las siguientes [7]:

- Lenguajes de descripción de arquitecturas.
- Fundamentos formales de la AS (bases matemáticas, caracterizaciones formales de propiedades extra-funcionales tales como mantenibilidad o teorías de la interconexión.

- Técnicas de análisis arquitectónicas.
- Métodos de desarrollo basados en arquitectura.
- Recuperación y reutilización de arquitectura.
- Codificación y guía arquitectónica.
- Herramientas y ambientes de diseño arquitectónico.
- Estudios de casos.

Paul Clements [5] define cinco temas fundamentales que se agrupan en la disciplina de AS:

- **Diseño o selección de la arquitectura:** Cómo crear o seleccionar una arquitectura en base de requerimientos funcionales, de rendimiento o de calidad.
- **Representación de la arquitectura:** Cómo comunicar una arquitectura. Este problema se ha manifestado como el problema de la representación de arquitecturas utilizando recursos lingüísticos, pero el problema también incluye la selección del conjunto de información a ser comunicada.
- **Evaluación y análisis de la arquitectura:** Cómo analizar una arquitectura para predecir cualidades del sistema en que se manifiesta. Un problema semejante es cómo comparar y escoger entre diversas arquitecturas en competencia.
- **Desarrollo y evolución basados en arquitectura:** Cómo construir y mantener un sistema dada una representación de la cual se cree que es la arquitectura que resolverá el problema correspondiente.
- **Recuperación de la arquitectura:** Cómo hacer que un sistema evolucione cuando los cambios afectan su estructura; para los sistemas de los que se carezca de documentación confiable, esto involucra primero una “arqueología arquitectónica” que extraiga su arquitectura.

Entre los temas fundamentales empleados para comprobar la calidad de una arquitectura de software se encuentra la evaluación de la AS, que contribuye a comprobar que una arquitectura cumple con los atributos de calidad necesarios. Diseñar una correcta arquitectura va a determinar el éxito o fracaso de un videojuego, en la medida que esta cumpla o no con sus objetivos, lo que evidencia la necesidad de evaluarla correctamente.

Una de las áreas de investigación de las AS que ha aumentado considerablemente su desarrollo en los últimos años es la de videojuegos y como emplearlas para estructurar un *software* de ese tipo. La necesidad de describir el funcionamiento de los videojuegos, sus componentes, interfaces y comunicación entre ellos ha traído consigo la urgencia de sentar bases para la posterior creación de *software* que engloben el mismo género, de manera que facilite el trabajo de los desarrolladores, reutilizando al máximo los componentes que lo permitan y acortando tiempos en la producción.

1.3 Conceptualización de videojuegos

Un videojuego o juego de video es un *software* creado para el entretenimiento en general y basado en la interacción entre una o varias personas y un aparato electrónico que ejecuta dicho videojuego; este dispositivo electrónico puede ser una computadora, una máquina *arcade*, una videoconsola, un dispositivo de mano (un teléfono móvil, por ejemplo) los cuales son conocidos como "plataformas" [11].

Aunque, usualmente el término "video" en la palabra "videojuego" se refiere en sí a un visualizador de gráficos, hoy en día se utiliza para hacer referencia a cualquier tipo de visualizador. Se entiende por videojuegos todo tipo de juego digital interactivo, con independencia de su soporte [12].

1.3.1 Géneros de videojuegos

A lo largo de todos estos años de evolución, han ido naciendo nuevas clases de videojuegos, e incluso los nuevos proyectos es difícil encasillarlos en un género, pues abarcan muchos o los fusionan, estos podrían ser [13]:

- Aventura.
- *Arcade*.
- Deportivos.
- Disparos en primera persona.
- Disparos en tercera persona.
- *Sandbox*.
- *Shoot 'em up*.
- *Beat 'em up*.
- Sigilo.
- Estrategia.
- Rol.
- Simulación.
- Plataformas.

En el presente trabajo se pretende realizar un prototipo funcional de un videojuego perteneciente al género plataformas con el objetivo de poder evaluar la propuesta de solución de la presente investigación, por lo que es necesario conceptualizar el género de una manera más cercana.

Plataformas

En los videojuegos de **plataformas** el jugador controla a un personaje que debe avanzar por el escenario evitando obstáculos físicos, ya sea saltando, escalando o agachándose. Además, de las capacidades de desplazamiento como saltar o correr, los personajes de los juegos de plataformas

poseen frecuentemente la habilidad de realizar ataques que les permiten vencer a sus enemigos, convirtiéndose así en juegos de acción. Inicialmente los personajes se movían por niveles con un desarrollo horizontal, pero con la llegada de los gráficos 3D este desarrollo se ha ampliado hacia todas las direcciones posibles.

Los videojuegos de plataformas son uno de los primeros tipos de juegos que aparecieron en los ordenadores. Aunque este género fue muy popular en los 80 y 90, su popularidad ha disminuido en los últimos años, aún más cuando se introdujeron los gráficos 3D en los videojuegos. Esto se debe en gran parte a que las tecnologías de desarrollo en 3D han hecho que se perdiese la simplicidad que caracterizaba este género [12].

1.3.2 Características principales

En resumen, estas son las características principales de un juego de plataformas:

- El personaje puede saltar.
- *Scroll* horizontal.
- Puede o no existir profundidad (2D).
- Objetos a recoger.
- Enemigos a eliminar que complican la existencia al jugador.
- Plataformas donde saltar.

Con ciertas variaciones, se puede considerar un juego de plataformas aquel que simplemente contenga la mayoría de estas características, de manera que ninguna de ellas sea completamente imprescindible. Es uno de los géneros más versátiles del mundo del videojuego [12].

1.3.3 Necesidad de utilizar una arquitectura de software en videojuegos

La arquitectura de software ha adquirido especial importancia en el desarrollo de videojuegos, ya que la manera en que se estructura un videojuego tiene un impacto directo sobre la capacidad del mismo para satisfacer lo que se conoce como atributos de calidad del sistema. Una de las múltiples estructuras que componen la AS se enfoca en partir el sistema en componentes que serán desarrollados por individuos o grupos de individuos. La identificación de esta estructura de asignación de trabajo es esencial para apoyar las tareas de planeación del proyecto.

Los diseños arquitectónicos que se crean en una organización pueden ser reutilizados para crear sistemas distintos. Esto permite reducir costos y aumentar la calidad, sobre todo si dichos diseños han resultado previamente en sistemas exitosos [14].

Desde hace unos años los desarrollos de videojuegos se hacen planificando por adelantado una arquitectura de software. En 1999, Rollings y Morris [15] comentaban que era muy habitual que los

estudios de videojuegos saltaran desde el diseño del videojuego a su implementación directamente, sin pasar por un periodo previo de maduración de la arquitectura de software que lo iba a sustentar. Con ese método de trabajo, el éxito o fracaso del desarrollo dependía casi por completo de la habilidad y nivel de experiencia de los desarrolladores.

Hoy en día la situación ha mejorado significativamente. Es difícil crear un videojuego que necesite para su desarrollo el uso de grandes cantidades de recursos y personal en menos de dos años. Con estos largos desarrollos en los que se ven involucradas decenas o cientos de personas, la ingeniería de software resulta completamente necesaria y, por consiguiente, la realización de una arquitectura también.

Aun así, en muchos casos la arquitectura planeada queda restringida a un único título, y el estudio vuelve a diseñar una gran parte de ella para el siguiente desarrollo, incluso perteneciendo a géneros similares o que tengan funcionalidades que podrían ser reutilizadas.

1.3.4 Motivación para definir una AS para videojuegos

Una buena arquitectura/diseño permite en los videojuegos [16]:

1. Reusabilidad.
2. Extensibilidad.
3. Manejabilidad.
4. Acortar tiempos de producción.
5. Reducir gastos de recursos.
6. Reducir tiempo necesario para su creación.

1.4 Desarrollo de Videojuegos

El videojuego como una de las incipientes industrias culturales se caracteriza en su diseño por la confluencia de dos grandes ámbitos de creación, el audiovisual y la informática. Sus peculiaridades como producto hacen que sea necesario el tránsito por unas fases concretas y determinantes de producción.

El desarrollo del juego, a lo largo de su ciclo de vida, se puede asemejar al de una película de cine, pudiéndose segmentar en tres fases ampliamente diferenciadas: Pre-Producción, Producción y Post-producción, cada una con sus etapas características [16], a pesar de que son las propias compañías las que fijan cuál será su filosofía de trabajo a lo largo de la creación de un juego.

Un videojuego no sólo es un producto artístico, debe de pasar por varias fases desde que es concebido hasta que es olvidado, es decir, que como todo *software*, tiene un ciclo de vida. El ciclo de vida da la pauta a lo que hay que obtener a lo largo del desarrollo del juego más no el cómo desarrollarlo, de eso se encarga el proceso de desarrollo de *software*. Dada la complejidad de un

videojuego, dicho proyecto debe manejarse a través de un proceso, involucrando así una serie de pasos organizados que guiarán cada actividad hasta el producto final [16].

1.4.3 Diseño de videojuegos

Diseño del videojuego es el momento en que se especifican los elementos que compondrán el juego, dando una idea clara a todos los miembros del grupo desarrollador de cómo éstos son. Se diseñan en profundidad todos los aspectos del videojuego y que en fase de preproducción únicamente se habían perfilado [16]. Se puede dividir en diseño artístico y mecánico:

- El apartado de **diseño artístico** trabaja elementos que tienen que ver con la "apariencia" del juego en sí, contenidos que van a estar plasmados evidentemente en el "físico" del videojuego como historia, sonido, interfaz y gráficos. Con estos cuatro elementos, se compendian todos los aspectos relacionados con la "forma" artística del videojuego que lo harán diferente a otros en el mercado.
- En la fase de **diseño mecánico** se marcan las pautas de interacción con el juego, las normas internas y el tipo de comunicación que debe darse en caso de que el destino del juego sea el entorno *on line* (en línea o conectado a la red). Los detalles cobran vida en este punto con el diseño de reacciones y comportamientos que darán vida a cada uno de los personajes.

El proceso de diseño de un videojuego se encuentra contenido en el Documento de Diseño, que es un artefacto que genera el proceso de desarrollo de un videojuego, se acarrea a lo largo del proyecto y deberá estar dispuesto a sufrir varios cambios desde la etapa de revisión. Contiene todas las especificaciones necesarias para comenzar el proyecto, éstas van desde el tema principal del videojuego hasta el número de niveles que tendrá. El diseñador del juego asienta la idea a este documento con información detallada del proyecto, desde el título del juego, el género, una visión general y mecánica, aspectos de jugabilidad, modos de juego, plataforma o *software* que se utilizará [16].

1.4.4 Motores de Videojuegos

En los últimos años, con la aparición de los dispositivos móviles como es el caso de los *smartphones* o las tabletas, han abierto un gran camino en el mercado de los videojuegos: aplicaciones para estos dispositivos. Al mismo tiempo, las herramientas como los motores de videojuego que se han ido aproximando a los desarrolladores, han permitido satisfacer esta demanda con un incremento en el número y variedad de ventajas al utilizarlos. Algunas de esas **ventajas** son [17]:

- Ahorro de código, puesto que en muchos casos te dan prácticamente hecho ciertas partes del mismo.

- El desarrollador se preocupa menos por temas como el manejo de memoria, carga de recursos, iluminación y renderizado de la escena. Todo esto ha sido ya diseñado y testeado por la compañía que hizo el motor.
- Generalmente, los proyectos se pueden compilar para varias plataformas (*cross platform*), así que ahorran tener que hacer *ports* (Colección de Puertos o Árboles de Puertos).

Por supuesto, también tienen sus contras:

- Para cada motor, hay que familiarizarse con la forma de desarrollar código, ya que probablemente no utilicen el mismo lenguaje.
- Si hay un *bug* en el motor, a no ser que sea código abierto, no se podrá arreglar.
- Generalmente, no suelen ser gratis o suelen tener limitaciones que se subsanan pagando.

Algunos de los motores más importantes en la actualidad son: Unity 3D, Unreal Engine y CryEngine.

Para la propuesta de solución se usará Unity 3D dadas sus características principales es el motor usado para desarrollar videojuegos en el centro, y la propuesta de solución usa la arquitectura propia de dicho motor.

Unity 3D

Unity 3D es un potente motor *cross-platform*, para realizar proyectos tanto 2D como 3D con un entorno de desarrollo muy ameno. Fácil de manejar para los principiantes y suficientemente potente para los expertos, permite crear fácilmente videojuegos y aplicaciones para multitud de plataformas. Posee las *stores* (tiendas), que son bibliotecas de *assets* (activos) comerciales y gratuitos creados por *Unity Technologies* y miembros de la comunidad. Hay una gran cantidad de *assets* disponibles, desde texturas, modelos y animaciones hasta ejemplos de proyectos completos, tutoriales y extensiones del editor. Son accesibles desde una interfaz simple dentro del Editor Unity y son descargados e importados directamente en sus proyectos.

Ventajas [17]:

- Soporta oficialmente dos tipos de lenguaje de programación: C#, UnityScript (básicamente Javascript). El más generalizado en la comunidad es C#.
- Facilidad para realizar juegos tanto 2D como 3D.
- Curva de aprendizaje sencilla. Posee el *Asset Store*, que es una gran comunidad de creadores de plugins y recursos, así como muchos manuales oficiales para aprender rápido.
- Las herramientas de edición gráfica son muy buenas y pueden ser mejoradas con plugins.
- Soporta multitud de formatos de recursos y los convierte de manera automática a los formatos más óptimos dependiendo de la plataforma objetivo.

- Despliegue a múltiples plataformas de manera sencilla, tanto móviles como de escritorio y consola. Desde Unity 5.0, la licencia gratuita cubre prácticamente todas las necesidades del desarrollador, y algunas de las licencias de pago son bastante asequibles.

1.5 Vistas arquitectónicas

Desde los inicios de la arquitectura de software se planteó la necesidad de contar con varias vistas para separar a los diferentes elementos del *software*. El arquitecto del *software* necesita un número de vistas diferentes de la arquitectura de software para varios usos y usuarios, las que son requeridas para enfatizar y entender diferentes aspectos de la arquitectura… [9]. Existen diferentes definiciones para el término vista:

“Un subconjunto resultante de practicar una selección o abstracción sobre una realidad, desde un punto de vista determinado” [9].

De acuerdo a la definición del estándar 1471 de la IEEE, es lo siguiente: “Una vista es una representación de un sistema completo desde la perspectiva de un conjunto de *concerns* relacionados” [10]. En conclusión, una vista no es más que una temática desde la que se enfoca el proyecto, ya sea desde el punto de vista de los datos, integración, entre otros y que hace posible enfrentar el diseño de la arquitectura desde diversas perspectivas reduciendo la complejidad presente en su desarrollo.

Las vistas arquitectónicas constituyen las estructuras fundamentales de la arquitectura. El diseño de las vistas arquitectónicas implica establecer una sincronización entre sus elementos para conformar una arquitectura integrada, y de poder ser capaz de mantener la consistencia entre sus elementos. Cada vista agrupa a elementos que pertenecen a diferentes intereses. Además, se llegan a establecer relaciones de correspondencia entre los elementos de distintas vistas, puesto que al formar parte de un mismo sistema, de alguna manera se llegan a relacionar. Estas relaciones de correspondencia constituyen el adhesivo que mantiene el enlace a las vistas que integran a las AS [18].

Las AS se enfrentan a la abstracción, descomposición y composición usando estilos y patrones. Para describir una arquitectura de software se usa un modelo compuesto de múltiples vistas o perspectivas. Con el fin de resumir estos modelos, Philippe Kruchten propone cinco vistas [19]:

- **La vista lógica:** es el objeto del diseño (cuando se usa el modelo orientado a objetos).
- **La vista de procesos:** captura la concurrencia y sincronización como aspectos del diseño.
- **La vista física:** describe el mapeo del *software* en el *hardware* y refleja su aspecto distribuido.
- **La vista de desarrollo:** describe la organización estática del *software* en su entorno de desarrollo.

- **Escenarios:** donde se ensamblan los elementos mostrados en las vistas anteriores.
  La descripción de una arquitectura (las decisiones tomadas) puede ser organizada alrededor de las vistas lógica, de procesos, física y de desarrollo, mostrándose en los escenarios. La arquitectura, de hecho, evoluciona a partir de estos escenarios.

