Universidad Cenfotec

Autor(es):

- Isabel Galeano Hernández, Cédula: 1-1888-0968
- José García Quirós, Cédula: 1-1782-0076
- Orlando Trejos Montano, Cédula: 7-0281-0993
- Daniel Zúñiga Rojas, Cédula: 1-1811-0097
- Isaías Arce Rodríguez, Cédula: 305250698
- Francisco Tovar Araya, Cédula: 1-1072-0821

Tema:
Documentación Proyecto Final

Curso:
BISOFT-12 Programación con Patrones

Profesor:
Erick Alfonso Brenes Umaña

Fecha de entrega:
11/08/2021

Cuatrimestre:
2021-2

Índice

Abstract

El desarrollo de software crece con el pasar de los años y cada vez se mejora la manera en la que este mismo se desarrolla, tanto en su ámbito conceptual como en la práctica, entre tantas mejoras se debe tomar en cuenta siempre que se pueda, el desarrollo de software con la utilización de patrones.

Estos patrones de diseño de software son una tipo de plantilla que nos ayuda a resolver problemas de una manera más eficiente, debido a que muchas veces son los mismos o similares los problemas que se deben resolver y que ya alguien más ha resuelto de una manera similar, estos patrones ayudan a lograr una solución adecuada a esos problemas que son tan comunes y pueden llegar a ahorrar hasta horas de trabajo.

Debido a que el desarrollo de software con la utilización de patrones no es tan común y resulta muchas veces un poco complicado o difícil, cuesta encontrar empresas u organizaciones que los implementen, sin embargo, el mantenimiento y la adición de nuevas funcionalidades al código muchas veces resulta ser más sencilla utilizándolos.

El total de estos patrones (en agosto 2021) son 23 y no está demás recalcar que estos patrones están enfocados a la programación orientada a objetos (POO) por sus siglas en inglés.

Estos 23 patrones se dividen en 3 secciones:

Patrones Creacionales: Suelen ser muy útiles cuando se necesita crear objetos, esto permite flexibilidad al sistema.

Patrones Estructurales: Facilitan soluciones y estándares eficientes con respecto a las composiciones de clase y las estructuras de objetos.

Patrones de Comportamiento: Estos facilitan la comunicación entre objetos y cómo estos deben comportarse e integrarse con el resto del sistema.

Documentación Patrones

I. Patrones creacionales

A. Fábrica Abstracta

Razones de uso: El patrón se adapta bien para la implementación de la creación de personajes
Parte del código donde se utiliza: Paquetes: patronFabrica,gestores, controllers.
CompraPersonajes.js

Nombre	Fábrica Abstracta
Propósito	Crear objetos con características predeterminadas
Problema	Crear personajes que tienen una lista de atributos ya definidos
Solución	Se implementa una Fábrica Abstracta que genera objetos dependiendo de la opción solicitada.
Participantes y Colaboradores	
Fábrica Abstracta	IPersonajesJuego
Fábrica Concreta	FabricaArquero ,FabricaEspadachin, FabricaBerserquer, FabricaEspia, FabricaMago, FabricaAsesino, FabricaJinete
Producto Abstracto	Personaje
Producto Concreto	Arquero, Espadachin, Berserquer, Espia, Mago, Asesino, Jinete
Cliente	Cliente
Consecuencias	
Las definiciones específicas de cada producto(personaje) quedan aisladas del cliente	

El sistema es el único que tiene manera de accesar la fábrica que se solicite desde el frontend
Implementación
Se define la interfaz IPersonajesJuego, esta es implementada por las fábricas de cada personaje distinto(fabricaArquero, fabricaEspadachin...) .
Cuando desde el frontend se hace una petición al sistema de crear un personaje el GestorFabricaAbstracta recibe un parámetro que representa la opción de personaje.
Dentro del gestor cada Fábrica Concreta tiene asignado un número del uno al siete, el gestor va a realizar un llamado a una Fábrica Concreta que tenga asignado un número coincida con el parámetro recibido desde el cliente.

B. Patrón Prototipo

Razones de uso: El patrón se adapta muy bien a la necesidad de creación de muchos objetos similares
Parte del código donde se utiliza: Paquetes: patronPrototipo,controllers,gestores.
Tablero.js

Nombre	Patrón Prototipo
Propósito	Crear gran cantidad de objetos de manera rápida
Problema	Se necesita crear cien objetos Casilla.
Solución	Se implementa el Patrón Prototipo para crear casillas clonando un objeto inicial.
Participantes y Colaboradores	
i-Prototipo	Casilla
Prototipo	CasillaNormal,CasillaGema,CasillaPowerUp

Prototipo Manager	GestorCasilla
Consecuencias	
La creación de nuevos objetos es más eficiente ya que se hace a partir de la clonación de un objeto que ya se encuentra en memoria.	
Se crean una gran cantidad de objetos Casilla idénticos y se realizan los cambios de atributos para satisfacer los requerimientos del sistema.	
Implementación	
La clase abstracta Casilla funciona como base de los prototipos concretos que van a ser clonados.	
Al ingresar la cantidad de jugadores en el cliente este envía una solicitud a GestorCasilla para que cree la cantidad requerida de casillas.	
Se crean catorce objetos CasillaPowerUp, quince CasillaGema y setenta y un objetos CasillaNormal	
Estos objetos se almacenan en arrays separados y se hace un cambio de datos aleatorio. Los arrays son integrados en uno solo y se envía como respuesta al cliente.	

