-nameNotFound- Team

"JavaChess"

DOCUMENTO DE DISEÑO

Autor(es)
Cáceres, Juan Manuel
Hidalgo, Fabian Nicolás
-
-

Versión de documento
1.0.0

Tabla de contenidos

1 ***HISTORIAL DE REVISIONES***

2 ***PÁGINA DE APROBACIONES***

3 ***INTRODUCCIÓN***
- **3.1** ***Referencias / Acrónimos / Glosario***

4 ***DESCRIPCIÓN GENERAL***
- **4.1** ***Introducción a Patrones de Diseño***
- **4.2** ***Patrones de Diseño Empleados en el proyecto***

5 ***PATRONES DE DISEÑO***
- **5.1** ***Patrón de Diseño Observer***
  - **5.1.1** ***Explicación del Patrón Observer***
  - **5.1.2** ***Implementación del Patrón Observer***
  - **5.1.3** ***Diagrama de Implementación del Patrón Observer***
- **5.2** ***Patrón de Diseño Strategy***
  - **5.2.1** ***Explicación del Patrón Strategy***
  - **5.2.2** ***Implementación del Patrón Strategy***
  - **5.2.3** ***Diagrama de Implementación del Patrón Strategy***
- **5.3** ***Patrón de Diseño Factory***
  - **5.3.1** ***Explicación del Patrón Factory***
  - **5.3.2** ***Implementación del Patrón Factory***
  - **5.3.3** ***Diagrama de Implementación del Patrón Factory***
- **5.4** ***Patrón de Diseño Singleton***
  - **5.4.1** ***Explicación del Patrón Singleton***
  - **5.4.2** ***Implementación del Patrón Singleton***
  - **5.4.3** ***Diagrama de Implementación del Patrón Singleton***

6 ***PRUEBAS UNITARIAS***
- **6.1** ***Qué son las Pruebas Unitarias***
- **6.2** ***Qué Buscamos con Nuestras Pruebas Unitarias***
- **6.3** ***Cómo Correr las Pruebas Unitarias y Verificar su Estado***

1 Historial de revisiones

Versión	Fecha	Observación	Autor(es)
1.0.0	02/06/2022	Versión base del documento	Cáceres, Juan Manuel Hidalgo, Fabian Nicolás

2 Página de Aprobaciones

A continuación se encuentra la información de las personas cuya aprobación es necesaria para realizar cambios significativos al siguiente plan.

* Aprobación sólo será necesaria en caso de cambios mayores, no será necesaria para cambios menores.
* Aprobación sólo será necesaria en caso de que el cambio sea realizado por personas que no son Change Managers.

Configuration Manager	Fecha	Firma
Hidalgo, Fabian Nicolás	02/06/2022	

Configuration Manager BackUp	Fecha	Firma
Cáceres, Juan Manuel	02/06/2022	

Build Manager	Fecha	Firma
Hidalgo, Fabian Nicolás	02/06/2022	

Release Manager - Issue Coordinator	Fecha	Firma
Cáceres, Juan Manuel	02/06/2022	

3 Introducción

En este documento, denominado Documento de Diseño se buscará dar detalles de cómo se desarrolló el software, exponiendo mediante gráficos las partes que lo componen y sus interacciones (tanto internas como entre los diferentes componentes), cuál es la secuencia de operación del sistema dependiendo del caso, entre algunos otros diagramas que se pueden indicar, a su vez también expresaremos qué patrones de diseño se emplearon y su funcionamiento en el código fuente y cómo afecta al sistema como un conjunto, explicando y fundamentando el porqué de su implementación en el proyecto.
Veremos a su vez en otra sección, qué pruebas unitarias se han planificado para ejecutar de manera automática mostrando y exponiendo cuál es el motivo de sus implementaciones, y expresando que funcionalidad buscan controlar en el conjunto de métodos de cada clase.