Por otra parte, Robert L. Nord y otros autores realizaron un estudio para conocer en una arquitectura, las estructuras que son de mayor importancia y el uso de éstas. El estudio se efectuó sobre varios sistemas de *software* de ámbito industrial tales como, sistemas de procesamiento de señales e imágenes, sistemas operativos en tiempo real, sistemas de comunicaciones, y sistemas de control e instrumentación. Tras el estudio realizado, Nord y colegas propusieron cuatro categorías o "vistas" para agrupar las estructuras principales de una arquitectura. Estas son, vista conceptual, vista de interconexión de módulos, vista de ejecución, y vista de código. Dentro de cada una, se describen las estructuras principales del sistema desde una perspectiva en particular. A continuación, se explica brevemente cada una de éstas [20]:

- **Vista conceptual:** Se describe el sistema en términos de sus elementos principales de diseño y las relaciones entre éstos, dentro de un dominio determinado. Esta vista es independiente de las decisiones de implementación y enfatiza en los protocolos de interacción entre los elementos de diseño.
- **Vista de módulos:** Se captura la descomposición funcional y las capas del sistema. El sistema es descompuesto lógicamente en subsistemas, módulos, y unidades abstractas. Cada capa representa las distintas interfaces de comunicación permitidas entre los módulos.
- **Vista de ejecución:** Se describe la estructura dinámica del sistema en términos de sus elementos en tiempo de ejecución. Por ejemplo, se modelan las tareas operativas del sistema, procesos, mecanismos de comunicación y asignación de recursos. Algunos de los aspectos que se consideran en esta vista son el desempeño y el entorno de ejecución.
- **Vista de código:** Se organiza el código fuente en directorios, archivos y bibliotecas. Algunos de los aspectos que se incluyen son: los lenguajes de programación a utilizar, herramientas de desarrollo, la administración de la configuración, la estructura y organización del proyecto.

Para el presente trabajo podrían seleccionarse modelos de vistas tales como 4+1 de Kruchten pero por las características que los videojuegos presentan, describirlos a través de una especificación de requisitos y casos de uso se hace complejo, pues genera una gran cantidad de diagramas de casos de uso para detallar los procesos y sin embargo, hay detalles propios del videojuego que no se encuentran contemplados en los mismos como elementos formales, elementos dramáticos y metas para la experiencia del jugador. Además, resulta complicado describir las características dinámicas e inteligentes que poseen estos juegos.

Por lo anteriormente planteado, el enfoque de Robert Nord fue seleccionado para describir el prototipo funcional de la presente investigación.

1.6 Estilos arquitectónicos

La arquitectura de software se encarga del diseño e implementación de estructuras de alto nivel de *software*. Es el resultado de unir cierto número de elementos de arquitectura en formas que satisfagan los requisitos funcionales y de rendimiento del sistema, así como otros requisitos no funcionales como confiabilidad, escalabilidad, portabilidad y disponibilidad [15].

Un estilo describe una clase de arquitectura, o piezas identificables de las arquitecturas empíricamente dadas. Esas piezas se encuentran repetidamente en la práctica, trasuntando la existencia de decisiones estructurales coherentes. Una vez que se han identificado los estilos, es lógico y natural pensar en reutilizarlos en situaciones semejantes que se presenten en el futuro [18].

Mary Shaw y Paul Clements [21] identifican los estilos arquitectónicos como un conjunto de reglas de diseño que identifica las clases de componentes y conectores que se pueden utilizar para componer en sistema o subsistema, junto con las restricciones locales o globales de la forma en que la composición se lleva a cabo. Los componentes, incluyendo los subsistemas encapsulados, se pueden distinguir por la naturaleza de su computación. Si retienen estado entre una invocación y otra, y de ser así, si ese estado es público para otros componentes. Los tipos de componentes también se pueden distinguir conforme a su forma de empaquetado, o dicho de otro modo, de acuerdo con las formas en que interactúan con otros componentes.

Mark Klein y Rick Kazman proponen una definición según la cual un estilo arquitectónico es una descripción del patrón de los datos y la interacción de control entre los componentes, ligada a una descripción informal de los beneficios e inconvenientes aparejados por el uso del estilo. Los estilos arquitectónicos, afirman, son artefactos de ingeniería importantes porque definen clases de diseño junto con las propiedades conocidas asociadas a ellos. Ofrecen evidencia basada en la experiencia sobre la forma en que se ha utilizado históricamente cada clase, junto con razonamiento cualitativo para explicar por qué cada clase tiene esas propiedades específicas [22].

Los estilos arquitectónicos sirven para sintetizar estructuras de soluciones de organización a nivel de sistema, expresando una estructura coherente, basada en elementos definidos y probados en diversos escenarios. Dependen del contexto del sistema que se vaya a desarrollar para decantarse por la utilización de uno u otro, pues proponen soluciones de manera adecuada a diferentes situaciones, pero algunos se ajustan más que otros al problema planteado.

1.6.1 Categorías de estilos arquitectónicos

Una posible categorización de los estilos arquitectónicos es, teniendo en cuenta la descripción de Carlos Reynoso [14]:

Estilos de Flujo de Datos

- Tubería y filtros.

Estilos Centrados en Datos

- Arquitecturas de Pizarra o Repositorio.

Estilos de Código Móvil

- Arquitectura de Máquinas Virtuales.

Estilos Peer-to-Peer

- Arquitecturas Basadas en Eventos.
- Arquitecturas Orientadas a Servicios (SOA).
- Arquitecturas Basadas en Recursos.

Estilos de Llamada y Retorno

- Modelo-Vista-Controlador (*Model-View-Controller*).
- Arquitecturas en Capas.
- Arquitecturas Orientadas a Objetos.
- Arquitecturas Basadas en Componentes.

Estilos Derivados

- C2.
- GenVoca.
- Rest.

Estilos Heterogéneos

- Sistema de Control de Procesos.
- Arquitecturas Basadas en Atributos.

1.7 Patrones arquitectónicos

Arquitectura en Capas

Garlan y Shaw definen la arquitectura en capas (Figura 1) como una organización jerárquica, tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior. Los conectores se definen mediante los protocolos que determinan las formas de la interacción. Las restricciones topológicas pueden incluir una limitación, más o menos rigurosa, que exige a cada capa operar sólo con capas adyacentes, y a los elementos de una capa entenderse sólo con otros elementos de la misma. Es de suponer que si esta exigencia

se relaja, el estilo deja de ser puro y pierde algo de su capacidad heurística [7]. También se pierde, naturalmente, la posibilidad de reemplazar totalmente una capa sin afectar a las restantes, disminuye la flexibilidad del conjunto y se complica su mantenimiento. En muchas ocasiones se sacrifica la pureza de la arquitectura en capas precisamente para mejorarla, colocando, por ejemplo: reglas de negocios en los procedimientos almacenados de las bases de datos o articulando instrucciones de consulta en la capa de la interfaz de usuario.

Ventajas:

- Modularidad.
- Soporta un diseño basado en niveles de abstracción crecientes, lo cual a su vez permite a los implementadores la partición de un problema complejo en una secuencia de pasos incrementales.
- Proporciona amplia reutilización.
- Soporta fácilmente la evolución del sistema; los cambios solo afectan a las capas vecinas. Se pueden cambiar las implementaciones respetando las interfaces con las capas adyacentes.

Desventajas:

- Es difícil razonar a priori sobre la separación en capas requeridas.
- Formatos, protocolos y transportes de la comunicación entre capas suelen ser específicos y propietarios.
- No todos los sistemas pueden estructurarse en capas.

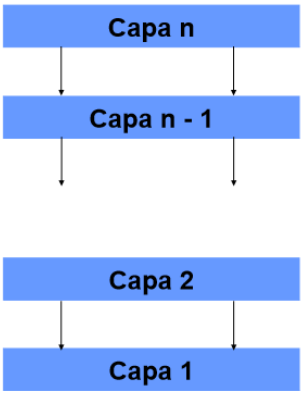


Figura 1: Ejemplo de arquitectura en capas.

Arquitectura basada en componentes

Hay un buen número de definiciones de componentes, pero Clemens Alden Szyperski propone: “un componente de software es una unidad de composición con interfaces especificadas contractualmente y dependencias del contexto explícitas”.

Las interfaces están separadas de las implementaciones, y las interfaces y sus interacciones son el centro de incumbencias en el diseño arquitectónico. Los componentes soportan algún régimen de introspección, de modo que su funcionalidad y propiedades puedan ser descubiertas y utilizadas en tiempo de ejecución. En cuanto a las restricciones, puede admitirse que una interfaz sea implementada por múltiples componentes. Usualmente, los estados de un componente no son accesibles desde el exterior [23].

Ventajas:

- Reutilización del software: Lleva a alcanzar un mayor nivel de reutilización.
- Simplifica las pruebas: Permite que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo de componentes ensamblados.
- Simplifica el mantenimiento del sistema: Cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema.
- Mayor calidad: Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo.

Desventajas:

- Si no existen los componentes, hay que desarrollarlos y se puede perder mucho tiempo, por otro lado, estos componentes pueden tener conflictos si de estos sale una nueva versión, por lo que es posible que haya que re-implementarlos.

1.8 Patrones de diseño

Existen, además de los estilos arquitectónicos y las vistas de una arquitectura, los patrones, que describen situaciones a las que se les ha dado solución, para poder usarse en oportunidades posteriores sin tener que volver a plantearse la misma problemática.

Como un elemento en el mundo, cada patrón es una relación entre cierto contexto, cierto sistema de fuerzas que ocurre repetidas veces en ese contexto y cierta configuración espacial que permite que esas fuerzas se resuelvan. Como un elemento de lenguaje, un patrón es una instrucción que muestra la forma en que esta configuración espacial puede usarse, una y otra vez, para resolver ese sistema de fuerzas, donde quiera que el contexto la torne relevante El patrón es, en suma, al mismo tiempo una cosa que pasa en el mundo y la regla que nos dice cómo crear esa cosa y cuándo debemos crearla. Es tanto un proceso como una cosa; tanto una descripción de una cosa que está viva como una descripción del proceso que generará esa cosa [24].

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe también el núcleo de su solución, de forma que puede utilizarse un millón de veces sin hacer dos veces lo mismo [25].

Los patrones ayudan a estandarizar el código, haciendo que el diseño sea más comprensible para otros programadores y solucionando problemas de reusabilidad y optimización. Ayudan a cumplir con muchas buenas prácticas a la hora de programar y con reglas del correcto diseño. Se pueden adaptar a la variante particular del problema que se desee aplicar, sin embargo, esto no quiere decir que sea óptimo usarlos en todos los escenarios. Siempre existen patrones que dependen de ciertas situaciones para ser usados con efectividad.

1.8.1 Tipos de patrones GOF

Los patrones GoF se descubren como una forma indispensable de enfrentarse a la programación a raíz del libro "*Design Patterns—Elements of Reusable Software*" de Erich Gamma, Richard Helm, Ralph Jonson y John Vlissides, a partir de entonces estos patrones son conocidos como los patrones de la pandilla de los cuatro (GoF, *Gang of Four*) [26].

Una posible agrupación de los patrones según su tipo podría ser [26]:

- **Patrones de creación:** Ofrecen mecanismos de creación de instancias de objetos y estructuras escalables dependiendo de las necesidades. Proporcionan una solución relacionada con la construcción de clases, objetos y otras estructuras de datos.
- **Patrones estructurales:** Tratan sobre la forma de organizar las jerarquías de clases, las relaciones y las diferentes composiciones entre objetos para obtener un buen diseño bajo un determinado contexto.
- **Patrones de comportamiento:** Las soluciones de diseño que proporcionan los patrones de comportamiento están orientadas al envío de mensajes entre objetos y cómo organizar ejecuciones de diferentes métodos para conseguir realizar algún tipo de tarea de forma más conveniente.

Patrones de creación

- *Object Pool*: Se obtienen objetos nuevos a través de la clonación. Utilizado cuando el costo de crear una clase es mayor que el de clonarla. Especialmente con objetos muy complejos.
- *Abstract Factory* (fábrica abstracta): Permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando.

- *Builder* (constructor virtual): Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto
- *Factory Method* (método de fabricación): Centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística para elegir el subtipo que crear.
- *Prototype* (prototipo): Crea nuevos objetos clonándolos de una instancia ya existente
- *Singleton* (instancia única): Garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia.

Patrones estructurales

- *Adapter* (Adaptador): Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
- *Bridge* (Puente): Desacopla una abstracción de su implementación.
- *Composite* (Objeto compuesto): Permite tratar objetos compuestos como si se tratase de uno solo.
- *Decorator* (Decorador): Añade funcionalidades a una clase dinámicamente
- *Facade* (Fachada): Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema.
- *Flyweight* (Peso ligero): Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.
- *Proxy*: Mantiene un representante de un objeto.
- Módulo: Agrupa varios elementos relacionados, como clases, *singletons*, y métodos, utilizados globalmente, en una entidad única.
- Modelo Vista Controlador (*Model-View-Controller*: MVC): separa los datos de una aplicación, la interfaz de usuario, y la lógica de negocio en tres componentes distintos.

Patrones de comportamiento

- *Chain of Responsibility* (Cadena de responsabilidad): Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.
- *Command* (Orden): Encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma.
- *Interpreter* (Intérprete): Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas necesarias para interpretarlo.

- *Iterator* (Iterador): Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.
- *Mediator* (Mediador): Define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto.
- *Memento* (Recuerdo): Permite volver a estados anteriores del sistema.
- *Observer* (Observador): Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.
- *State* (Estado): Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
- *Strategy* (Estrategia): Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución.
- *Template Method* (Método plantilla): Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, permitiendo que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
- *Visitor* (Visitante): Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.

1.8.5 Patrones GRASP

En diseño orientado a objetos, GRASP son patrones generales de *software* para asignación de responsabilidades, es el acrónimo de "GRASP (*object-oriented design General Responsibility Assignment Software Patterns*)". Aunque se considera que más que patrones propiamente dichos, son una serie de "buenas prácticas" de aplicación recomendable en el diseño de *software*. Los patrones GRASP son [27]:

- Experto en información: Es el principio básico de asignación de responsabilidades. Nos indica, por ejemplo, que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo.
- Creador: Se encarga de identificar quién debe ser el responsable de la creación (o instanciación) de nuevos objetos o clases.
- Controlador: El patrón controlador es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado.

- Alta cohesión: Nos dice que la información que almacena una clase debe de ser coherente y debe estar (en la medida de lo posible) relacionada con la clase.
- Bajo acoplamiento: Es la idea de tener las clases lo menos ligadas entre sí que se pueda. De tal forma que en caso de producirse una modificación en alguna de ellas, se tenga la mínima repercusión posible en el resto de clases
- Polimorfismo. Se usa cuando las alternativas o comportamientos relacionados varían según el tipo (clase), asignando la responsabilidad para el comportamiento a los tipos para los que varía el comportamiento.
- Fabricación pura: La fabricación pura se da en las clases que no representan un ente u objeto real del dominio del problema, sino que se ha creado intencionadamente para disminuir el acoplamiento, aumentar la cohesión y/o potenciar la reutilización del código.
- Indirección: El patrón de indirección nos aporta mejorar el bajo acoplamiento entre dos clases asignando la responsabilidad de la mediación entre ellos a un tercer elemento (clase) intermedio.
- Variaciones Protegidas: Es el principio fundamental de protegerse del cambio, de tal forma que todo lo que preveamos en un análisis previo que es susceptible de modificaciones, lo envolvamos en una interfaz, utilizando el polimorfismo para crear varias implementaciones y posibilitar implementaciones futuras.

1.8.6 Objetivos de los patrones de diseño

El paradigma de diseño y desarrollo basado en patrones es ampliamente utilizado actualmente, especialmente en *software* de gran complejidad. La utilización de este paradigma se aplica en el diseño de las arquitecturas de software y los motores de juego.

Los patrones de diseño pretenden [28]:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas *software*.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no tienen como objetivo:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.

1.9 Arquitecturas de software para el desarrollo de videojuegos

Al igual que ocurre en otras disciplinas en el campo de la informática, el desarrollo de videojuegos se ha beneficiado de la aparición de herramientas que facilitan dicho desarrollo, automatizando determinadas tareas y ocultando la complejidad inherente a muchos procesos de bajo nivel. Las arquitecturas de software hacen la vida más sencilla a los desarrolladores de videojuegos debido a que [29]:

- El proceso de diseño de cualquier aplicación *software* suele ser iterativo y en diferentes etapas de forma que se vaya refinando con el tiempo (existen infinidad de modelos de desarrollo que pueden aplicarse)
- La etapa de diseño es una tarea crítica en el ciclo de vida del software (gran impacto sobre el producto final y el futuro del mismo La experiencia previa en el diseño de sistemas similares en útil para crear buenos diseños, pero no suficiente.
- Al construir aplicaciones similares se presentan situaciones recurrentes y que se asemejan a situaciones pasadas

Básicamente, cuando un juego contiene parte de su lógica o funcionamiento en el propio código (*hard-coded logic*), entonces no resulta práctico reutilizarla para otro juego, ya que implicaría modificar el código fuente sustancialmente. Sin embargo, si dicha lógica o comportamiento no está definido a nivel de código, sino por ejemplo mediante una serie de reglas específicas a través de un lenguaje de *script*, entonces la reutilización sí es posible, ya que optimiza el tiempo de desarrollo [29].

