Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computadores

Profesor: Marco Hernández Vásquez

Curso: Principios de modelado para ingeniería

Estudiantes:

Marín Gutiérrez Emanuel – 2019067500

Rodríguez Rojas Jose Andrés – 2019279722

Soto Varela Óscar – 2020092336

Vargas Jiménez Felipe – 2020211831

Patrones de diseño

5 de junio de 2024

I Semestre – 2024

Singleton

El patrón singleton es un patrón de diseño creacional que asegura que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. Este patrón es especialmente útil en situaciones donde se necesita un único punto de control, como en administradores de recursos o configuraciones globales. La implementación del singleton se logra mediante un constructor privado y un método estático que devuelve la única instancia de la clase. Este método puede incluir mecanismos de sincronización para asegurar que solo se crea una instancia, incluso en entornos multihilo.

Las principales ventajas del patrón singleton incluyen el control centralizado de acceso, la consistencia en la gestión de recursos y la flexibilidad para extender funcionalidades sin modificar el código cliente. Al tener una única instancia, se evita la creación de múltiples objetos que podrían llevar a inconsistencias y sobrecarga, especialmente en la gestión de recursos críticos como conexiones de base de datos o impresoras. Además, permite un acceso global a esta instancia, facilitando su uso en diferentes partes del programa sin necesidad de crear nuevas instancias.

Sin embargo, el patrón singleton también presenta desventajas significativas. Una de las principales es la dificultad en las pruebas unitarias, ya que sustituir el singleton por una versión de prueba puede ser complicado. Además, el patrón introduce un punto único de falla: si la instancia única tiene un error o es mal diseñada, todo el sistema que depende de ella se verá afectado. En entornos multihilo, asegurar que solo se crea una instancia puede ser complejo y requerir mecanismos de sincronización, lo que puede impactar el rendimiento del sistema.

Para mitigar algunos de estos problemas, existen variantes del patrón singleton como el singleton eager y el double-checked locking. El singleton eager crea la instancia en el momento de la carga de la clase, garantizando que siempre esté disponible cuando se necesite. Por otro lado, el double-checked locking utiliza bloqueos para reducir el costo de sincronización y asegurar que la instancia sea creada solo una vez, incluso en entornos multihilo.

En la práctica, el patrón singleton se aplica comúnmente en la administración de recursos, configuración global y registro de logs. En la administración de recursos, como conexiones

de base de datos, es crucial mantener una única instancia para evitar conflictos y asegurar una gestión uniforme. En la configuración global, el patrón singleton permite almacenar configuraciones accesibles desde diferentes partes del programa. En el registro de logs, garantiza que todos los mensajes se manejen de manera consistente y centralizada. En resumen, el patrón singleton es una solución poderosa, pero debe ser utilizado con cuidado para evitar sus potenciales desventajas.

El patrón es utilizado en el gameManeger.cs, con él se guardan variables o se puede acceder a métodos que sirven de base para el juego y tener control de la partida. Por ejemplo con el gameManager se puede saber qué jugador está actualmente jugando o editando su perfil, también se puede saber si se acaba de finalizar una partida o los patrones que fueron configurados para esta.

State

El patrón State es un patrón de diseño de comportamiento que permite a un objeto cambiar su comportamiento cuando cambia su estado interno, pareciendo que el objeto cambia su clase. Este patrón es especialmente útil cuando un objeto debe cambiar su comportamiento en función de su estado, eliminando la necesidad de utilizar una gran cantidad de condicionales que verifican el estado actual del objeto.

El patrón State se implementa mediante la creación de una interfaz común para todos los posibles estados del objeto. Cada estado concreto implementa esta interfaz y define el comportamiento específico para ese estado. El objeto principal, conocido como el contexto, mantiene una referencia a un objeto estado que representa su estado actual. El contexto delega las operaciones a su estado actual, permitiendo que el comportamiento cambie dinámicamente cuando el estado cambia.

Una de las principales ventajas del patrón State es que facilita el mantenimiento y la extensión del código. Al encapsular los comportamientos relacionados con un estado particular en sus propias clases, el código se vuelve más modular y fácil de entender. Además, agregar nuevos estados es sencillo y no requiere modificar las clases existentes, lo que mejora la extensibilidad del sistema. Esto también reduce la complejidad al eliminar las estructuras de control condicionales extensas, como los bloques if-else o switch-case, que pueden ser difíciles de gestionar y propensos a errores.

Sin embargo, el patrón State también tiene algunas desventajas. La principal es el aumento en el número de clases, ya que cada estado y cada posible transición entre estados requieren su propia clase. Esto puede llevar a una sobrecarga administrativa y hacer que el sistema sea más complejo de gestionar. Además, puede ser difícil de implementar en sistemas donde los estados y las transiciones son dinámicos o no están bien definidos desde el principio.

En la práctica, el patrón State se aplica en situaciones donde un objeto debe cambiar su comportamiento en función de su estado. Un ejemplo clásico es el de una máquina de estados finitos, como una máquina expendedora o un semáforo, donde cada estado (esperando moneda, seleccionando producto, entregando producto) tiene un comportamiento distinto que debe ser gestionado de manera ordenada y predecible.