3.1 Referencias / Acrónimos / Glosario

Acrónimo	Descripción
SCM	Software Configuration Management - Administración de las Configuraciones de Software
CI	Continuous Integration - Integración Continua
CD	Continuous Deployment - Implementación Continua
ID	Identification Code - Código de Identificación
AI	Artificial Intelligence - Inteligencia Artificial
UI	User Interface - Interfaz de Usuario
1vsCOM	Jugador vs Computadora
1vs1	Jugador vs Jugador
SRS	Software Requirements Specifications - Especificación de Requerimientos de Software
MVC	Model-View-Controller - Modelo-Vista-Controlador
UT	Unit Test - Test Unitario
IT	Integration Test - Test de Integración

Tabla 1 - Acrónimos/Glosario

4 Descripción General

4.1 Introducción a Patrones de Diseño

Los patrones de diseño de software son plantillas que sirven para solucionar problemas que surgen habitualmente en la programación o desarrollo de software. Estos diseños han sido probados en diversas situaciones verificando que son una solución óptima y evitando tener que encontrar soluciones a problemas comunes.
Al utilizar patrones de diseño se revelan también durante el desarrollo problemas que no se pueden apreciar a simple vista, permitiendo tomar medidas a tiempo para solucionarlos y que no supongan un problema mucho más grave en fases avanzadas del proyecto.

Existen 3 tipos de patrones de diseño:

1) Los patrones de creación (Creational Patterns) centran su atención en la resolución de los problemas con la interacción del software con los usuarios. Aquí se destacan los siguientes patrones: Abstract Factory, Builder, Factory Method, Prototype y Singleton.
2) Los patrones estructurales (Structural Patterns) centran su atención en la relación de los objetos entre sí, y cómo se agrupan y combinan para formar estructuras más complejas y avanzadas. Aquí se destacan los siguientes patrones: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
3) Los patrones de comportamiento (Behavioral Patterns) centran su atención en la comunicación existente entre los distintos objetos en relación con el algoritmo utilizado, y son los encargados de solucionar problemas referentes al comportamiento del software, simplificando procesos de control. Aquí se destacan los siguientes patrones: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

Las ventajas de aplicar patrones de diseño de software son el ahorro de tiempo y esfuerzo al disponer de una guía o plantilla para afrontar problemas habituales, disminuyen los costes de desarrollo debido al ahorro de tiempo, facilitan la lectura y la comprensión del código fuente del proyecto, la productividad de los programadores se ve incrementada de forma notable por lo que se incrementa la calidad de los programas desarrollados, se cometen menos errores, entre otros.

4.2 Patrones de Diseño Empleados en el Proyecto

Debido a los desafíos que se presentan en la realización del proyecto, hemos intentado pensar de antemano cuales son los patrones de diseño de software más óptimos que se pueden llegar a aplicar al sistema, y se concluyó que son cuatro, siendo estos seleccionados, los más recomendables debido a los problemas que simplifican.
Los patrones de diseño empleados son el patrón Observer, Strategy, Factory y Singleton, donde si bien más adelante se detallarán bien los usos específicos que se le dieron a lo

largo de la realización del proyecto, podemos decir que se presentan problemas recurrentes donde el uso de uno u otro patrón ayudan a solventarlo.
Los problemas más frecuentes son a la hora de las vistas, ya que como se ha descrito en el documento de la arquitectura, se utiliza el patrón MVC, que se basa en desacoplar las vistas, del controlador y de los modelos, por lo que ante un modo multijugador se necesita que un cambio en los modelos del juego, como ser el movimiento de una pieza, un contador de tiempo o el cambio en un menú de selección, se muestra en las vistas de ambos jugadores, por lo que un patrón Observer resulta de mucha utilidad para esto. También utilizamos el patrón factory para la generación de vistas y ciertos componentes debido a su utilidad.
Luego decidimos utilizar el patrón Strategy a la hora de elegir una estrategia en el modo de juego 1vsCOM donde la computadora deberá tomar ciertas decisiones, y estas decisiones tendrán una dificultad asociada, y el usuario debe ser capaz de modificarla antes de empezar una partida o incluso en tiempo de ejecución.
Estas son algunas de las aplicaciones que se le dieron a cada patrón, aunque existen más detalles en la sección siguiente, donde se explicarán los distintos funcionamientos de los métodos y en qué situación se aplican y de qué manera.

5 Patrones de Diseño

A continuación se enlistan y detallan todos los patrones de diseño que fueron empleados en el proceso de diseño y desarrollo del proyecto, además de la explicación de cómo se realiza su implementación en el sistema.