1.9.1 Ejemplos de arquitecturas para videojuegos

Un ejemplo que se podría citar como referencia es la **Arquitectura para Videojuegos Serios con Aspectos Culturales**, publicada por Ricardo Emmanuel Gutiérrez Hernández, Francisco Álvarez Rodríguez y Jaime Muñoz [30], de la Universidad Autónoma de Aguascalientes en México, que consiste en una arquitectura en 6 capas como se detalla en la Figura 3.

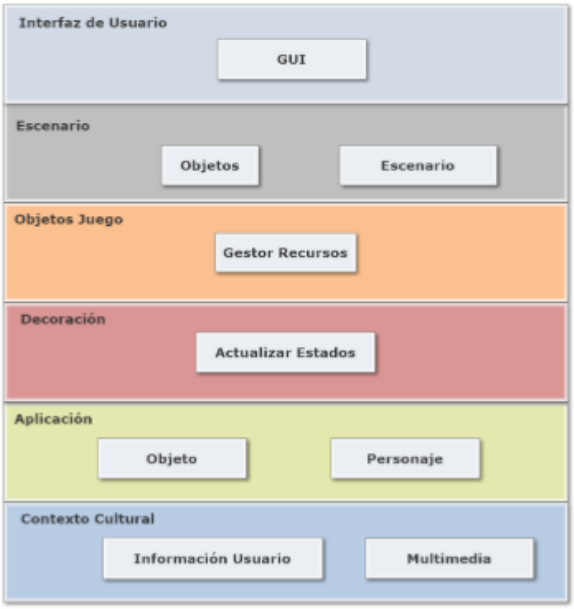


Figura 2: Arquitectura para Videojuegos Serios con Aspectos Culturales

A continuación, se especifican los niveles de la arquitectura:

- La Capa **Contexto Cultural**: se encuentra la información del usuario y recursos de juego donde se agrupan todos los datos de usuario y archivos multimedia para aplicarlos en el videojuego.
- En el nivel de **Aplicación**: se encarga de especificar los elementos genéricos del videojuego, en base a la lógica y los recursos disponibles, respondiendo y proporcionando una interfaz para administrar eventos y retroalimentación al usuario.
- La capa de **Decoración**: es responsable de actualizar los estados de los elementos, observando la interacción del usuario usando patrones de diseño, centrándose en los aspectos culturales visibles como los colores y códigos de caracteres, en este nivel es donde se adaptan las características observables de la cultural para los objetos del videojuego.
- En la capa de **Objetos de Juego**: se integran los módulos y objetos responsables de ofrecer las características propias del juego, gestionando los recursos lógicos y las vistas entre los componentes y paquetes del videojuego para un escenario.
- En el nivel de **Escenario**: se integran las clases relativas al correcto funcionamiento interno del videojuego como los objetos, multimedia, reglas de interacción, así como las vistas consideradas en la clase general del objeto, esta capa proporciona la simulación en tiempo real la variedad de elementos de objetos y personajes del videojuego.

- En el nivel de **Interfaz de Usuario**: se muestran el conjunto de escenarios, objetos del videojuego y se procesan los datos de entrada del usuario para realizar las diferentes interacciones en la historia o niveles de juego del contexto cultural del usuario/jugador.

El estudio de la organización y estructura que proponen en “Arquitectura para Videojuegos Serios con Contextos Culturales” brinda una mayor visión de cómo estructurar un sistema en capas, definiendo los elementos que conforman cada una de ellas. El uso de este estilo arquitectónico simplifica el uso y la comunicación entre capas.

Otro ejemplo más cercano a la situación problemática en cuestión sería la tesis que lleva como título **“Diseño y Desarrollo de un Prototipo Básico de un Videojuego de Plataformas en 2D”** publicada por Carlota Esteban Cazalla [31], que se acerca más al problema en cuestión. Para esta tesis se usó como motor Unity 3D y una arquitectura Modelo-Vista-Controlador (MVC).

Modelo: Es la representación de los datos con los que interviene el sistema. En este caso, todas las clases que representen una entidad en el videojuego (personaje, enemigo o plataformas).

Vista: Es la presentación del modelo. En la arquitectura, son todas aquellas clases que cambian de manera visual los modelos o generan sonidos (animaciones del personaje, animaciones de enemigos, partículas de cambio de mundo y sonidos).

Controlador: Los controladores responden a eventos (inputs, selecciones en el menú, eventos enviados por colisiones y disparadores) y modifican los modelos y vistas en base a estos eventos. Existen diversos controladores en el videojuego (controladores de nivel, controladores de personaje o controladores de cambio de mundo).

El estilo MVC permite separar los datos de la lógica del videojuego. Sin embargo, desacoplar el modelo de la vista no significa que los desarrolladores del modelo puedan ignorar la naturaleza de las vistas. Si el modelo experimenta cambios frecuentes, podrían desbordarse las vistas con una lluvia de requerimientos de actualización, lo que trae consigo que no cumpla con el atributo de calidad mantenibilidad en proyectos con cambios frecuentes en los requisitos.

1.9.2 Ejemplos de arquitecturas de software para videojuegos usadas en el centro

En el centro Vertex se han desarrollado videojuegos de diferentes géneros, contando con equipos de diversos programadores y arquitectos de software, que en la mayoría de los casos no coinciden, por lo que sus criterios a la hora de definir una AS para el videojuego difieren. Esto, sumado a que en muchos casos no se contaba con un estudio del arte referente al tema, ha traído consigo desorganización y dio origen a la necesidad de proponer una AS que satisfaga la situación problemática antes descrita.

Especies Invasoras:

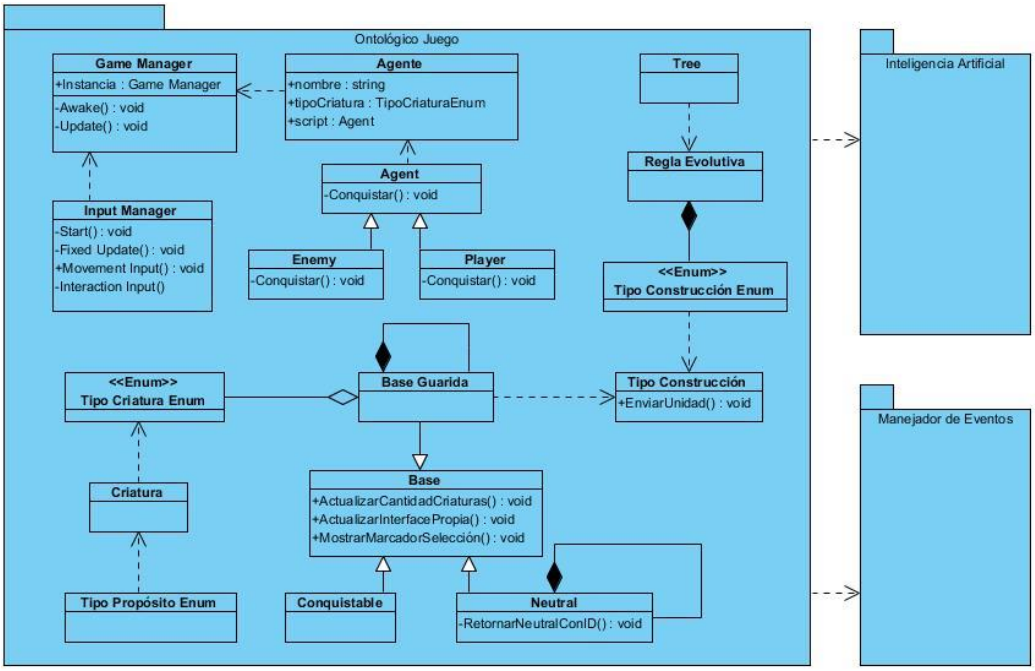


Figura 3: AS del videojuego Especies Invasoras

En el videojuego Especies Invasoras se usó una arquitectura en 3 capas, funcionando de la siguiente manera:

Ontológico Juego: En esta capa se encuentra la lógica del videojuego, junto a las clases principales, cuya comunicación se evidencia de la manera planteada en la Figura 3. En el *Game Manager* se implementó el patrón de diseño *singleton,* de manera que se tiene en el videojuego una única instancia de dicha clase para usarse cuando se requiera. Se puede apreciar también cómo heredan atributos y métodos las clases Conquistable, Neutral y Base Guardia de Base y *Enemy* y *Player* de Agent respectivamente.

Manejador de Eventos: En esta capa se implementó el patrón de diseño *Event Handler,* estando suscritas algunas de las clases del juego al manejador de eventos y controlando los mismos (el orden entre ellos) mediante una lista, debido a que el videojuego contaba con demasiados estados como para ser controlado con eficiencia mediante consultas simples.

Inteligencia Artificial: En esta capa se implementó la lógica de la inteligencia artificial y el comportamiento tanto de los personajes como los enemigos.

De la AS usada en el videojuego “Especies Invasoras” buenas prácticas observadas que se pueden incorporar a la propuesta de solución son:

- La herencia de algunas de sus clases, que garantiza reutilización y extensibilidad

- El polimorfismo observado en métodos como Conquistar(), que permite separar los elementos que cambian de los que no lo hacen. El método mencionado se implementa de diversas formas, facilitando la ampliación, el mantenimiento y la reutilización del código.
- El uso del patrón de diseño *singleton* en el *Game Manager*, que garantiza que haya una sola instancia de dicha clase, reduciendo la carga en memoria de datos y optimizando el código.
- El uso del manejador de eventos (*Event Handler*), lo que posibilita optimizar el control de acciones en un juego que tiene demasiados estados.

Caos Numérico:

Como solución a la estructura del videojuego Caos Numérico se usó una arquitectura por capas, definida de la forma que se plantea en la Figura 4.

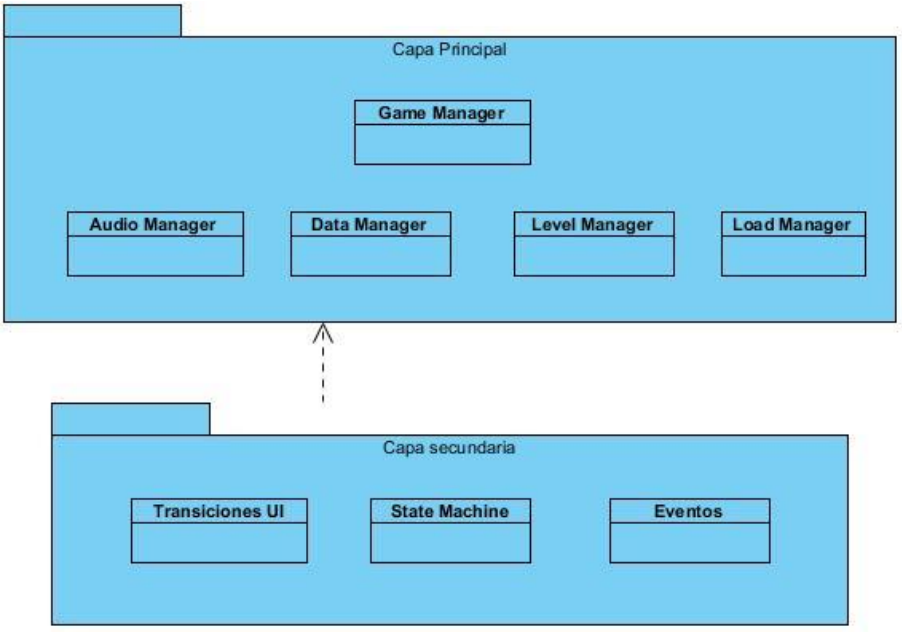


Figura 4: AS Caos Numérico

En la capa principal se encuentran los managers (***Game Manager, Data Manager, Audio Manager, Level Manager, Load Manager***), cada uno de ellos tiene un patrón Singleton (una única instancia) implementado para comunicarse entre ellos y con la capa inferior.

En la capa secundaria se encuentran las clases:

Transiciones UI: Se encarga de las transiciones de la interfaz de usuario.

***State Machine*:** Se encarga de los diferentes estados que puede tener la escena principal, que es en la que se implementó la máquina de estados.

Eventos: Usa tipo de datos *delegate* o delegado (propio de C#), representando métodos con una lista de parámetros determinada y un tipo de valor devuelto. Se usa para llamar a los métodos a través de la instancia del delegado.

De la AS usada en el videojuego “Caos Numérico” unas buenas prácticas observadas que se pueden incorporar a la propuesta de solución son:

- El uso del tipo de datos delegado para llamar a los métodos a través de instancias de ese tipo.
- El uso del patrón de diseño *Singleton* en cada uno de los managers (*Game Manager, Audio Manager, Data Manager, Level Manager, Load Manager*).
- La organización y estructuración del sistema en capas.

A pesar de estas buenas prácticas cabe señalar que en la capa principal pudo implementarse el patrón de diseño fachada en el *Game Manager*. El objetivo de usarlo sería mantener una sola instancia de dicha clase y no de todas las que se accederá desde el Game Manager (*Audio Manager, Data Manager, Level Manager, Load Manager*).

1.10 Conclusiones parciales

El análisis realizado permitió determinar la relevancia que tiene la arquitectura de software para el desarrollo de videojuegos en el mundo, y en especial para el centro Vertex. Se estudiaron estilos arquitectónicos con el objetivo de formar una visión de la arquitectura, así como patrones de diseño con el fin de usarlos en la propuesta de solución. Se abordó el tema de los diferentes tipos de videojuegos, de las AS usadas en los mismos, y las estructuras que fueron usadas en los videojuegos desarrollados en el centro, para proponer una solución que cumpla con los requerimientos necesarios para ser usada para futuros desarrollos. El estudio de las AS usadas en diferentes videojuegos permitió determinar las características arquitectónicas que pueden ser reutilizables en la propuesta de solución.

Capítulo #2. Solución propuesta

2.1 Introducción

En este capítulo se plantea la arquitectura propuesta. Se especifica la metodología de *software* con la que se describe la AS. También se especifican las herramientas que se utilizarán en el desarrollo del prototipo funcional, así como se representan ingenierilmente los elementos que proporcionan una base para el entendimiento de la AS propuesta.

2.2 Metodologías y herramientas de desarrollo

Para describir los principales elementos ingenieriles del prototipo funcional se usó como metodología de desarrollo AUP -vUCI (*Agile Unified Process* versión UCI). Para visualizar de una manera más precisa tanto el prototipo funcional como la AS, se empleó el lenguaje de modelado UML (*Unified Modeling Language*), soportado por la herramienta de modelado *Visual Paradigm for UML 8.0.* El motor de videojuegos empleado fue Unity 3D, usando el lenguaje de programación C#.

2.2.1 Metodologías de desarrollo de software AUP-vUCI

Al no existir una metodología de *software* universal, ya que toda metodología debe ser adaptada a las características de cada proyecto (equipo de desarrollo, recursos, personal) exigiéndose así que el proceso sea configurable, se decide hacer uso de la metodología AUP –vUCI [32].

A partir de que el modelado de negocio de AUP propone tres variantes a utilizar en los proyectos (CUN [Casos de Uso del Negocio], DPN [Diagrama de Procesos del Negocio] o MC [Modelo Conceptual] y existen tres formas de encapsular los requisitos (CUS [Casos de Uso del Sistema], HU [Historias de Usuario], DRP), surgen cuatro escenarios para modelar el sistema, manteniendo en dos de ellos el MC, quedando de la siguiente forma [33]:

- **Escenario No 1:** Proyectos que modelen el negocio con CUN solo pueden modelar el sistema con CUS.
- **Escenario No 2:** Proyectos que modelen el negocio con MC solo pueden modelar el sistema con CUS.
- **Escenario No 3:** Proyectos que modelen el negocio con DPN solo pueden modelar el sistema con DRP.
- **Escenario No 4:** Proyectos que no modelan negocio solo pueden modelar el sistema con HU.

En el área ingenieril el escenario No. 4 es el más acercado a las características del tipo de proyecto de videojuegos, dado que no se modela negocio, pero no cumple con el nivel de especificación requerida, por lo que se sustituyeron los siguientes artefactos:

- Historias de usuario.

- Especificación de requisitos de *software*.

Por:

- Diseño del videojuego.
- Especificación de mecanismos.