II. Patrones Estructurales

A. Proxy

Razones de uso: Es una buena manera de ocultar los datos del jugador, creando una única vía de acceso al objeto.
Parte del código donde se utiliza: Paquete proxy, controllers y gestores.
turnos.js

Nombre	Patrón Proxy
Propósito	Controlar el acceso y creación a otros objetos por medio de un objeto mediador.
Problema	Cambiar de turno durante la partida, cambiando a su vez la información del jugador activo en pantalla
Solución	Se genera el código implementando el patrón Proxy

Participantes y Colaboradores	
Proxy	JugadorProxy
Sujeto	IJugador
SujetoReal	Jugador
Consecuencias	
La información del jugador queda encapsulada y es solo accesible por medio del proxy jugador.	
Otras partes del sistema desconocen dónde reside la definición de la clase jugador y la información de un jugador específico.	
Implementación	
En el controlador se instancia un JugadorProxy.	
El controlador recibe un array de jugadores desde el frontend y el proxy itera sobre el array de jugadores para accesar la información real, esta iteración depende del jugador activo en el turno.	

B. Decorador

Razones de uso: Cambiar los datos de un objeto Personaje al cumplirse una funcionalidad
Parte del código donde se utiliza: Paquetes decorador, cadenaResponsabilidad, controllers y gestores

Nombre	Patrón Decorador
Propósito	Proporciona una forma de vincular dinámicamente un nuevo estado y comportamiento a un objeto. El objeto no sabe que está siendo "descalificado", lo que hace que este sea un patrón útil para sistemas en evolución.
Problema	Desarrolle un juego de estrategia (Reinos Cenfotecos).

Solución	Se crea parte del código utilizando el patrón Decorador.
Participantes y Colaboradores	
Componentes	IPower.
Decorador	ObjetoDecorador.
Decorador Concreto	PowerDownAtaque, PowerDownDefensa, PowerUpAtaque, PowerUpDefensa.
Gestores	GestorDecorador.
api	ControllerDecorador.
Consecuencias	
El objeto original desconoce cualquier decoración.	
No hay una gran clase cargada de funciones con todas las opciones en ella.	
Las decoraciones son independientes entre sí.	
Implementación	
El GestorCadena hace los llamados al Decorador cuando se maneja una casilla powerUP	
Dependiendo del tipo de mejora se llama a un decorador concreto para que realice los cambios sobre el personaje.	

III. Patrones de Comportamiento

A. Cadena de Responsabilidad

Razones de uso: Dependiendo de los valores que le pasamos por parámetro, el patrón ejecutará una funcionalidad en específico con respecto a la información.
Parte del código donde se utiliza: Paquete cadenaResponsabilidad, gestores y controllers.
PowerUp.js

Nombre	Cadena de Responsabilidad

Propósito	Este patrón funciona con una lista de objetos de Handler que tienen limitaciones en la naturaleza de las solicitudes que pueden tratar. Si un objeto no puede tener una solicitud, pasa al siguiente objeto de la cadena. Al final de la cadena, puede haber un comportamiento predeterminado o excepcional.
Problema	Desarrolle un juego de estrategia (Reinos Cenfotecos).
Solución	Se crea parte del código utilizando el patrón de Cadena de Responsabilidad.
Participantes y Colaboradores	
abstracto	Manejador.
concreto	ManejadorGema, ManejadorNormal, ManejadorPowerUp.
Gestores	GestorCadena
api	ControllerCadena
Consecuencias	
Mayor distribución de responsabilidades entre objetos.	
La recepción de la solicitud no está garantizada.	
La cadena puede construirse con una fábrica abstracta.	
Implementación	
La clase abstracta Manejador contiene el método *manejar().* Los manejadores concretos heredan de esta clase y tienen una funcionalidad específica en el método *manejar()*.	
Los manejadores tienen un atributo autorreferencial(*Manejador nextInChain*) que señala al siguiente manejador en la cadena.	
La cadena configurada recibe un personaje y va pasando desde el primer elemento. Si el tipo de casilla asociada al personaje coincide con el tipo de manejador actual este realiza cambios al personaje, si no es así lo pasa al siguiente elemento de la cadena.	

B. Visitante

Razones de uso: El patrón se adapta a la necesidad de remover mejoras de personajes al final de cada turno
Parte del código donde se utiliza: Paquete patronVisitante, controllers y gestores.