5.1 Patrón de Diseño Observer

5.1.1 Explicación del Patrón Observer

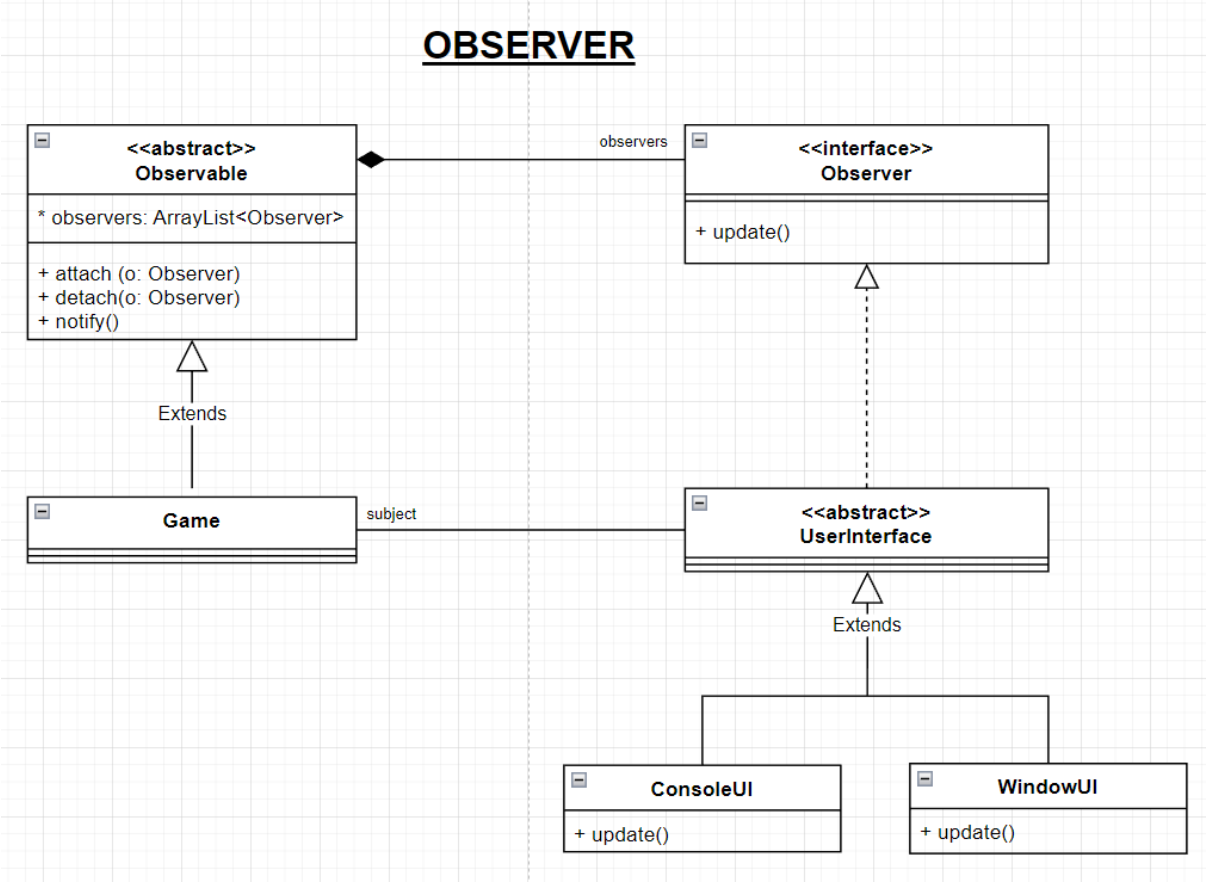
Es un patrón de diseño que define una dependencia “uno entre muchos” entre objetos, de manera que cuando uno de los objetos cambia de estado, notifica este cambio a todos los dependientes. Se trata de un patrón de comportamiento, por lo que está relacionado con algoritmos de funcionamiento y asignación de responsabilidades a clases y objetos.
Su principio de funcionamiento es que el sujeto observado mantiene una lista de las referencias a sus observadores, y los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el sujeto es capaz de notificar a sus observadores suscritos los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado.
Este patrón es muy usado en entornos de interfaces gráficas orientadas a objetos, para capturar y lanzar eventos, y por lo tanto se lleva excelente con el patrón de arquitectura MVC, siendo éste el patrón principal del proyecto.