Las características del sistema (los mecanismos) y reglas del videojuego se describieron de la manera antes mencionada.

2.2.2 Lenguaje de modelado UML

Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo), está compuesto por diversos elementos gráficos que se combinan para conformar diagramas y tiene como principal objetivo especificar, construir, documentar y visualizar los artefactos que se crean durante el desarrollo de un sistema de software e [34].

2.2.3 Herramienta de modelado Visual Paradigm For UML 8.0

Como herramienta de modelado se utilizó **Visual Paradigm for UML 8.0**. Esta herramienta de *computer aided software engineering* (CASE) posee soporte multiplataforma así como licencia comercial y gratuita, puede ser utilizada durante todo el ciclo de vida del desarrollo de *software* para modelar los diferentes diagramas y artefactos que se generan durante el desarrollo de un producto [35].

2.2.4 Motor de juegos Unity 3D

Unity 3D es un potente motor *cross-platform*, para realizar proyectos tanto 2D como 3D con un entorno de desarrollo muy ameno. Fácil de manejar para los principiantes y suficientemente potente para los expertos, permite crear fácilmente juegos y aplicaciones para multitud de plataformas. Soporta oficialmente dos tipos de lenguaje: C#, *UnityScript* (básicamente *Javascript*). El más generalizado en la comunidad es C# [36].

2.2.5 Lenguaje de programación C#

C# es uno de los lenguajes de programación diseñados para la infraestructura de lenguaje común.
Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.
Entre sus principales características se encuentran [37]:

- Lenguaje de programación orientado a objetos simple, moderno y de propósito general.

- Inclusión de principios de ingeniería de software tales como revisión estricta de los tipos de datos, revisión de límites de vectores, detección de intentos de usar variables no inicializadas, y recolección de basura automática.
- Capacidad para desarrollar componentes de *software* que se puedan usar en ambientes distribuidos.
- Portabilidad del código fuente.
- Fácil migración del programador al nuevo lenguaje, especialmente para programadores familiarizados con C, C++ y Java.
- Soporte para internacionalización.
- Adecuación para escribir aplicaciones de cualquier tamaño: desde las más grandes y sofisticadas como sistemas operativos hasta las más pequeñas funciones.
- Aplicaciones económicas en cuanto a memoria y procesado.

2.3 Propuesta de solución

A partir del estudio realizado se propone, una arquitectura basada en la combinación de los estilos arquitectónicos: arquitectura basada en capas y arquitectura basada en componentes, para estructurar los diferentes elementos necesarios en un videojuego de plataformas. Los componentes de cada capa se comunican con otros componentes en otras capas a través de interfaces definidas o instancias de clases (en el caso de las clases se comunican con el *Game Manager* únicamente dado que es el que provee una fachada para las clases de la Capa Principal para interactuar con las demás). En la Figura 5 se observa la distribución de las capas presentes en la arquitectura propuesta.

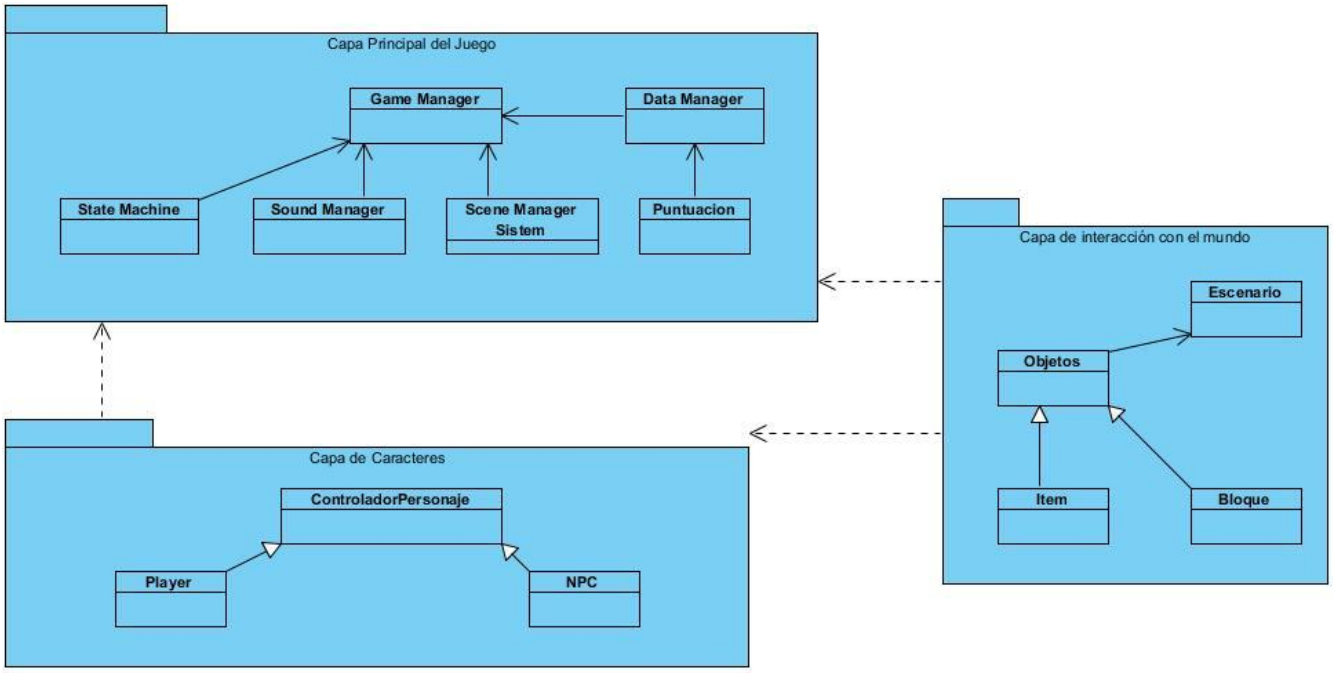


Figura 5: Propuesta de Solución.

Según Pressman, desde un punto de vista orientado a objetos, un componente es un conjunto de clases que colaboran [4], por lo que en cada *script* de la AS propuesta se definen las clases requeridas para definirlo como un componente. Para cada uno de los componentes de la AS propuesta se definen las clases (con los atributos y operaciones apropiadas). Cada clase dentro de un componente se elabora por completo para que incluya todos los métodos y acciones relevantes para su implementación.

Fueron elaborados los detalles de cada componente para que den información suficiente para poder ser implementado. Como estos componentes fueron construidos teniendo en mente lo reutilizable, se describen sus interfaces, las funciones que realizan y la comunicación y colaboración que requieren [4].

Como parte de la solución, también se definen todas las interfaces que permiten que las clases se comuniquen y colaboren con otras clases dependiendo de la capa en la que se encuentren.

Las capas se definen de la manera:

Capa Principal del Juego: En esta capa se encuentra el controlador principal del juego (*Game Manager*), la máquina de estados (*State Machine*), el sonido (*Sound Manager*), datos (*Data Manager*), escenas (*Scene Manager Sistem*) y Puntuación.

Game Manager: Es el controlador principal del sistema. Se implementa como un *gameObject* de Unity que se encuentra en todas las escenas del videojuego. Tiene asociados el resto de los *managers,* almacenando una instancia de cada uno de ellos (mediante el patrón fachada [ver acápite 2.3.1]). En caso de que se necesite usar puntuaciones para el videojuego, los datos se guardan en *hashtables* o tablas hash, que se encarga de asociar datos con valores y puede ser usada en la AS propuesta para almacenar grandes cantidades de información si es necesario. Si se desea acceder al controlador principal del juego se usa un patrón de diseño *singleton* que se implementó en el mismo (ver acápite 2.3.1).

State Machine: Se encarga de definir los estados del videojuego. Guarda los mismos en un *enum* que posee todos los diferentes estados que puede asumir el videojuego en un momento dado.

Sound Manager: Controla el sonido del juego y particularmente de cada elemento o acción que active un sonido. Guarda los tipos de sonidos (2 tipos de sonido, o sea, dos listas: FX para *Effects* y BGM para *Background*) en listas usando estructuras (*struct*), formadas por 3 elementos: un *AudioClip*, un *AudioSource* para poder reproducir el *AudioClip* y un *enum* que guarda los nombres de los sonidos para diferenciarlos a la hora de agregarlos al videojuego.

Data Manager: Maneja los datos del juego. Se encarga de cargar, guardar el estado del videojuego o la puntuación.

Scene Manager: Maneja las escenas del juego y los cambios entre ellas.

Puntuación*:* Controla la puntuación del juego. En caso de no existir se puede eliminar esta clase de la arquitectura.

Capa de Caracteres: En esta capa se encuentran los caracteres (ControladorPersonaje), diferenciándose entre ellos por ser jugables (*Player*) o no jugables (*NPC*). Si se desea agregar otro tipo de personaje que intervenga en el videojuego se incorpora en esta capa.

Capa de Interacción con el mundo: En esta capa se encuentran el escenario y los objetos que pertenecen al mismo. Si existen otros tipos de objetos que modifiquen el videojuego se agregan a esta capa.

2.3.1 Patrones de diseño usados

En el diseño e implementación de la propuesta de solución, se tuvo en cuenta el estudio realizado sobre los patrones de diseño. Basándose en las características y atributos de la arquitectura propuesta como:

- Se pretende usar en futuros desarrollos (reusabilidad).
- Se le piensa agregar nuevas funcionalidades y mejoras (extensibilidad).

Se utilizaron los patrones de diseño *singleton*, *fachade* y *observer*.

A continuación, se muestra cómo se evidencia el uso de estos patrones dentro de la solución:

Única Instancia (*Singleton*)

El patrón de diseño *singleton* (Figura 13) se asegura de que una clase solo tenga una instancia, y provee un punto de acceso global a ella. Para la propuesta de solución se usó este patrón en la clase *GameManage*r, dado que dicha clase es la controladora de la mayoría de los aspectos del juego, por lo que para optimizar código y evitar sobrecarga de recursos cargando demasiadas instancias de la misma clase se necesita un único acceso a la misma.

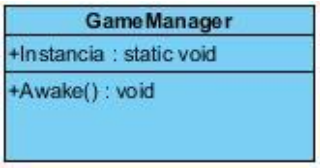


Figura 6: Patrón de diseño *singleton*

En el prototipo funcional se le implementó el patrón s*ingleton al GameManager*, donde se encuentran las tablas de puntos (*hashtable*) usadas para administrar la economía del videojuego, lo que, unido a su persistencia en las escenas (se necesita que esté presente administrando las escenas usando el

método *DontDestroyOnLoad*), hace necesario el uso de una instancia estática de la clase (a la que se pueda acceder desde cualquier otra clase).

El patrón *singleton* se implementó en un *script* que puede ser usado cada vez que se necesite implementar el singleton haciendo a la clase heredar de dicho *script* de la manera siguiente:

```
5  public class GameManager : Singleton<GameManager> {
6
7      public static GameManager Instancia
8      {
9          get
10         {
11             return ((GameManager)mInstance);
12         }
13     }
14
```

Figura 7: Implementación del patrón de diseño *singleton* en el *Game Manager*.

Observador (*Observer*)

El patrón de diseño *observer* (Figura 15) define una dependencia de uno a muchos para que cuando uno de los objetos cambie se notifique al observador. Dependiendo del tipo de notificación que se establezca, el observador también puede ser actualizado con nuevos valores.

En la propuesta de solución se le aplica este patrón para observar las diferentes acciones del videojuego, usando un centro de notificaciones (*Notification Center*), que es donde se implementa la lógica de gestionar los observadores (Adicionar, Eliminar y Notificar después de realizado el evento que se observaba).

Una vez ocurrido el evento que se estaba observando se procede a actualizar la puntuación con valores que se guardan en *hashtables* (administradas por el *GameManager*) para controlar las notificaciones y la puntuación (en el caso de ser necesario). Se usa para comenzar a generar los objetos del juego cuando el personaje empieza a correr (optimizando uso de memoria) o para activar el menú de *game over* en el momento que el personaje caiga de alguna plataforma hacia el vacío.

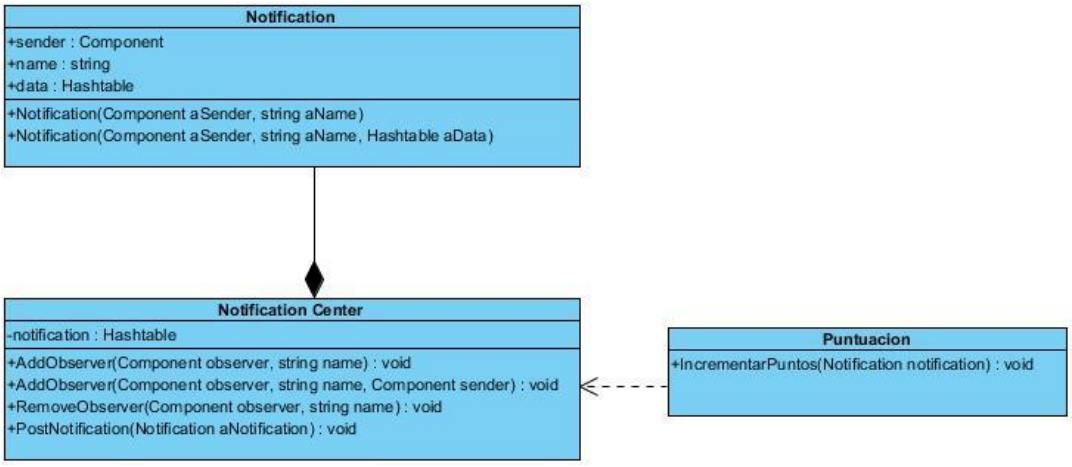


Figura 8: Patrón de diseño *Observer* implementado en el prototipo.

Fachada (*Facade*)

El patrón fachada (Figura 16) provee una interfaz a una serie de clases para evitar el exceso de instancias de las mismas, dejando a las subclases accesibles para ser usadas directamente. Se implementa en el *Game Manager*, que es la clase encargada de gestionar los aspectos principales del juego, junto al patrón *singleton,* permitiendo el acceso al resto de las clases asociadas a ella o convirtiendo al *Game Manager* en la fachada para todos los elementos de la Capa Principal del Juego (ver Figura 5). Dicho patrón de diseño hace que el sistema sea fácil de usar unificando diversas clases en una interfaz o fachada.

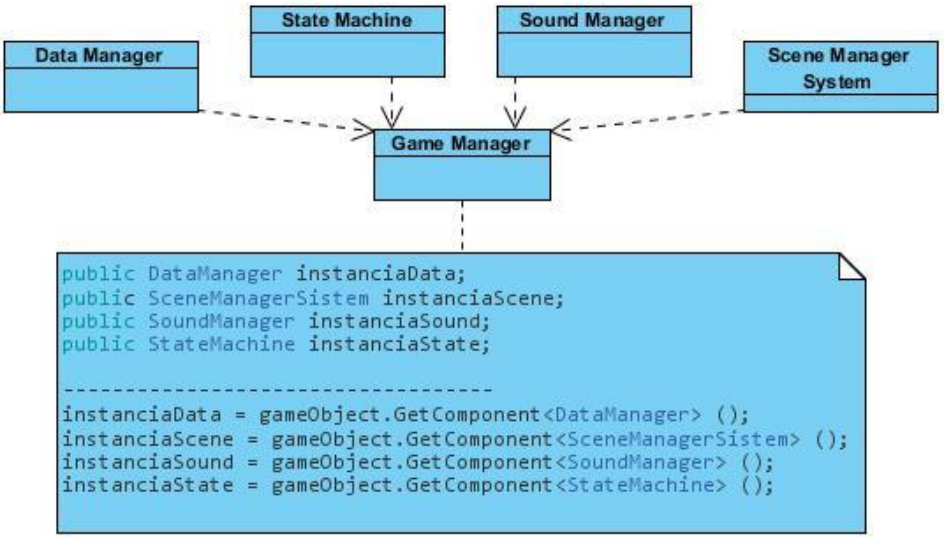


Figura 9. Patrón fachada implementado en el prototipo.

2.3.2 Restricciones arquitectónicas

- La arquitectura debe garantizar que los productos que se desarrollen sean multiplataforma (Android, Windows, Ubuntu).
- La arquitectura debe permitir la actualización, modificación o incorporación de compontes de forma natural.
- Las prestaciones de *hardware* dependerán de los requerimientos no funcionales de los productos que se desarrollen utilizando la arquitectura propuesta.
- Se debe usar como motor de videojuegos Unity 3D.