Memento

El patrón Memento es un patrón de diseño de comportamiento que permite capturar y externalizar el estado interno de un objeto sin violar su encapsulamiento, de manera que el objeto pueda ser restaurado a este estado más adelante. Este patrón es útil para implementar funcionalidades de deshacer/rehacer en aplicaciones, como editores de texto, donde es crucial poder revertir cambios y mantener la integridad del estado del objeto.

La implementación del patrón Memento involucra tres componentes principales: el Originator, el Memento y el Caretaker. El Originator es el objeto cuyo estado se desea guardar. El Memento es una representación del estado interno del Originator y es responsable de almacenar esta información. El Caretaker es responsable de solicitar al Originator la creación de un Memento, almacenarlo y, en caso necesario, devolver el Memento al Originator para restaurar su estado.

Una de las principales ventajas del patrón Memento es que preserva el encapsulamiento del estado del objeto. Los detalles internos del estado del Originator se almacenan en el Memento y no se exponen a otros objetos. Esto facilita la implementación de operaciones de deshacer y rehacer, mejorando la usabilidad y la seguridad de la aplicación. Además, permite almacenar múltiples estados de un objeto, lo que puede ser útil en aplicaciones complejas que requieren seguimiento detallado de cambios.

Sin embargo, el patrón Memento también tiene desventajas. La principal es el posible aumento en el uso de memoria, ya que cada Memento puede almacenar una copia completa del estado del Originator. En aplicaciones con estados complejos o grandes, esto puede llevar a un consumo significativo de recursos. Además, la gestión de múltiples Mementos puede complicar el código y requerir una cuidadosa planificación para evitar problemas de rendimiento y manejo de memoria.

En la práctica, el patrón Memento se utiliza en aplicaciones que requieren funcionalidades de deshacer y rehacer, como editores de texto, aplicaciones de dibujo, y sistemas de gestión de versiones. Por ejemplo, en un editor de texto, cada vez que un usuario realiza un cambio, se puede crear un Memento que almacena el estado actual del documento, permitiendo que el usuario deshaga los cambios y vuelva a estados anteriores del documento según sea necesario.

Los patrones State y Memento se usan en conjunto para poder alternar la partida, se puede guardar el estado de un jugador (como tomar un screenshot justo en el momento que pierde o hay una colisión) y a su vez restaurar el estado previo del otro jugador para así poder retomar dónde o cómo quedó el jugador antes de perder una vida. En realidad es muy útil porque simplifica la alternancia entre los jugadores.

Adapter

El patrón Adapter es un patrón de diseño estructural que permite que clases con interfaces incompatibles trabajen juntas. Funciona como un puente entre dos interfaces, adaptando la interfaz de una clase a lo que el cliente espera. Este patrón es especialmente útil cuando se necesita integrar clases que no fueron diseñadas para trabajar juntas, sin cambiar su código original.

La implementación del patrón Adapter involucra la creación de una clase adaptadora que implementa la interfaz esperada por el cliente y traduce las llamadas a los métodos de una clase existente que tiene una interfaz diferente. Existen dos tipos principales de adaptadores: el adaptador de objeto y el adaptador de clase. El adaptador de objeto utiliza la composición para referenciar la instancia de la clase que necesita ser adaptada, mientras que el adaptador

de clase utiliza la herencia para extender tanto la interfaz esperada como la clase que necesita ser adaptada.

Una de las principales ventajas del patrón Adapter es su capacidad para reutilizar clases existentes sin modificar su código, lo que facilita la integración de componentes de software desarrollados independientemente. Además, permite que el código cliente trabaje con una interfaz uniforme, simplificando la lógica del cliente y mejorando la coherencia del sistema. El patrón Adapter también favorece el principio de inversión de dependencia, ya que el cliente depende de abstracciones (interfaces) en lugar de implementaciones concretas. Sin embargo, el patrón Adapter también tiene algunas desventajas. La principal es la introducción de complejidad adicional al sistema, ya que se agrega una capa extra de direccionamiento. Esto puede hacer que el código sea más difícil de entender y mantener. Además, si se utilizan muchos adaptadores en un sistema, puede llevar a un diseño fragmentado donde es difícil seguir las relaciones entre las clases.

En la práctica, el patrón Adapter se utiliza en situaciones donde se necesita integrar librerías de terceros con interfaces diferentes, o cuando se quiere utilizar una clase existente con una nueva interfaz. Por ejemplo, en un sistema de pagos, se podría utilizar un adaptador para que una nueva API de pagos se ajuste a la interfaz utilizada por el sistema existente. Otro ejemplo es en aplicaciones de interfaz gráfica, donde un adaptador puede permitir que diferentes tipos de controles gráficos (botones, campos de texto, etc.) trabajen juntos bajo una interfaz común.

En resumen, el patrón Adapter es una solución efectiva para hacer que clases con interfaces incompatibles trabajen juntas sin necesidad de modificar su código original, favoreciendo la reutilización de código y la flexibilidad en el diseño del sistema.

El patrón Adapter es utilizado para adaptar la información que provee el user,json a la que ocupa el hallOfFame.json para poder mostrar o actualizar el top 5 de las mejores 5 partidas del juego.