5.1.2 Implementación del Patrón Observer

Dada la arquitectura que fue seleccionada para el proyecto de nuestro sistema (recordando que es una arquitectura Model-View-Controller) se optó por la aplicación del patrón de diseño Observer en la mayoría de los casos donde es posible su implementación. Ya era sabido que es un patrón de diseño que cuadra muy bien con el patrón de arquitectura MVC por lo que su implementación resulta fácil.
Se usa principalmente en la sección que relaciona las vistas de la aplicación, donde buscamos que al haberse realizado un cambio en el modelo (tablero, piezas, etc.) se visualice el cambio notificando a las múltiples vistas activas en el momento, pudiendo ser por ejemplo, el menú principal, el tablero de juego, el scoreboard, etc. Desacoplando así las vistas del modelo y pudiendo elegir el formato de representación deseado para el juego sin necesidad de tener un código fuente complejo que deba reestructurar su funcionamiento si se realiza un cambio en los modelos que se encuentran en visualización.
Además el patrón observer tiene el beneficio de poder actualizar la información presentada al usuario sin la necesidad de enviar información a la vista, la cual debe ser traducida e interpretada, sino que como hemos dicho ante cualquier modificación a los modelos estas simplemente se actualiza, lo cual es más simple y eficiente.
Si bien hemos aplicado manualmente el patrón de diseño con clases propias, es sabido que el lenguaje Java de por sí utiliza e implementa el patrón observer en la construcción de

ventanas, para lo que sería capturar y actuar frente a eventos tales como hacer clic en alguna zona predeterminada, presionar un botón, deslizar una barra, etc.
Por último, decidimos implementar como observable no al tablero en sí sino al juego ya que cualquier modificación al modelo ocurrirá ante la llamada de este método, y como los métodos que modifican a este dentro de su propia clase son varios, decidimos que sería más simple informar la necesidad de actualizar a los observers desde la clase game.

5.1.3 Diagrama de Implementación del Patrón Observer



5.2 Patrón de Diseño Strategy

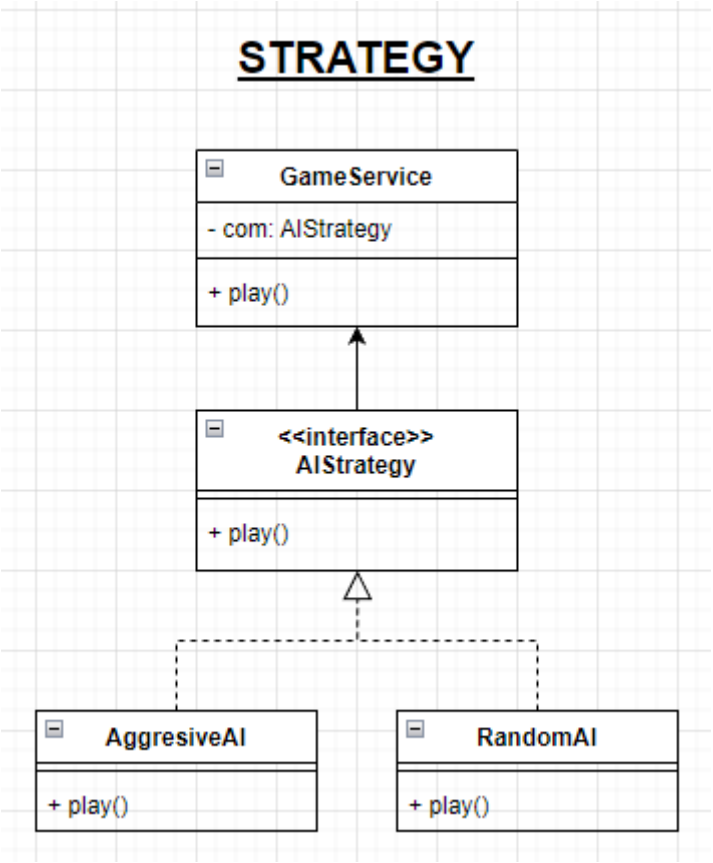
5.2.1 Explicación del Patrón Strategy

Es un patrón de diseño que se clasifica como patrón de comportamiento porque determina cómo se debe realizar determinada tarea, y permite así mantener un conjunto de algoritmos de entre los cuales el objeto cliente pueda elegir aquel que le conviene e intercambiarlo dinámicamente según las necesidades presentes en un momento determinado.
Su aplicación es en todo programa que ofrezca un servicio o función determinada que pueda ser realizada de varias maneras, donde puede haber una cantidad cualquiera de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución.