2.3.3 Prototipo funcional

Para validar la propuesta de solución se realizó un prototipo funcional de un videojuego de plataformas, que se describe usando los documentos de diseño de videojuegos y especificación de

mecanismos usados en el centro Vertex. El documento de diseño de videojuegos toma como base el libro "*Game Design Workshop-A playcentric approach to creating innovative games*" de la autora Tracy Fullerton [38] y el documento que describe la especificación de mecanismos de un videojuego toma como base "*Gameplay and Game Mechanics Desing: A Key to Quality in Video Games*" del autor Carlo Fabricatore [39].

2.4 Diseño del videojuego

El diseño del videojuego establece la visión y el enfoque del que guiará al proyecto hasta el final del proceso. En este acápite se diseña el juego teniendo en cuenta los elementos que conforman un juego:

- Elementos Formales: Definen la estructura del juego.
- Elementos Dramáticos: Definen el entretenimiento y el nivel de inmersión de los jugadores en el juego.

2.4.1 Resumen del juego

En un mundo medieval un caballero fantasma debe recorrer el mundo llevando un mensaje y coleccionando espadas, que aumenta su poder, evitando caer de las plataformas en las que se sitúa. Cada vez que lo haga empezará desde cero. Mientras más lejos llegue, mayor cantidad de personas conocerán el mensaje que trae.

El usuario debe tocar la pantalla (plataformas *Android*) o presionar clic o espacio (plataformas PC) para darle un pequeño impulso al personaje y saltar de plataforma en plataforma.

2.4.2 Metas para la experiencia del jugador

- El jugador tendrá la libertad de saltar en el escenario siempre tratando de mantenerse sobre una plataforma (puede realizar doble salto).
- Cada vez que salte a una nueva plataforma obtendrá un punto.
- Para incrementar aún más la puntuación el jugador puede intentar obtener espadas, cada una de ellas sumaría 5 puntos al acumulado total de la puntuación.

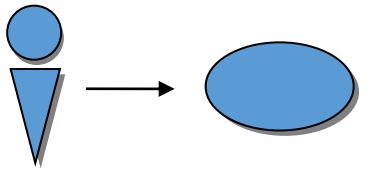
2.4.3 Elementos Formales

Los elementos formales describen el videojuego, definiendo su estructura.

Jugadores:

Tabla 1: Elementos formales.

Aspecto	Descripción

Invitación a Jugar	Botón de comienzo: “Jugar”.
Cantidad	Sólo un jugador.
Roles	“Caballero fantasma”
Patrón de Interacción	vs Juego 

Objetivo:

Tabla 2: Objetivo.

Objetivo	**Descripción**
No caer de la plataforma	El jugador debe saltar entre plataformas manteniéndose siempre sobre una de ellas.
Tipo	Explícito.
Categorías	• Plataformas. • Aventuras.

Procedimientos:

La idea base consiste en avanzar por el escenario recolectando objetos para aumentar la puntuación evitando caer de las plataformas en las que se sitúa el personaje. No se tiene un control directo sobre la velocidad con la que se mueve el personaje, o con la cantidad de objetos o plataformas que se generan.

Reglas:

1. Siempre se empieza con la puntuación en 0.
2. Cada vez que se empiece el juego se reiniciará la puntuación.
3. Cada vez que el personaje se caiga de una plataforma se perderá la partida.

Recursos:

1. Bonificaciones (espadas).

Conflictos:

1. Las plataformas se generan automáticamente.

2. Las espadas se generan automáticamente, más lento que las plataformas.

Frontera o Límite:

La historia del juego es de una época que no tiene nada que ver con la realidad actual, en un juego fantástico que le muestra una historia en la época de como la edad media combinada con algo futurista en el escenario donde se utilizaban armas como espadas.

Resultado:

Básicamente el objetivo principal del juego es evitar caer de la plataforma y recolectar la mayor cantidad de espadas posibles. Pero además de esto cada jugador busca obtener una puntuación superior a la alcanzada anteriormente.

2.4.4 Elementos Dramáticos

Los elementos dramáticos se encargan de definir el nivel de inmersión de los jugadores en el juego. Debido a que el videojuego realizado es un prototipo funcional que tiene como objetivo validar la propuesta de solución no se tomarán en cuenta todos los elementos dramáticos para describir dicho videojuego. Para la descripción del prototipo funcional se plantean los elementos dramáticos: premisa y retos.

Premisa:

“A Knight´s Tale” surge de la necesidad de validar la propuesta de solución, empleando la AS en un prototipo funcional del género plataformas.

Retos:

1. El principal reto que posee este juego consiste en mantenerse con vida sin importar la dificultad que supone no caerse de la plataforma.
2. El reto secundario sería aumentar el récord de puntuación obtenido, que se guarda una vez se pierde el juego.

2.5 Especificación de mecanismos

La especificación de mecanismos tiene como objetivo identificar los mecanismos que forman al videojuego, así como sus propiedades y organización arquitectónica. Dichos mecanismos se encargan de agrupar por paquetes las funcionalidades del videojuego, permitiendo la interacción del usuario con el sistema e incrementando la jugabilidad. Esta sección esencialmente describe lo que el jugador puede hacer y cómo puede hacerlo.

2.5.1 Catálogo de mecanismos

Tabla 3: Catálogo de Mecanismos.

No.	Nombre	Descripción	Organización arquitectónica perteneciente
M1	**Mecanismo de Física (Physics)**	**Objetos:** • Personaje. **Propiedades:**	Mecanismos núcleos.

		• Personaje*:* Salto y doble salto. **Comportamientos:** • Personaje. Salta entre plataformas. El videojuego presenta un movimiento horizontal, en el que se generan automáticamente bloques (plataformas) y el personaje se desplaza siempre hacia la derecha teniendo una velocidad constante. Con dar un clic (PC) o tocar el táctil (dispositivos Android) el personaje salta, teniendo la capacidad de realizar un doble salto, descendiendo luego producto de la fuerza de gravedad. **Relaciones:** • M2.	
M2	**Mecanismo de Recolección.**	**Objetos:** • Personaje. • Espada. **Propiedades:** • Personaje: Choca con los objetos (Espada). • Espada: Incrementan los puntos de bonificaciones para el personaje. **Comportamientos:** • Personaje: Se mueve entre plataformas. Aumenta la cantidad de puntos según vaya colisionando con las espadas que encuentra. • Espadas: Estáticas para ser recolectados por el personaje. **Relaciones:** • M1.	Mecanismos núcleos.

2.6 Vistas arquitectónicas

Siguiendo los pasos propuestos para describir el prototipo funcional, en conjunto la metodología AUP –vUCI y utilizando el modelo de Robert Nord mencionado en el Capítulo 1, se proponen las siguientes vistas:

- Vista conceptual.

- Vista de módulos.
- Vista de código.
- Vista de ejecución.

El objetivo de las vistas de la arquitectura es la simplificación o abstracción de los modelos, de los cuales se destacan los detalles más significativos y se obvian los que no representan un aporte significativo a la visión de la solución del problema [17].

2.7.1 Vista conceptual

En esta vista se describe el sistema en términos de sus elementos principales de diseño y las relaciones entre éstos, dentro de un dominio determinado. Esta vista es independiente de las decisiones de implementación y enfatiza en los protocolos de interacción entre los elementos de diseño [18].

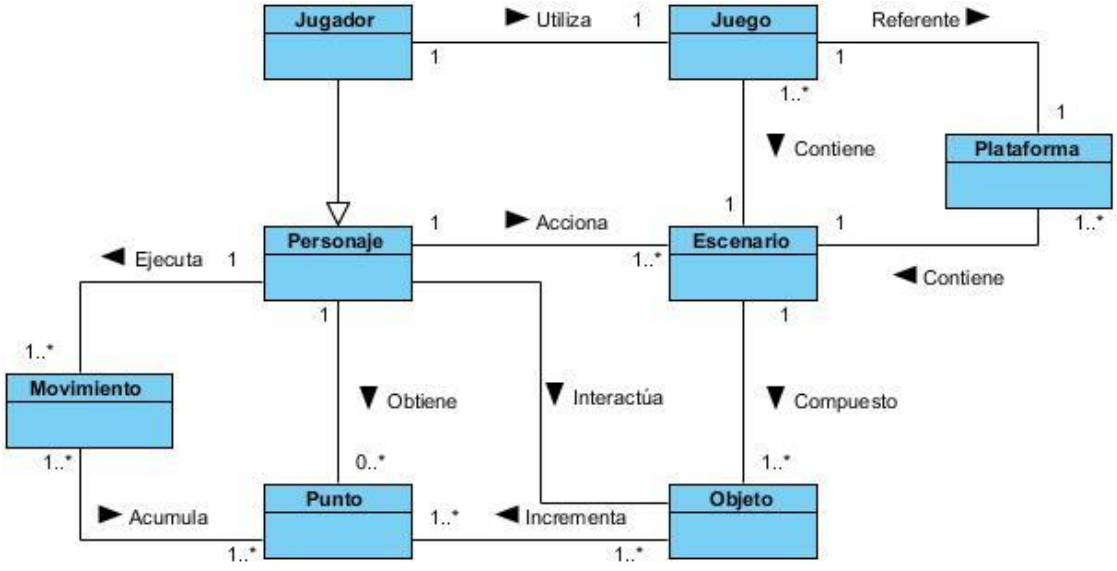


Figura 10: Diagrama Conceptual.

Jugador: Representa al usuario que interactúa con el sistema.

Personaje: Representa al modelo mediante el cual se va a interactuar con el sistema.

Videojuego: Contiene la lógica del sistema, así como los escenarios y perfiles.

Escenario: Se refiere al entorno de movimiento del modelo con el que se vaya a jugar.

Objeto: Representa los objetos con los que se puede interactuar.

Punto: Son los puntos que obtiene el personaje al avanzar en el juego.

2.7.2 Vista de módulos

En esta vista se captura la descomposición funcional y las capas del sistema. El sistema es descompuesto lógicamente en subsistemas, módulos, y unidades abstractas. Cada capa representa las distintas interfaces de comunicación permitidas entre los módulos [18].

2.7.2.1 Diagrama de paquetes

Un paquete es un mecanismo utilizado para agrupar elementos de UML, es una parte de un modelo. Contiene elementos del modelo al más alto nivel, tales como clases y sus relaciones, máquinas de estado, diagramas de casos de uso, interacciones y colaboraciones: cualquier elemento que no esté contenido en otro. Los paquetes pueden contener otros paquetes. Las dependencias entre paquetes resumen dependencias entre los elementos internos a ellos, es decir, las dependencias del paquete se derivan a partir de las dependencias entre los elementos individuales [31].

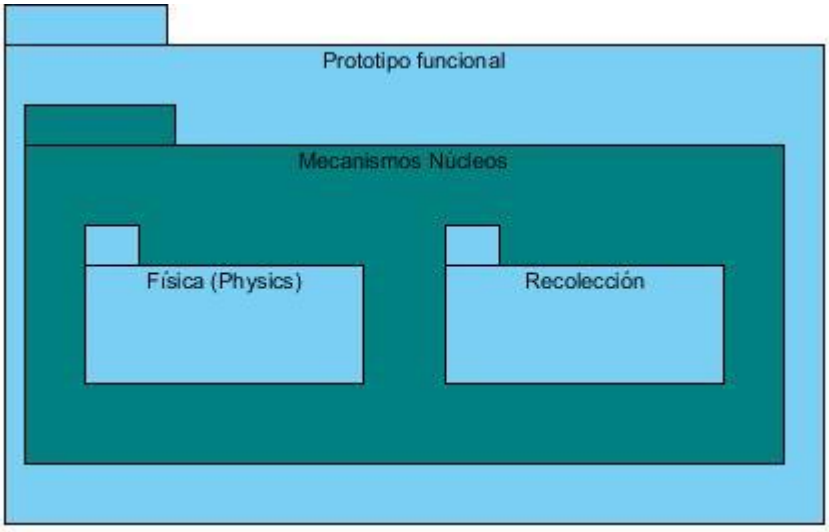


Figura 11: Diagrama de paquetes.

Mecanismos Núcleos: Incluye a los mecanismos sin los cuales el juego no puede existir. Representan a los mecanismos centrales del videojuego que constituyen el sentido de enganche del jugador y el propósito esencial del juego. Permiten la navegación por el entorno y la recolección de diferentes objetos que aparecen durante la navegación.

2.7.2.2 Diagramas de transición de estados

El diagrama de transición de estados es una herramienta de modelado que sirve para describir el comportamiento requerido de los sistemas de tiempo real, al igual que la porción de la interfaz humana que la mayoría de los sistemas en línea tiene.

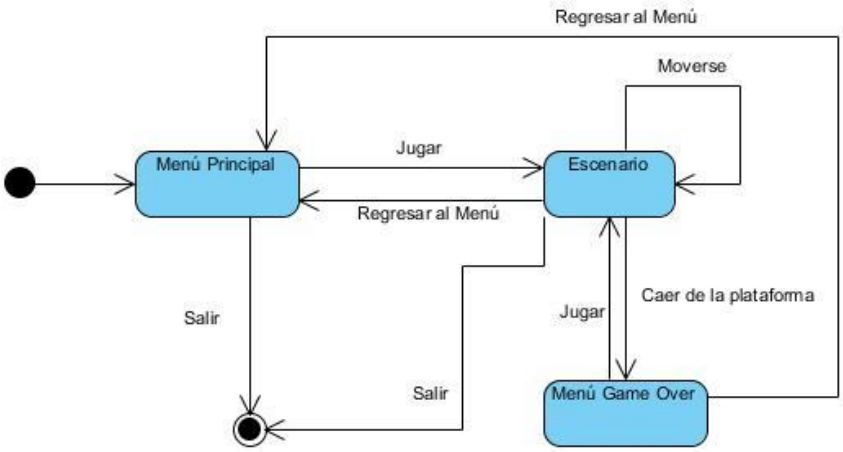


Figura 12: Diagrama de transición de estados: Mecanismo de Física.

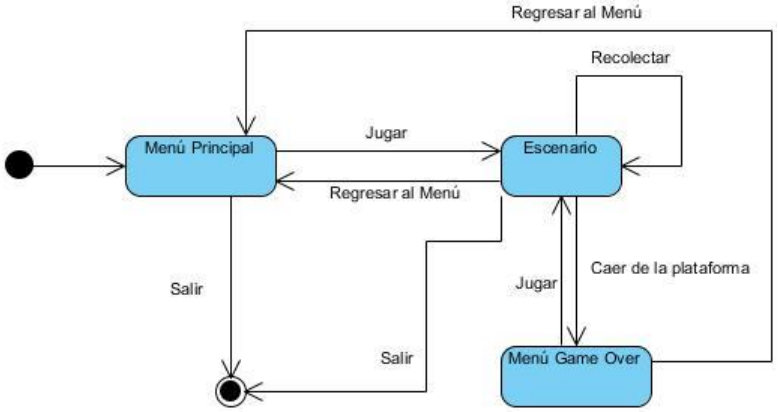


Figura 13: Diagrama de transición de estados: Mecanismo de Recolección.

2.7.3 Vista de código

En esta vista se organiza el código fuente representando la estructura y organización del prototipo funcional. Se puede apreciar la distribución de los componentes en cada una de las capas, y algunos patrones de diseño que se especificarán en acápites posteriores.

Definición de clases

En la figura 11 se puede apreciar un ejemplo de la definición de clases del prototipo funcional (*Game Manager*), usando las clases y métodos de Unity 3D, como la clase *MonoBehaviour* y el método *DontDestroyOnLoad*.

```
public class GameManager : MonoBehaviour {

    public static GameManager Instancia;
    public Hashtable puntosAIncrementar = new Hashtable();
    public DataManager instanciaData;
    public SceneManagerSistem instanciaScene;
    public SoundManager instanciaSound;
    public StateMachine instanciaState;

    public void Awake(){
        if (Instancia == null) {
            Instancia = this;
            DontDestroyOnLoad (gameObject);
        }
        else if(Instancia != this){
            Destroy (gameObject);
        }
    }
}
```

Figura 14: Ejemplo de definición de clases.

Definición de métodos

Un ejemplo de definición de métodos se puede apreciar en la figura 12 con el método Cargar, perteneciente al *Data Manager*.

```
void Cargar(){
    if (File.Exists (rutaArchivo)) {
        BinaryFormatter bf = new BinaryFormatter ();
        FileStream file = File.Open (rutaArchivo, FileMode.Open);
        DatosAGuardar datos = (DatosAGuardar)bf.Deserialize (file);
        puntuacionMaxima = datos.puntuacionMaxima;
        file.Close ();
    }
    else {
        puntuacionMaxima = 0;
    }
}
```

Figura 15: Ejemplo de definición de métodos.