Turnos.js

Nombre	Patrón Visitante
Propósito	El patrón de visitante define y realiza nuevas operaciones en todos los elementos de una estructura existente, sin alterar sus clases. Separa la estructura de un objeto de las operaciones que actúan en una estructura.
Problema	Desarrolle un juego de estrategia (Reinos Cenfotecos).
Solución	Se crea parte del código utilizando el patrón Visitante.
Participantes y Colaboradores	
abstracto	IVisitor.
concreto	RemoverPowerDownAtaque, RemoverPowerDownDefensa, RemoverPowerUpAtaque, RemoverPowerUpDefensa.
Gestores	GestorVisitante.
api	ControllerVisitante.
Consecuencias	
Es fácil añadir nuevas operaciones (el visitante contiene el código en lugar de cada una de las clases individuales).	
Los visitantes pueden recoger las operaciones relacionadas en una sola clase en lugar de obligar a cambiar o derivar clases para agregar estas operaciones.	
Encapsular buscando datos desde un número de instancias de varias clases.	
Implementación	
El patrón se implementa para completar la mecánica de cambios de estadísticas a los personajes por medio de mejoras o casillas trampa.	
Se utiliza una interfaz visitante que tiene el método *visit()*.	
Se definen visitantes concretos que implementan la interfaz visitante. Cada método *visitar()* realiza un cambio de estadísticas diferente sobre un personaje o un array de personajes.	

Batería de Pruebas

Prueba	Fecha	Prioridad	Responsable	Resultado
Generar las 100 casillas en el tablero, tanto normales como con contenido	07/07/2021	Alta	Isaías	Prueba exitosa.
Permitir esconder gemas en el tablero.	10/08/2021	Alta	Isabel	Prueba exitosa.
Establecer 71 casillas regulares las cuales su única función es permitir que el personaje avance.	30/07/2021	Alta	Isabel	Prueba exitosa.
Permitir a las casillas PowerUp potenciar el ataque del personaje durante un turno. La cual es válida una única vez. Aumenta el ataque del personaje en 2.	08/08/2021	Alta	Orlando	Prueba exitosa
Permitir a las casillas PowerUp potenciar la defensa del personaje durante un turno. Se activa una sola vez y aumenta la defensa del personaje en 2 puntos.	08/08/2021	Alta	Orlando	Prueba exitosa
Permitir establecer casillas PowerDown disminuir el ataque de un personaje, solo puede activarse una vez, disminuye el ataque del personaje en 2 por un turno.	08/08/2021	Alta	Orlando	Prueba exitosa
Permitir establecer casillas PowerDown disminuir la defensa de un personaje, solo puede activarse una vez, disminuye la defensa del personaje en 1 por un turno.	08/08/2021	Alta	Orlando	Prueba exitosa

Establecer 15 casillas con gema.	30/07/2021	Alta	Isabel	Prueba exitosa.
Distribuir las 15 casillas con gemas aleatoriamente al inicio del juego.	30/07/2021	Alta	Isabel	Prueba exitosa.
Establecer castillos en el tablero, de acuerdo a la cantidad de jugadores	04/08/2021	Alta	Isabel	Prueba exitosa
Inicializar el castillo de cada jugador	04/08/2021	Alta	Isabel	Prueba exitosa
Permitir al castillo tener un máximo de 2 ballestas.	06/08/2021	Alta	Isaías	Prueba exitosa
Permitir al castillo tener un máximo de una catapulta.	06/08/2021	Alta	Isaías	Prueba exitosa
Establecer la ballesta como una defensa propia del castillo.	01/08/2021	Alta	Tovar	Prueba exitosa
Colocar la ballesta en el castillo	01/08/2021	Alta	Tovar	Prueba exitosa
Establecer la catapulta como la defensa más alta propia del castillo.	01/08/2021	Alta	Tovar	Prueba exitosa
Colocar la catapulta en el castillo.	06/08/2021	Alta	Isaías	Prueba exitosa
Permitir al personaje avanzar la cantidad de espacios indicados por el dado.	10/08/2021	Alta	Daniel	Prueba exitosa
No deberá permitir a los personajes moverse de forma diagonal solo de forma recta.	09/08/2021	Alta	Daniel	Prueba fallida
Deberá establecer que en caso de que el número del dado sea mayor a la cantidad de movimientos permitidos por el	10/08/2021	Alta	Daniel	Prueba fallida

personaje, se avanza al máximo permitido de ese personaje y se usa otro para los movimientos restantes.				
Deberá acabar el turno del personaje al atacar o al quedarse sin movimientos.	10/08/2021	Alta	Daniel	Prueba fallida
Permitir establecer un costo por cada personaje.	05/08/2021	Alta	Orlando	Prueba exitosa
No deberá permitir a los personajes llevar consigo más de una mejora de PowerUp.	08/08/2021	Alta	Jose	Prueba exitosa
Permitir a cada uno de los personajes poseer características especiales. (Espía)	08/08/2021	Alta	Jose	Prueba fallida
No deberá permitir a todos los personajes tener la capacidad de extraer oro de un castillo enemigo.	08/08/2021	Alta	Jose	Prueba fallida

Puntuación Integrantes

Integrante	Puntuación
Orlando Trejos	10
José García	10
Isaías Arce	10
Isabel Galeano	10

Daniel Zuñiga	10
Francisco Tovar	10