5.2.2 Implementación del Patrón Strategy

Dado que el proyecto contará con un apartado para iniciar una partida de 1 solo jugador, resulta necesaria la creación de un algoritmo controlado por la computadora capaz de arrojar movimientos bajo determinadas condiciones de juego, y por lo tanto es prudente crear una clase especial que se encargue de gestionar estos algoritmos.
Para ello utilizamos la implementación del patrón de diseño Strategy, donde dada las características del patrón, tendremos una clase Strategy y subclases que extenderán a la misma, donde éstas serán las estrategias o algoritmos preconfigurados pensados para actuar en las condiciones de juego de un jugador, realizando movimientos según cada estrategia puntual que el usuario elija.
El uso de este patrón de diseño nos permite cambios de estrategia antes de iniciar una partida, como también en medio del tiempo de ejecución, además de que brinda modularidad a la hora del desarrollo del código, teniendo clases separadas controladas por una clase principal que las controla.
Para su implementación se pensó en la realización de solo 2 estrategias posibles, una estrategia “0” que se base en la realización de movimientos completamente aleatorios de las piezas en el tablero (cumpliendo con las normas propias del juego ajedrez) y otra estrategia “1” donde ya exista un algoritmo más complejo, pensado para que se realicen movimientos más agresivos, que priorice el atacar y tomar piezas rivales dependiendo de la cercanía de sus piezas con las del rival.
La idea de este patrón es que el usuario pueda acceder al menú en el apartado de configuración y cambiar la estrategia, tanto antes de empezar una partida, como en el medio de una, aplicando las ventajas que ofrece el patrón de diseño Strategy.
El beneficio que tiene este patrón es la posibilidad de poder agregar más estrategias en un futuro sin necesitar una reestructuración del código fuente importante, tan solo con la adición de una nueva clase que herede de AIStrategy sería suficiente.

5.2.3 Diagrama de Implementación del Patrón Strategy



5.3 Patrón de Diseño Factory

5.3.1 Explicación del Patrón Factory

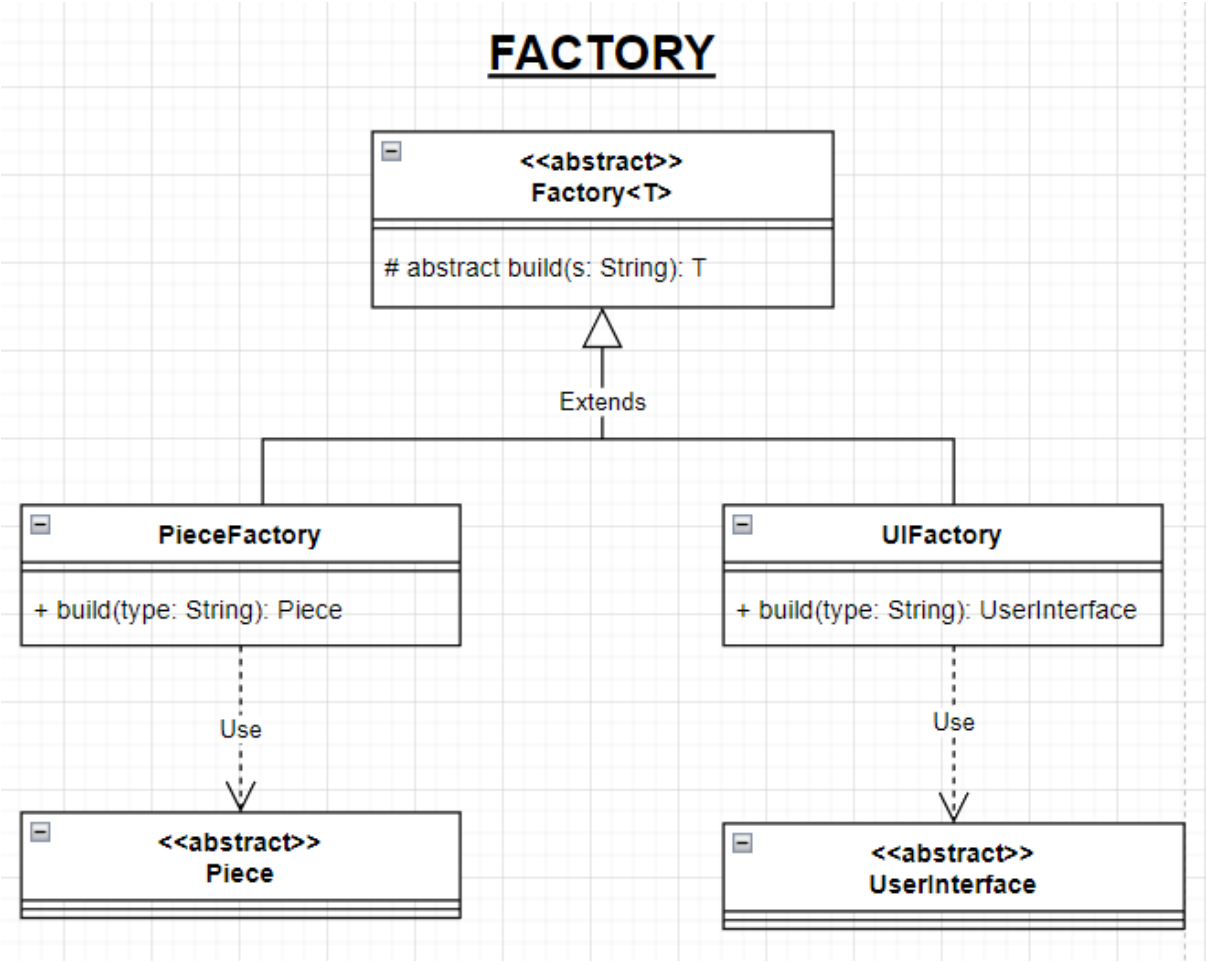
Es un patrón de diseño que consiste en utilizar una clase constructora abstracta con unos cuantos métodos definidos y otros abstractos, que son los dedicados a la construcción de objetos de un subtipo determinado, y es así entonces qué podemos decir que describe un enfoque de programación que sirve para crear objetos sin tener que especificar su clase exacta, por lo tanto el objeto creado puede intercambiarse con flexibilidad y facilidad por otro.
Las clases principales en este patrón son el creador y el producto, donde el creador necesita crear instancias de productos, pero los tipos concretos de cada producto no deben estar reflejadas en el propio creador sino que las posibles subclases del creador deben poder especificar los tipos concretos, subclases, de los productos para utilizar.
Su uso puede especificarse en una interfaz o simplemente mediante la clase hijo o la clase base y opcionalmente sobreescribirse (mediante clases derivadas). Al hacerlo, el patrón o método toma el lugar del constructor de la clase normal para separar la creación de objetos de los propios objetos.