Asignación de variables

Como muestra de cómo fueron asignadas las variables se puede apreciar en la figura 13, en el *Game Manager*, los valores de las instancias especificadas.

```
instanciaData = gameObject.GetComponent<DataManager> ();
instanciaScene = gameObject.GetComponent<SceneManagerSistem> ();
instanciaSound = gameObject.GetComponent<SoundManager> ();
instanciaState = gameObject.GetComponent<StateMachine> ();
```

Figura 16: Ejemplo de asignación de variables.

Llamadas a funciones

En la figura 14 se aprecia la llamada al método ActualizarMarcador desde IncrementarPuntosCinco, contenido en el *script* Puntuación.

```
void IncrementarPuntosCinco (NotificationCenter.Notification notification){
    puntuacion += (int)GameManager.Instancia.puntosAIncrementar ["puntos de cinco"];
    ActualizarMarcador ();
}

void ActualizarMarcador(){
    marcador.text = puntuacion.ToString();
}
```

Figura 17: Ejemplo de llamadas a funciones.

Estructuras de control

Como ejemplo de estructura de control usada se aprecia el *if-else* presente en la clase Destructor.

```
if (other.tag == "Player") {
    NotificationCenter.DefaultCenter.PostNotification (this, "PersonajeHaMuerto");
    GameObject personaje = GameObject.Find ("Personaje");
    personaje.SetActive (false);
}
else if (other.tag == "bloque"){
    Destroy (other.gameObject);
}
```

Figura 18: Ejemplo de estructuras de control.

Diagrama de componentes

En la figura 16 se puede apreciar desde una vista más cercana al código como el prototipo funcional es dividido en componentes y muestra las dependencias entre estos componentes.

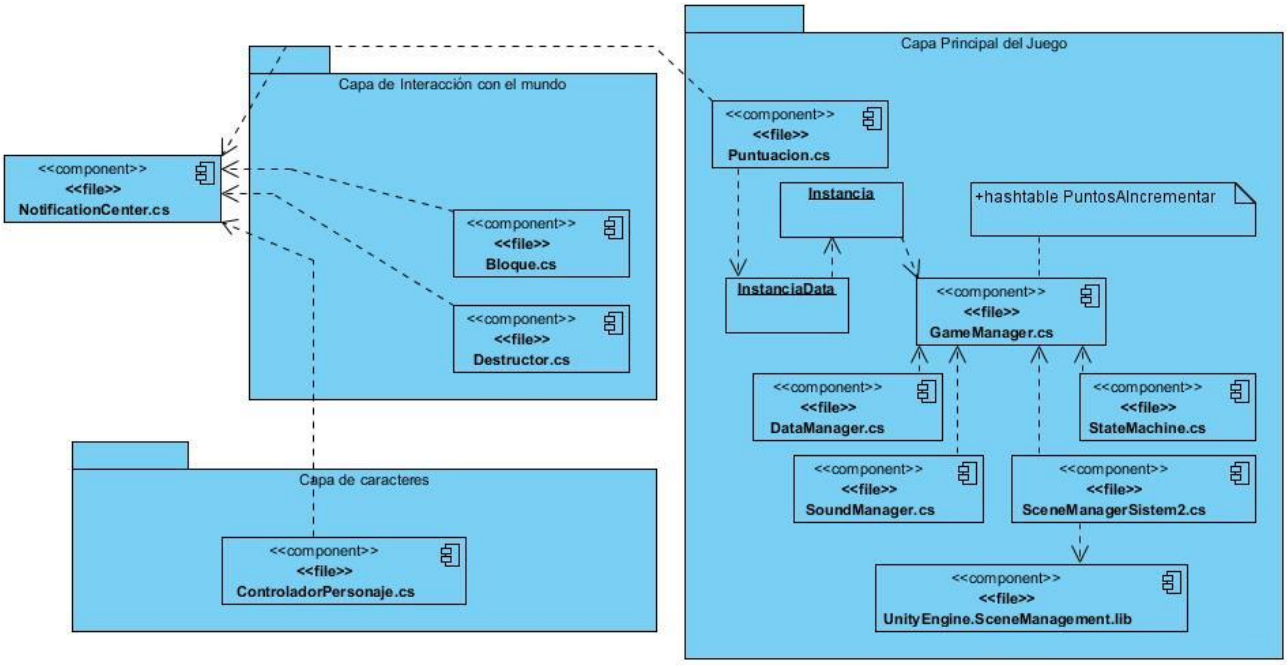


Figura 19: Diagrama de componentes: Vista de código.

2.7.4 Vista de ejecución

En la vista de ejecución se describe la estructura dinámica del sistema en términos de sus elementos en tiempo de ejecución. Por ejemplo, se modela las tareas operativas del sistema, procesos, mecanismos de comunicación y asignación de recursos. Algunos de los aspectos que se consideran en esta vista son, el desempeño y el entorno de ejecución [18].

El diagrama de despliegue muestra la configuración de los nodos (procesadores y dispositivos) que participan en la ejecución y de los componentes que residen en los mismos, mostrando la colaboración entre nodos mediante protocolos de comunicación. De acuerdo a la planificación del proyecto, las aplicaciones a desarrollar próximamente, sobre la base de la arquitectura propuesta, serán desplegadas en estaciones de trabajo independientes que cumplan con los requerimientos de *hardware* y *software* especificados en la sección 2.5 (restricciones arquitectónicas).

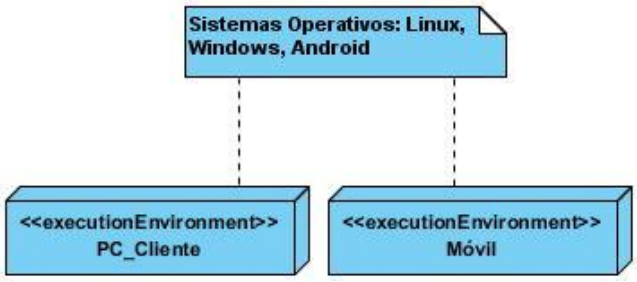


Figura 20: Diagrama de despliegue.

2.9 Conclusiones parciales

Con la realización de este capítulo se logró desarrollar como propuesta de solución una arquitectura de software para videojuegos, implementada sobre el prototipo funcional de un videojuego de plataformas, para ello se usó el tipo de arquitectura en capas basado en componentes. Se realizó una caracterización de las herramientas y metodologías usadas para desarrollar el prototipo funcional, para tener un mayor entendimiento de cómo se desarrollará el videojuego. Se usaron las vistas propuestas por Robert Nord con el objetivo de conceptualizar el videojuego y representar la AS del sistema. Finalmente, fueron descritos los patrones de diseño que ayudaron a conformar la AS propuesta y que fomentan la reutilización de la solución propuesta.

Capítulo #3. Evaluación de la propuesta

3.1 Introducción

En este capítulo se exponen algunos conceptos asociados a la evaluación de las arquitecturas de software, fundamentalmente las técnicas y métodos que se emplean, así como los atributos de calidad que se miden. Se evalúa la arquitectura propuesta empleando las técnicas basadas en escenarios y prototipo, usando también el método de evaluación ATAM, tomando como base los atributos de calidad de la norma ISO/IEC 27010.

3.2 Evaluación de la arquitectura de software

La arquitectura de software posee gran impacto sobre la calidad de un sistema, por lo que se hace necesario evaluarla para determinar su potencial para alcanzar los atributos de calidad requeridos. Es importante destacar que la evaluación no define si una arquitectura es buena o no, simplemente expresa donde se encuentran los riesgos y fortalezas de la misma.

El primer paso para evaluar una arquitectura es conocer qué es lo que se quiere evaluar, de esta forma es posible establecer la base para la evaluación, otra decisión importante es determinar cuándo se realizará la evaluación. Para esto, aunque es posible evaluar la arquitectura en cualquier fase del desarrollo, existen dos variantes definidas de cuando realizar la evaluación: evaluación temprana y evaluación tardía [40].

Para realizar la evaluación temprana no es necesario esperar que la arquitectura esté totalmente especificada, esta puede realizarse desde las fases tempranas del diseño y a lo largo del proceso de desarrollo, lo que permite tomar decisiones arquitectónicas producto a una evaluación en función de los atributos de calidad esperados. La evaluación tardía se realiza cuando la arquitectura ya está establecida y la implementación se ha culminado, realizar la evaluación en este momento se considera muy útil, pues se puede observar el cumplimiento de los atributos de calidad asociados al sistema y su comportamiento de forma general [40].

3.2.1 Atributos de calidad.

El modelo de calidad representa la piedra angular en torno a la cual se establece el sistema para la evaluación de la calidad del producto. En este modelo se determinan las características de calidad que se van a tener en cuenta a la hora de evaluar las propiedades de un producto *software* determinado.

La calidad del producto *software* se puede interpretar como el grado en que dicho producto satisface los requisitos de sus usuarios aportando de esta manera un valor. Son precisamente estos requisitos (funcionalidad, rendimiento**,** seguridad, mantenibilidad) los que se encuentran representados en el modelo de calidad, el cual categoriza la calidad del producto en características y subcaracterísticas.

El modelo de calidad del producto definido por la ISO/IEC 25010 se encuentra compuesto por las ocho características de calidad que se muestran en la Figura 17 [41]:

Figura 21: Características del producto *software*.

3.2.2 Modelos de calidad de software

Los atributos de calidad se pueden organizar y descomponer de maneras diferentes, en lo que se conoce como modelos de calidad de *software* (MCS), estos modelos facilitan el entendimiento del proceso de ingeniería de software [42].

Se usará como modelo de calidad ISO/IEC 25010, que define [43]:

- Un modelo de calidad en uso que se compone de cinco características (algunas de las cuales se subdividen en subcaracterísticas). Se relacionan con el resultado de la interacción cuando un producto se utiliza en un contexto particular de uso.
- Un modelo de calidad del producto que se compone de ocho características (que se subdividen en subcaracterísticas). Se refieren a propiedades estáticas de *software* y las propiedades dinámicas del sistema informático. El modelo es aplicable a los productos de *software* y sistemas informáticos.

Las características definidas por ambos modelos son relevantes para todos los productos de *software* y sistemas informáticos. Las características y subcaracterísticas proporcionan coherencia terminológica para especificar, medir y evaluar la calidad del producto *software* y sistemas informáticos.

3.3 Técnicas de evaluación de arquitectura

Las técnicas utilizadas para la evaluación de atributos de calidad requieren grandes esfuerzos por parte del ingeniero de software para crear especificaciones y predicciones. Estas técnicas requieren información del sistema a desarrollar que no está disponible durante el diseño arquitectónico, sino al principio del diseño detallado del sistema [44].

En vista de que el interés es tomar decisiones de tipo arquitectónico en las fases tempranas del desarrollo, son necesarias técnicas que requieran poca información detallada y puedan conducir a resultados relativamente precisos. Las técnicas existentes en la actualidad para evaluar arquitecturas

permiten hacer una evaluación cuantitativa sobre los atributos de calidad a nivel arquitectónico, pero se tienen pocos medios para predecir el máximo (o mínimo) teórico para las arquitecturas de software. Sin embargo, debido al costo de realizar este tipo de evaluación, en muchos casos los arquitectos de software evalúan cualitativamente, para decidir entre las alternativas de diseño [45].

3.3.1 Evaluación basada en escenario

Un escenario es una breve descripción de la interacción de alguno de los involucrados en el desarrollo del sistema con este. Consta de tres partes: el estímulo, el contexto y la respuesta. El estímulo es la parte del escenario que explica o describe lo que el involucrado en el desarrollo hace para iniciar la interacción con el sistema. Puede incluir la ejecución de tareas, cambios en el sistema, ejecución de pruebas, entre otros. El contexto describe qué sucede en el sistema al momento del estímulo. La respuesta describe, a través de la arquitectura, cómo debería responder el sistema ante el estímulo. Este último elemento es el que permite establecer cuál es el atributo de calidad asociado [46].

3.3.2 Evaluación basada en la experiencia

En muchas ocasiones los arquitectos e ingenieros de software otorgan valiosas ideas que resultan de utilidad para la evasión de decisiones erradas de diseño, estas ideas se basan en factores subjetivos (como la experiencia) [43] y están respaldadas por una línea lógica de razonamiento, que se puede adquirir por el trabajo realizado en proyectos similares. Por tanto, el principal instrumento de evaluación con que cuenta esta técnica es precisamente la intuición y experiencia con que cuentan los arquitectos y demás miembros del equipo de desarrollo. Existen dos tipos de evaluación basada en experiencia: la evaluación informal, que es realizada por los arquitectos de software durante el proceso de diseño, y la realizada por equipos externos de evaluación de arquitecturas.

3.3.3 Evaluación basada en prototipo

Esta técnica consiste en implementar una parte de la arquitectura de software y ejecutarla en el contexto del sistema. Para su uso se necesita mayor información sobre el desarrollo y disponibilidad de *hardware*, y los elementos que constituyen el contexto del sistema de *software*. Mediante esta técnica se obtiene un resultado de evaluación con mayor exactitud [43].

3.4 Métodos de evaluación de arquitectura

Los métodos de evaluación arquitectónica, evalúan el potencial del diseño arquitectónico para alcanzar los niveles deseados en cuanto a los requisitos de calidad [44], estos métodos se ven asistidos de las técnicas de evaluación arquitectónica analizadas anteriormente. Actualmente existen diversos métodos para realizar pruebas a la arquitectura de software, cada uno con características específicas, escoger un

método de evaluación requiere tener bien definidos los atributos que se desean evaluar. Algunos de los métodos de evaluación son los siguientes:

- Método de Análisis de Arquitectura de Software (*Software Architecture Analysis Method*, SAAM).
- Método de Revisión Intermedio de Diseño (*Active Reviews for Intermediate Designs*, ARID).
- Método de Análisis de Acuerdos de Arquitectura de Software (*Architecture Trade-off Analysis Method*, ATAM).

3.4.1 Métodos de evaluación

- SAAM

El SAAM fue el primer método en ser ampliamente difundido y documentado. Se creó originalmente para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida atributos de calidad, tales como la portabilidad, escalabilidad e integrabilidad [40].

- ARID

El ARID es conveniente para realizar la evaluación de diseños parciales en las etapas tempranas del desarrollo. Es un híbrido entre Active Design Review (ADR) y ATAM. Las preguntas giran en torno a la calidad y completitud de la documentación y la suficiencia, ajuste y conveniencia de los servicios que provee el diseño propuesto [40].

- ATAM

El ATAM está inspirado en tres áreas distintas: los estilos arquitectónicos, el análisis de atributos de calidad y el método de evaluación SAAM [40]. El método se concentra en la identificación de los estilos arquitectónicos o enfoques arquitectónicos utilizados y ayuda a los involucrados en el proyecto a entender las consecuencias de las decisiones arquitectónicas tomadas con respecto a los atributos de calidad. Su desarrollo se basa nueve pasos agrupados en cuatro fases.

3.5 Evaluación de la arquitectura propuesta

Después del análisis realizado sobre las diferentes técnicas y métodos de evaluación de arquitecturas, se seleccionan, para evaluar la arquitectura propuesta, la técnica de evaluación basada en prototipo y en escenarios en conjunto con el método de evaluación ATAM.

3.5.1 Evaluación mediante ATAM

El método ATAM está compuesto por nueve pasos divididos en cuatro fases, los cuales se adaptarán de acuerdo a los requerimientos del proyecto. Después de la descripción de la solución propuesta y el análisis realizado en los capítulos 1 y 2, se procede a evaluar la arquitectura propuesta empleando este método.

3.5.1.1 Árbol de utilidad

Los atributos presentes en el árbol de utilidad, usados para evaluar la arquitectura mediante el método ATAM, en conjunto con técnicas basadas en escenarios y prototipo, son los más relevantes a comprobar sobre el dominio de aplicación desarrollado (videojuegos), dado que aportan interpretaciones consistentes, pertinentes, facilitando el análisis de su comportamiento desde el núcleo arquitectónico hasta los horizontes de la arquitectura relacionados con las funcionalidades más básicas del videojuego (prototipo funcional).

Tabla 4: Árbol de utilidad.

Atributo	Subatributos	Escenario	Prioridad
Adecuación Funcional	Corrección Funcional.	El videojuego debe proveer resultados correctos con el nivel de precisión requerido.	Alta
Eficiencia	Comportamiento Temporal.	Los tiempos de respuesta y procesamiento del videojuego con respecto a cada una de las acciones del usuario deben ser cortos.	Alta
	Utilización de recursos.	Las cantidades y tipos de recursos utilizados cuando el videojuego lleva a cabo su función deben ser bajos.	Alta
Compatibilidad	Coexistencia.	El videojuego debe poder coexistir con otro *software* independiente, en un entorno común, compartiendo recursos comunes sin detrimento.	Media
	Interoperabilidad.	El videojuego debe poder intercambiar información y utilizar la información intercambiada con otro sistema.	Media
Usabilidad	Estética.	La interfaz de usuario debe agradar y satisfacer la interacción con el usuario.	Media
	Aprendizaje.	El usuario debe comprender el sistema con facilidad.	Media
	Operabilidad (Jugabilidad).	El sistema debe satisfacer la experiencia del jugador.	Media
Fiabilidad	Tolerancia.	El sistema opera según lo previsto en presencia de fallos *hardware* o *software*.	Alta
	Disponibilidad.	El sistema está disponible cuando se requiere su uso.	Alta

Seguridad	Confidencialidad.	El sistema protege contra el acceso a datos e información de personas no autorizadas, ya sea accidental o deliberadamente.	Media
	Integridad.	El sistema previene accesos o modificaciones no autorizados a sus datos.	Alta
Mantenibilidad	Modularidad.	Al realizar un cambio en un componente del sistema debe tener un impacto mínimo en los demás.	Alta
	Reusabilidad.	Los elementos del videojuego deben ser reusables en otros desarrollos.	Alta
	Capacidad de ser modificado.	El sistema puede ser modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.	Alta
Portabilidad	Adaptabilidad	El producto puede ser adaptado de forma efectiva y eficiente a diferentes entornos determinados de *hardware*, *software*, operacionales o de uso.	Media
	Facilidad de instalación.	El producto se puede instalar y/o desinstalar de forma exitosa en un determinado entorno.	Media

3.5.1.2 Especificación de los escenarios

1. El videojuego debe proveer resultados correctos con el nivel de precisión requerido.

Tabla 5: Escenario 1.

Atributo de Calidad	**Adecuación Funcional.**
Sub-atributos/Sub-característica	Corrección Funcional.
Objetivo	El videojuego debe proveer resultados correctos con el nivel de precisión requerido.
Origen	Usuario del sistema.
Artefacto	Script Puntuación
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Crear nueva puntuación	
Se alcanza una puntuación máxima.	El sistema guarda la información, mostrando la nueva

	puntuación alcanzada correctamente.
Medida de respuesta	
Navegar en el sistema.	

2. Los tiempos de respuesta y procesamiento del videojuego con respecto a cada una de las acciones del usuario deben ser cortos.

Tabla 6: Escenario 2.

Atributo de Calidad	**Eficiencia.**
Sub-atributos/Sub-característica	Comportamiento Temporal.
Objetivo	Los tiempos de respuesta y procesamiento del videojuego con respecto a cada una de las acciones del usuario deben ser cortos.
Origen	Usuario del sistema.
Artefacto	Pantallas del videojuego.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Se interactúa con el sistema (usando la herramienta *profiler* de Unity)	
Se usa la herramienta *profiler* de Unity con el sistema desplegado para comprobar tiempos de respuesta de los elementos. Los mismos oscilan entre: • *Scripts:* 0.1 ms (4000 fps) y 0.25 ms (10000 fps).	El sistema responde eficientemente.
Medida de respuesta	
Navegar en el sistema.	

3. Las cantidades y tipos de recursos utilizados cuando el videojuego lleva a cabo su función deben ser bajos.

Tabla 7: Escenario 3.

Atributo de Calidad	**Eficiencia.**
Sub-atributos/Sub-característica	Utilización de recursos.
Objetivo	Las cantidades y tipos de recursos utilizados cuando el videojuego lleva a cabo su función deben ser bajos.

Origen	Administrador del sistema.
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo **1. Se despliega el sistema**	**Respuesta: Flujo de eventos (Escenarios)**
Se usa la herramienta *profiler* de Unity con el sistema desplegado para comprobar uso de memoria del sistema.	El sistema responde eficientemente usando, en conjunto con los procesos del sistema únicamente en la: • Unity: 58.1 mb. • Mono: 7.1 mb. • GfX Driver: 23.6 mb. • FMOD: 23.6 mb. • Profiler: 8.0 mb. Se puede determinar que el sistema es eficiente y funciona correctamente.
Medida de respuesta	
Desplegar el sistema	

4. El videojuego debe poder coexistir con otro *software* independiente, en un entorno común, compartiendo recursos comunes sin detrimento.

Tabla 8: Escenario 4.

Atributo de Calidad	**Compatibilidad.**
Sub-atributos/Sub-característica	Coexistencia.
Objetivo	El videojuego debe poder coexistir con otro *software* independiente, en un entorno común, compartiendo recursos comunes sin detrimento.
Origen	Administrador del sistema.
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo **1. Se despliega el videojuego conjuntamente con otro *software* que requiera recursos del mismo entorno.**	**Respuesta: Flujo de eventos (Escenarios)**

Se despliegan los siguientes *software*: • Prototipo funcional. • Photoshop. • AIMP player.	El sistema cuenta con un alto grado de co-existencia pues es capaz de convivir con otras aplicaciones que comparten los recursos de la estación de trabajo donde se encuentra sin impactar perjudicialmente a otras aplicaciones.
Medida de respuesta	
Desplegar ambos sistemas.	

5. El videojuego debe poder intercambiar información y utilizar la información intercambiada con otro sistema.

Tabla 9: Escenario 5.

Atributo de Calidad	**Compatibilidad.**
Sub-atributos/Sub-característica	Interoperabilidad.
Objetivo	El videojuego debe poder intercambiar información y utilizar la información intercambiada con otro sistema.
Origen	Administrador del sistema.
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Se inserta un *script* (Aspect Utility) al prototipo funcional proveniente de otro sistema.	
Se inserta el *script* en la cámara.	El sistema responde eficientemente usando un *script* que proviene de otro videojuego (tutorial). Nota: Aspect Utility se usa para mantener las dimensiones de la cámara constantes a medida que se navega por el videojuego.
Medida de respuesta	
Modificar el prototipo usando la información de otro videojuego.	

6. La interfaz de usuario debe agradar y satisfacer la interacción con el usuario.

Tabla 10: Escenario 6.

Atributo de Calidad	**Usabilidad.**
Sub-atributos/Sub-característica	Estética.
Objetivo	La interfaz de usuario debe agradar y satisfacer la

	interacción con el usuario.
Origen	Usuario del sistema
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. El usuario interactúa con el sistema	
El usuario interactúa con el sistema.	El usuario encuentra las interfaces del sistema agradables estéticamente. La combinación de colores entre escenario, personaje y elementos del escenario se corresponde y los botones son intuitivos, a la vez que no entorpecen la jugabilidad.
Medida de respuesta	
Navegar en el sistema.	

7. El usuario debe comprender el sistema con facilidad.

Tabla 11: Escenario 7.

Atributo de Calidad	**Usabilidad.**
Sub-atributos/Sub-característica	Aprendizaje.
Objetivo	El usuario debe comprender el sistema con facilidad.
Origen	Usuario del sistema
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. El usuario interactúa con el sistema	
El usuario interactúa con el sistema.	El usuario comprende fácilmente el sistema.
Medida de respuesta	
Navegar en el sistema.	

8. El sistema debe satisfacer la experiencia del jugador.

Tabla 12: Escenario 8.

Atributo de Calidad	**Usabilidad.**
Sub-atributos/Sub-característica	Operabilidad (Jugabilidad)
Objetivo	El sistema debe satisfacer la experiencia del jugador.
Origen	Usuario del sistema

Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo **1. Número de pasos a ejecutar para llegar a jugar.**	**Respuesta: Flujo de eventos (Escenarios)**
El usuario interactúa con el sistema.	1. El sistema muestra la pantalla de inicio. 2. El usuario presiona el botón jugar. 3. Se produce el cambio de escena y se procede a jugar. 4. El usuario procede a jugar.
Medida de respuesta	
Navegar en el sistema.	

9. El sistema opera según lo previsto en presencia de fallos *hardware* o *software*.

Tabla 13: Escenario 9.

Atributo de Calidad	**Fiabilidad.**
Sub-atributos/Sub-característica	Tolerancia
Objetivo	El sistema opera según lo previsto en presencia de fallos *hardware* o *software*.
Origen	Administrador del sistema
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo **1. Recuperación ante un fallo del fluido eléctrico.**	**Respuesta: Flujo de eventos (Escenarios)**
Se produce un fallo eléctrico.	1. El sistema se encuentra ejecutándose. 2. El sistema falla por ausencia del fluido eléctrico. 3. El sistema recupera la última puntuación guardada (se guardan automáticamente al perder). 4. El sistema concluye el proceso antes afectado satisfactoriamente.
Medida de respuesta	
Navegar en el sistema en presencia de fallos eléctricos	

10. El sistema está disponible cuando se requiere su uso.

Tabla 14: Escenario 10.

Atributo de Calidad	**Fiabilidad.**
Sub-atributos/Sub-característica	Disponibilidad.
Objetivo	El sistema está disponible cuando se requiere su uso.
Origen	Administrador del sistema.
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Se ejecuta el sistema en diferentes momentos.	
Se ejecuta el sistema en diversas ocasiones.	El sistema funciona eficientemente, ejecutándose todas las veces que se requiere su uso.
Medida de respuesta	
Desplegar el sistema.	

11. El sistema protege contra el acceso a datos e información de personas no autorizadas, ya sea accidental o deliberadamente.

Tabla 15: Escenario 11.

Atributo de Calidad	**Seguridad.**
Sub-atributos/Sub-característica	Confidencialidad.
Objetivo	El sistema protege contra el acceso a datos e información de personas no autorizadas, ya sea accidental o deliberadamente.
Origen	Usuarios del sistema
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Se accede al sistema sin autorización	
Se accede al sistema sin autenticarse.	El sistema no posee autenticación de usuarios.
Medida de respuesta	
Desplegar el sistema.	

12. El sistema previene accesos o modificaciones no autorizados a sus datos.

Tabla 16: Escenario 12.

Atributo de Calidad	**Seguridad.**
Sub-atributos/Sub-característica	Integridad.
Objetivo	El sistema previene accesos o modificaciones no autorizados a sus datos.
Origen	Usuarios del sistema.
Artefacto	Videojuego.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Se accede al sistema sin autorización y se modifican las puntuaciones.	
Se accede al sistema sin autenticarse.	El sistema no posee autenticación de usuarios, por lo que se pueden modificar las puntuaciones.
Medida de respuesta	
Modificar datos del sistema.	

13. Al realizar un cambio en un componente del sistema debe tener un impacto mínimo en los demás.

Tabla 17: Escenario 13.

Atributo de Calidad	**Mantenibilidad.**
Sub-atributos/Sub-característica	Modularidad.
Objetivo	Al realizar un cambio en un componente del sistema debe tener un impacto mínimo en los demás.
Origen	Administrador del sistema.
Artefacto	Videojuego.
Entorno	El sistema.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Se realiza un cambio en un componente del sistema (*aspect ratio* de la cámara).	
Se modifica un componente del sistema	Al modificar un componente, el cambio realizado tiene un impacto mínimo o nulo en los demás, dependiendo de cómo se encuentren conectados, demostrando la extensibilidad de la arquitectura, pues se le pueden agregar o modificar componentes sin perder calidad.
Medida de respuesta	

Modificar componentes del sistema.

14. Los elementos del videojuego deben ser reutilizables en otros desarrollos.

Tabla 18: Escenario 14.

Atributo de Calidad	**Mantenibilidad.**
Sub-atributos/Sub-característica	Reusabilidad.
Objetivo	Los elementos del videojuego deben ser reusables en otros desarrollos.
Origen	Administrador del sistema.
Artefacto	El código fuente.
Entorno	El ambiente de desarrollo del sistema.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Capacidad de modificación del sistema	
La arquitectura del sistema está diseñada para que los elementos sean reutilizables (*NotificationCenter, AspectUtility, Managers*).	N/A.
Medida de respuesta	
Comprobar reutilización de componentes del sistema.	

15. El sistema puede ser modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.

Tabla 19: Escenario 15.

Atributo de Calidad	**Mantenibilidad.**
Sub-atributos/Sub-característica	Capacidad de ser modificado.
Objetivo	El sistema puede ser modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.
Origen	Administrador del sistema.
Artefacto	El código fuente.
Entorno	El ambiente de desarrollo del sistema.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Capacidad de modificación del sistema	
La arquitectura del sistema está diseñada para	N/A.

brindar facilidades a la hora de introducir modificaciones en el sistema. Esto permite que se puedan introducir cambios o modificaciones, que no afecten el correcto desempeño del resto de las funcionalidades de la solución.	
Medida de respuesta	
Introducir una modificación al sistema.	

16. El producto puede ser adaptado de forma efectiva y eficiente a diferentes entornos determinados de *hardware*, *software*, operacionales o de uso.

Tabla 20: Escenario 16.

Atributo de Calidad	**Portabilidad.**
Sub-atributos/Sub-característica	Adaptabilidad.
Objetivo	El producto puede ser adaptado de forma efectiva y eficiente a diferentes entornos determinados de *hardware*, *software*, operacionales o de uso.
Origen	Administrador del sistema.
Artefacto	El videojuego.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Capacidad de adaptación del sistema a diferentes entornos.	
Se prueba el sistema en diferentes sistemas operativos y en diferente *hardware*, para comprobar su adaptabilidad.	El sistema funciona correctamente.
Medida de respuesta	
Desplegar el sistema.	

17. El producto se puede instalar y/o desinstalar de forma exitosa en un determinado entorno.

Tabla 21: Escenario 17.

Atributo de Calidad	**Portabilidad.**
Sub-atributos/Sub-característica	Facilidad de instalación.

Objetivo	El producto se puede instalar y/o desinstalar de forma exitosa en un determinado entorno.
Origen	Administrador del sistema.
Artefacto	El videojuego.
Entorno	Instalación del sistema.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
1. Facilidad de instalación del sistema	
Se comprueba la facilidad de instalación del sistema.	El sistema no cuenta con un instalador, una vez copiado el sistema se puede ejecutar con facilidad.
Medida de respuesta	
Ejecutar el sistema.	

3.5.1.3 Resultados de la evaluación con ATAM

A partir de la información obtenida tras la aplicación del ATAM, se identificaron dos riesgos en la arquitectura propuesta, estos riesgos están asociados a los escenarios 11 y 12.

Tabla 22: Riesgos encontrados.

Escenario	Riesgo
Seguridad: El sistema protege contra el acceso a datos e información de personas no autorizadas, ya sea accidental o deliberadamente.	El sistema no posee autenticación de usuarios, por lo que cualquier usuario puede acceder al mismo.
Seguridad: El sistema previene accesos o modificaciones no autorizados a sus datos.	Comportamiento Temporal.

Luego del análisis de los riesgos detectados se determinó que, en el caso de los riesgos asociados a los escenarios 11 y 12, estos no afectan considerablemente el desempeño de las aplicaciones que se pretenden desarrollar con la arquitectura propuesta y provienen del prototipo funcional, no de la AS, por lo que no se toman en cuenta como riesgos de la arquitectura.

3.5.2 Evaluación basada en prototipo

Para evaluar la AS fue implementado un prototipo funcional de un juego de plataformas en el que, luego de tener una comprensión mejor del mismo mediante las vistas planteadas en el capítulo 2, se puede apreciar la interacción de los componentes del prototipo y con la arquitectura propuesta como solución en cada uno de los mecanismos del videojuego de la manera:

Figura 22: Escena de juego del prototipo funcional

Para ejecutar cada uno de los estados el personaje se vale del componente *animator* de Unity, contando con los siguientes *scripts* para completar las acciones:

ControladorPersonaje: Se encarga de determinar en qué estado se encuentra el personaje y realizar la acción correspondiente. Se encuentra suscrito al *NotificationCenter (Observer)* para posibilitar que en el momento que empiece a correr y, por consiguiente, se generen los elementos y objetos del videojuego.

Para controlar la puntuación se usa:

Puntuación: Controla la puntuación ejecutando las acciones de incrementar 1 punto si el personaje se sitúa sobre un bloque o 5 puntos en caso de que acumule un objeto. Suscrito también al *NotificationCenter,* accede a las tablas de puntos contenidas en el *GameManager*, así como al *DataManager* para obtener la puntuación máxima guardada ahí y al método de guardar para sobrescribirla en caso de que se acumule una puntuación máxima superior.

Figura 23: Menú principal del prototipo funcional

Para interactuar entre escenas se usa el *SceneManager*, que se encarga de los cambios de escenas, siendo implementado ahí los métodos que se encargan de dicho cambio entre escenas.