5.3.2 Implementación del Patrón Factory

Este patrón de diseño se implementa en casi todas las etapas del programa que son necesarias para instanciar piezas, el tablero al inicio de una partida e incluso para generar las distintas vistas que posee la aplicación en su tiempo de ejecución.

Al respecto de las piezas, como el proyecto poseen diferentes tipos de éstas, cada una con sus atributos propios, es posible aplicar el patrón de diseño Factory para aplicar polimorfismo, donde la clase Factory es la encargada de mantener la lógica encargada de la construcción de los distintos objetos “Pieza” según el algoritmo lo tenga indicado en cada subclase. Esto simplifica la lectura del código, ya que con un simple array podemos indicarle qué piezas generar en lugar de tener que agregar un gran número de líneas para crear muchos objetos de distintos tipos.
Con respecto a las vistas ya que le otorgamos la posibilidad al usuario de que decida verlo por consola o vistas de java, decidimos implementar un factory que recibe el input del usuario y genera la vista deseada.

5.3.3 Diagrama de Implementación del Patrón Factory



5.4 Patrón de Diseño Singleton

5.4.1 Explicación del Patrón Singleton

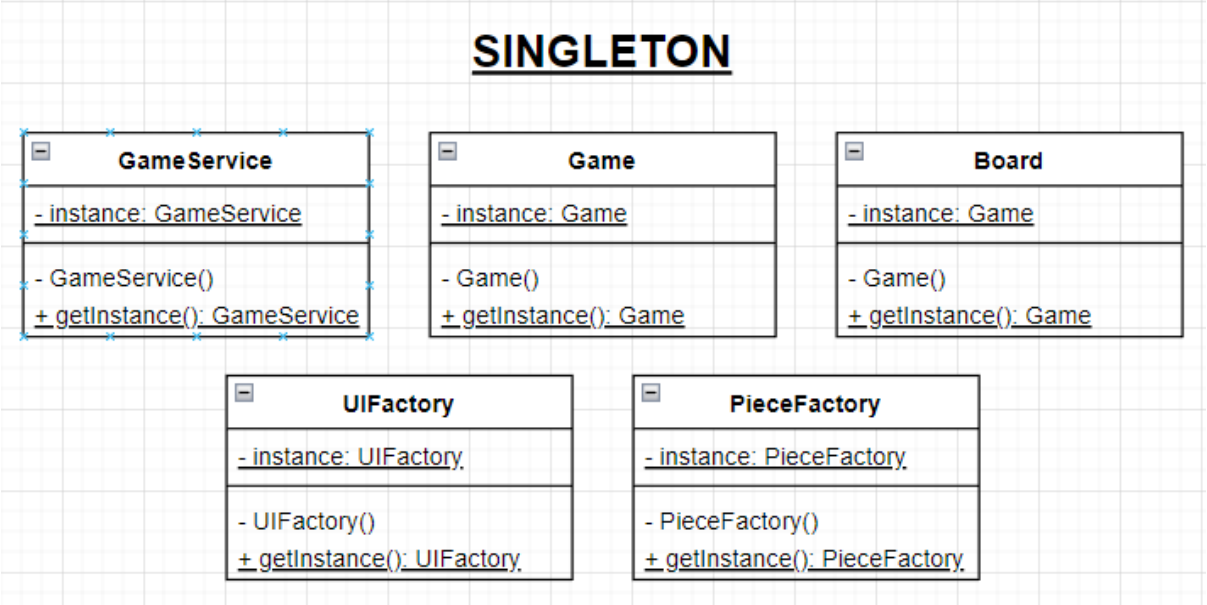
Es un patrón de diseño que su función es permitir restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

Este patrón se implementa creando en nuestra clase un método que crea una instancia del objeto solo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor con modificadores de acceso como protegido o privado.
Este patrón de diseño puede ser delicada en programas con múltiples hilos de ejecución, donde si dos hilos intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar la exclusión mutua en el método de creación de la clase que implementa el método.
Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único, como puede ser un ratón o un archivo abierto en modo exclusivo, o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

5.4.2 Implementación del Patrón Singleton

Este patrón de diseño se encuentra implementado en el proyecto en todo lo que tenga que ver en cuanto a cantidad de objetos, limitando las cantidades que pueden llegar a afectar el desarrollo normal del programa. Es por esto que se utiliza en la construcción del tablero y las vistas.