3.4 Conclusiones parciales

Luego que se le realizaran pruebas a la arquitectura mediante el prototipo funcional y usando técnicas basadas en escenarios y prototipo en conjunto con el método de evaluación ATAM, se pudo comprobar que:

- Como resultado de la evaluación basada en escenarios, prototipos y el análisis subsiguiente, los riesgos y beneficios de la arquitectura, así como los atributos de calidad con los que la propuesta de solución cuenta quedaron documentados.
- Los escenarios representan el cambio esperado, modificaciones del sistema que pueden causar cambios en la arquitectura, y demostraron la eficiencia con que la arquitectura propuesta responde a estos cambios.
- Se pudo validar la propuesta de solución mediante el prototipo funcional, cumplimentando el objetivo principal del capítulo.

Conclusiones generales

Con la realización de este trabajo, se definió y se validó una arquitectura de software para el desarrollo de videojuegos, probada sobre un prototipo funcional de un videojuego de plataformas, que permite reutilizar componentes, facilitar implementación y uso de recursos generando mejores desarrollos futuros. De esta forma se dio cumplimiento al objetivo propuesto al inicio de la investigación, además:

- El análisis del estado del arte, específicamente de las AS de videojuegos desplegados a nivel internacional y en el centro Vertex, en conjunto con los patrones de diseño y estilos arquitectónicos usados para su desarrollo, permitió la incorporación de elementos comunes y reutilizables que se tuvieron en cuenta a la hora de proponer una AS para videojuegos usando el motor de videojuegos Unity 3D.
- La combinación de los estilos arquitectónicos: Arquitectura basada en capas, Arquitectura basada en componentes, permitió desarrollar una AS que satisface los atributos de calidad: reusabilidad, mantenibilidad, extensibilidad y eficiencia, que eran pautas a tener en cuenta a la hora de proponer una solución.
- El uso de las vistas propuestas por Robert Nord, en conjunto con los acápites especificación de mecanismos y diseño del videojuego (basados en los documentos del mismo nombre), permitió tener una mayor visión de la propuesta de solución implementada en el prototipo funcional.
- Las pruebas realizadas, mediante el método ATAM y basadas en escenarios y prototipo, arrojaron la presencia de riesgos en el prototipo funcional, que fueron registrados.

Recomendaciones

Para darle seguimiento a la solución propuesta se recomienda:

- Extender la arquitectura propuesta para todo videojuego que se desarrolle sobre el motor de juego Unity 3D.
- Contribuir a la aplicación práctica de la arquitectura con la definición de un estándar de codificación que permita utilizar la misma estructura y lógica en la implementación de videojuegos.
- Definir un componente de seguridad genérico para el desarrollo de videojuegos que pueda ser reutilizable.

Referencias bibliográficas

[1] Patrick Stack. «History of video game consoles». Time Magazine website.

[2] «Gamespeak: A glossary of gaming terms». Specusphere.

[3] What is a Game Engine? GameCareerGuide.com

[4] Pressman, Roger S. Ingeniería del Software. Un enfoque prático. S.l.: Connecticut, 2002.

[5] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd Edition, Addison Wesley, 2003

[6] IEEE Computer Society. 1471-2000 - Recommended Practice for Architectural Description for Software-Intensive Systems. 2000.

[7] Garlan, D. y Perry, D. E. Special Issue on Software Architecture. S.l.: IEEE Transctions on Software Engineering, 1995.

[8] Garlan, D. y Perry, D. E. Advances in Software Engineering and Knowledge Engineering. In V. Ambriola and G. Tortora (ed.), Series on Software Engineering and Knowledge Engineering, Vol 2, World Scientific Publishing Company, Singapore, pp. 1-39, 1993

[9] Kruchten, Philippe. The Rational Unified Process. S.l.: Addison Wesley Longman, 2003.

[10] Christine Hofmeister; Philippe Krutchen; Robert L. Nord; Henk Obbink; Alexander Ran; Pierre de América (2007). "Un modelo general de diseño de arquitectura de software derivado de cinco enfoques industriales".

[11] Patrick Stack. History of video game consoles, Time Magazine website 2005.

[12] Specusphere. Gamespeak: A glossary of gaming terms.

[13] Television gaming apparatus and method. United States Patents.

[14] Reynoso, Carlos. Carlos Reynoso Investigación, Publicaciones y Cursos de Antropología, Ciencia Cognitiva y Complejidad. [Online] 2004. carlosreynoso.com.ar/arquitectura-de-software/.

[15] Rollings, A. y Morris, D. Game Architecture and Design. Coriolis Group, 1999.

[16] Procesos de desarrollo para videojuegos, Gerardo Abraham Morales Urrutia, Claudia Esther Nava López, Luis Felipe Fernández Martínez, y Mirsha Aarón Rey Corral.

[17] Documentación oficial de Unity <http://docs.unity3d.com/es/current/Manual/index.html>.

[18] Reynoso, Carlos y Kicillof, Nicolás. Estilos y Patrones en la Estrategia de Arquitectura de Microsoft, Marzo de 2004.

[19] Kruchten, Philippe. Architectural Blueprints, The “4+1” View Model of Software Architecture. Noviembre 1995 pp.42-50

[20] González, Omar. Documentando la arquitectura de software.

[21] Mary Shaw y Paul Clements. “A field guide to Boxology: Preliminary classification of architectural styles for software systems”. Documento de Computer Science Department and Software Engineering Institute, Carnegie Mellon University. Abril de 1996. Publicado en Proceedings of the 21st International Computer Software and Applications Conference, 1997.

[22] Mark Klein y Rick Kazman. “Attribute-based architectural styles”. Technical Report, CMU/SEI-99-TR-022, ESC-TR-99-022, Carnegie Mellon University, Octubre de 1999.

[23] Clemens Alden Szyperski. Component software: Beyond Object-Oriented programming. Reading, Addison-Wesley, 2002.

[24] Buschman, Frank, et al. Pattern-Oriented Software Architecture - A System of Patterns. Germany: Siemens AG, 1996. Vol. 1. ISBN: 0 471 95869 7

[25] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, A Pattern Language. New York: Oxford University Press, 1977.

[26] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael. Pattern-oriented software architecture – A system of patterns. 1996: John Wiley & Sons.

[27] Roberto Canales Mora, Patrones de Grasp, Diciembre 22, 2003.

[28] Godoy Barrera, Pedro Francisco, Metodologías de diseño e implementación, 2012-2013.

[29] Ernest Adams, Fundamentals of Game Design.

[30] Ricardo Emmanuel Gutiérrez-Hernández, Francisco J. Álvarez, Jaime Muñoz-Arteaga, Arquitectura de Software para Juegos Serios con Aspectos Culturales: Caso de Estudio en un Videojuego para Fórmulas Temperatura.

[31] Carlota Esteban Cazalla. Diseño y desarrollo de un prototipo básico de un videojuego plataformas en 2D. Diciembre de 2014.

[32] Ing. Andy Hernández Páez, Permiso de no utilización de artefactos, 26/06/2015.

[33] Demián Gutierrez, UML Diagramas de Paquetes (UML ilustrado) ,Septiembre 2009.

[34] UML El Lenguaje Unificado de Modelado, Grady Booch, Jim Rumbaugh e Ivar Jacobson.

[35] Ingeniería del Software. Curso 2013-2014, Guión Visual Paradigm for UML.

[36] Huanay Martínez, Franz, Tutorial Básico – UNITY.

[37] Marco Besteiro y Miguel Rodríguez, Introducción a la programación con C#.

[38] Tracy Fullerton, Game Design Workshop-A playcentric approach to creating innovative games-2nd Edition.

[39] Fabricatore, Carlo. Gameplay and Game Mechanics Desing: A Key to Quality in Video Games.

[40] Clements, Paul, Kazman, Rick and Klein, Mark. Evaluating software architectures: methods and case studies. s.l. : Addison-Wesley, 2002. ISBN: 9780201704822.

[41] http://iso25000.com/index.php/normas-iso-25000.

[42] Quality Characteristics for Software Architecture. Losavio, Francisca, et al. Marzo-Abril 2003, Journal of Object Technology, Vols. 2, no.2, pp. 133-150.

[43] Bosh, Jan. Design & Use of Software Architectures. Adopting and evolving a product-line approach. S.l.: Addison-Wesley Professional, 2000. ISBN: 978-0201674941.

[44] ISO/IEC 9126-1. (2001). Software engineering — Product quality — Part 1: Quality model.

[45] Bosch, Jan. Design & Use of Software Architectures. Addison-Wesley.

[46] Rick Kazman, Mark Klein, Paul Clements. Evaluating software architectures: methods and case studies. s.l.: Addison-Wesley, 2002. ISBN 9780201704822.

Anexos

Modelo Vista Controlador

El patrón conocido como Modelo-Vista-Controlador (MVC) separa el modelado del dominio, la presentación y las acciones basadas en datos ingresados por el usuario en tres clases diferentes [11]:

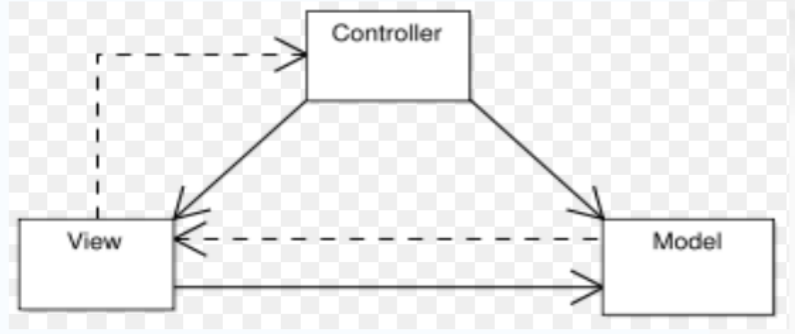


Figura 24: Modelo Vista Controlador

- **Modelo:** El modelo administra el comportamiento y los datos del dominio de aplicación, responde a requerimientos de información sobre su estado (usualmente formulados desde la vista) y responde a instrucciones de cambiar el estado (habitualmente desde el controlador).
- **Vista:** Maneja la visualización de la información.
- **Controlador:** Interpreta las acciones del ratón y el teclado, informando al modelo y/o a la vista para que cambien según resulte apropiado.

Tanto la vista como el controlador dependen del modelo, el cual no depende de las otras clases. Esta separación permite construir y probar el modelo independientemente de la representación visual. La separación entre vista y controlador puede ser secundaria en aplicaciones de clientes ricos y, de hecho, muchos *frameworks* de interfaz implementan ambos roles en un solo objeto.

Entre las **ventajas** del estilo señaladas están las siguientes [11]:

- Soporte de vistas múltiples. Dado que la vista se halla separada del modelo y no hay dependencia directa del modelo con respecto a la vista, la interfaz de usuario puede mostrar múltiples vistas de los mismos datos simultáneamente. Por ejemplo, múltiples páginas de una aplicación de Web pueden utilizar el mismo modelo de objetos, mostrado de maneras diferentes.
- Adaptación al cambio. Los requerimientos de interfaz de usuario tienden a cambiar con mayor rapidez que las reglas de negocios. Los usuarios pueden preferir distintas opciones de representación, o requerir soporte para nuevos dispositivos como teléfonos celulares o PDAs.
- Dado que el modelo no depende de las vistas, agregar nuevas opciones de presentación generalmente no afecta al modelo.

Entre las **desventajas**, se han señalado:

- Complejidad. El patrón introduce nuevos niveles de indirección y por lo tanto aumenta ligeramente la complejidad de la solución. También se profundiza la orientación a eventos del código de la interfaz de usuario, que puede llegar a ser difícil de depurar. En rigor, la configuración basada en eventos de dicha interfaz corresponde a un estilo particular (arquitectura basada en eventos) que aquí se examina por separado.
- Costo de actualizaciones frecuentes. Desacoplar el modelo de la vista no significa que los desarrolladores del modelo puedan ignorar la naturaleza de las vistas. Si el modelo experimenta cambios frecuentes, por ejemplo, podrían desbordar las vistas con una lluvia de requerimientos de actualización. Hace pocos años sucedía que algunas vistas, tales como las pantallas gráficas, involucraban más tiempo para plasmar el dibujo que el que demandaban los nuevos requerimientos de actualización.

AS del videojuego Aventuras en la Manigua

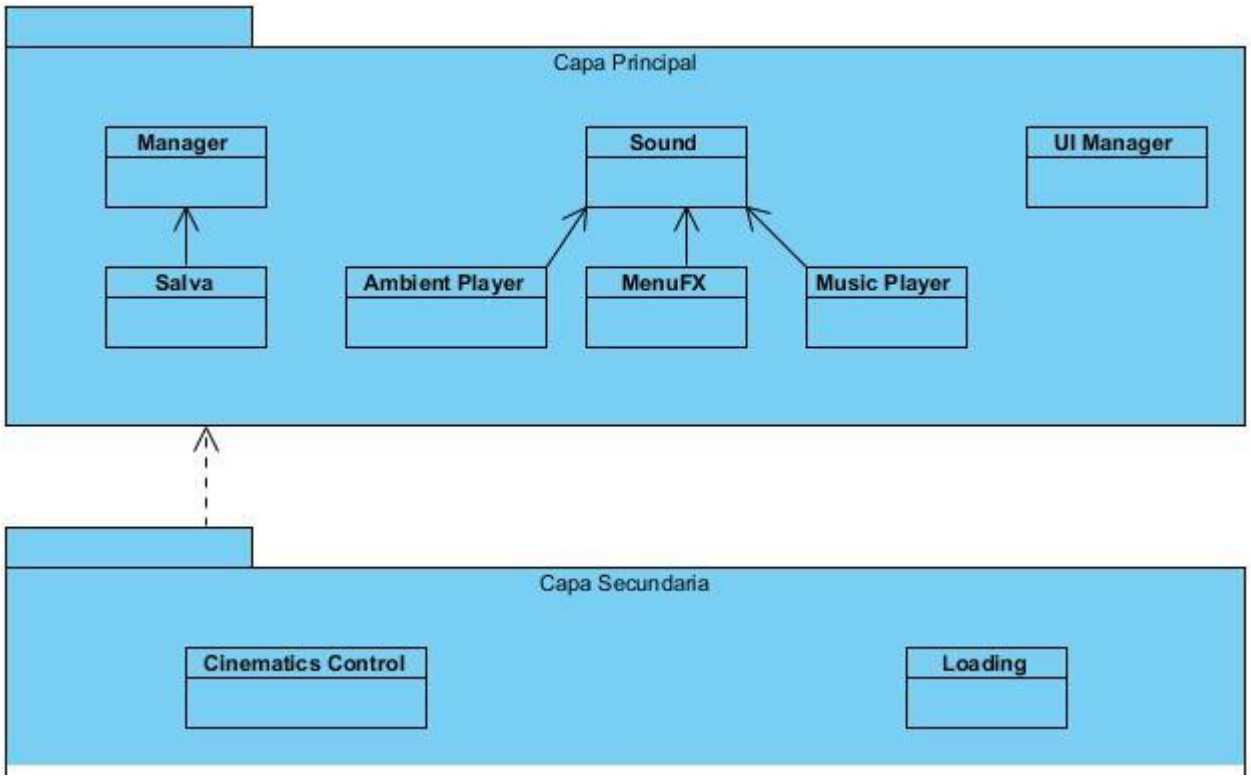


Figura 25: AS de Aventuras en la manigua.

Las clases de la **Capa Principal** se encargan de:

Manager: Para acceder al manager principal del juego se usa un *singleton*. Tiene una referencia de la clase Salva. Es el encargado de salvar y guardar los datos del juego. Controla la carga de las escenas.

*Ambient Player***:** Controla el audio ambiente del juego.

*MenuFx***:** Controla el audio del menú.

Music Player: Controla el audio del personaje principal.

Sound: clase para manejar el sonido que hace uso del *Ambient Player*, *MenuFx* y *Music Player*.

Player movement: Controla todo lo relacionado con el movimiento del personaje.

UI_Manager: Se encarga de controlar las interfaces de usuario. Posee los siguientes elementos:

```
public static UI_Manager Instance;
public GameObject Main_Camera;
public GameObject Black_Background;
public GameObject Pause_Menu;
public GameObject Pause_Confirmation_Dialog;
public GameObject Map_Menu;
public GameObject Credits_Menu;
public GameObject Options_Menu;
public GameObject Tutorial_Menu;
```

Las clases de la **Capa Secundaria** se encargan de:

CinematicsControl: Controla la reproducción de las cinemáticas. Tiene un *singleton* implementado.

Loading: Se emplea para la carga de las escenas. Tiene un *singleton* implementado.

Figura 26: Rendimiento del sistema determinando usando la herramienta Profiler de Unity 3D.