Al utilizar el patrón Singleton en las piezas, hacemos que no se puedan duplicar al realizar un movimiento, o que al ser reemplazadas por otra, tampoco se dupliquen, como también en el tablero por ejemplo, para que en el modo de 2 jugadores, ambos se encuentren visualizando el mismo tablero y no 2 diferentes, lo que podría llegar a crear problemas, ya que en ese caso tendríamos 2 modelos de tablero y posiblemente uno de ellos no reciba ningún tipo de cambio en las posiciones de sus piezas, o también podría parecer como que cada jugador está jugando en un tablero individual propio y eso podría ocasionar que la partida nunca avance como corresponde.
A su vez la implementación más importante, directa y visible del patrón de diseño Singleton es en las vistas, ya que aquí se ve constantemente actuando para, por ejemplo, evitar tener múltiples partidas en curso o llegar a repetir el proceso de creación de partida, si es que tocamos repetidas veces el botón de “iniciar nueva partida” donde lo que hará el patrón de diseño Singleton será abrir o focalizar en la pantalla la instancia de la vista que se encuentre activa en el momento o en caso de no haber ninguna, instanciar una nueva.

5.1.3 Diagrama de Implementación del Patrón Singleton


SINGLETON
GameService
- instance: GameService
- GameService()
+ getInstance(): GameService
Game
- instance: Game
- Game()
+ getInstance(): Game
Board
- instance: Game
- Game()
+ getInstance(): Game
UIFactory
- instance: UIFactory
- UIFactory()
+ getInstance(): UIFactory
PieceFactory
- instance: PieceFactory
- PieceFactory()
+ getInstance(): PieceFactory

6 Pruebas Unitarias

6.1 Qué son las Pruebas Unitarias

Las pruebas unitarias o Unit Test son un método de pruebas de software que se realizan escribiendo fragmentos de código para testear determinadas unidades de código fuente, donde su objetivo es asegurar que cada unidad funciona como debería de forma independiente.
Pueden ser clases o métodos específicos dentro del código que se encargan de realizar los tests y son las primeras pruebas que se deben realizar al proyecto, donde estas son creadas por el mismo desarrollador o por un grupo aparte especializado en la realización de tests.
Los principios que suelen respetar las pruebas unitarias son que:

1) Deben completar su ejecución lo más rápido posible.
2) El objeto de prueba no debe depender de otra unidad de código.
3) Estas pruebas deben poder ser repetibles.
4) Deben mostrar de forma clara el resultado de la prueba.
5) Las pruebas deben realizarse lo antes posible, incluso antes de que haya código escrito de ser posible.

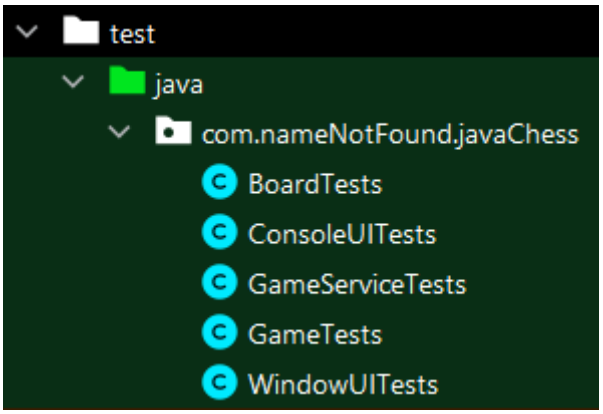
6.2 Qué Buscamos con Nuestras Pruebas Unitarias

Con la realización de algunas de nuestras pruebas unitarias buscamos de forma independiente que los fragmentos de código fuente que se han implementado o que se van a implementar cumplan con los resultados que se esperan.
Existen métodos del proyecto que en caso de no responder de forma debida pueden llegar efectuar problemas en el sistema, haciendo por ejemplo que se permitan realizar movimientos indebidos o que el programa colapse y se deba reiniciar la aplicación debido a un fallo por ejemplo cuando se ingresa una posición inexistente en una matriz y se lanza una excepción que no fue capturada, o permitiendo que el usuario ingrese cualquier cosa como posición solicitada y el método encargado de validar la entrada falle, causando que esa posición a medida que se la empieza a analizar a lo largo del algoritmo, comience a causar distorsiones en el desarrollo del juego.
Los módulos más críticos para someter a pruebas unitarias son aquellos relacionados a las interfaces de usuario, para evitar ingresos de datos erróneos, y luego la parte de la lógica del juego, o sea la clase Game y GameService, siendo estas las encargadas del funcionamiento principal del proyecto, o sea, el poder jugar ajedrez de manera correspondiente.

6.3 Cómo Correr las Pruebas Unitarias y Verificar su Estado

Gracias a la utilización de Maven, contamos con una estructura de carpetas donde las pruebas se encuentran separadas en su propio paquete. Con la ejecución de maven test podemos ejecutar todas las pruebas contenidas en este paquete.

Esto nos permite ejecutarlas dentro de un pipeline en github y asegurarnos que todo push hacia el repositorio remoto esté libre de errores significativos y que cumpla no solo con las funcionalidades que está añadiendo sino con todas las funcionalidades ya implementadas. Es decir que modificaciones en el código fuente no hayan comprometido la funcionalidad del proyecto ya establecida.
El framework que utilizamos para testear es JUnit, el cual nos permite ejecutar instrucciones 'assert' que permiten controlar qué objetos sean de cierto tipo o estado.
La ejecución en el ordenador personal se realiza ejecutando el comando maven test desde la pestaña de lifecycle de maven en el entorno de desarrollo o ejecutando el comando en la consola 'mvn test'. Este mismo comando es el que se ejecuta en el pipeline, el cual bloquea el push al repositorio remoto ante fallas de los tests.



6.3.1 Piece Tests:

BishopCorrectMovements(): Verifica que estén cargadas las posiciones correctas de movimiento.

BishopCorrectTakes(): Verifica que estén cargadas las posiciones correctas donde puede tomar la pieza.

KingCorrectMovements(): Verifica que estén cargadas las posiciones correctas de movimiento.

KingCorrectTakes(): Verifica que estén cargadas las posiciones correctas donde puede tomar la pieza.

KnightCorrectMovements(): Verifica que estén cargadas las posiciones correctas de movimiento.

KnightCorrectTakes(): Verifica que estén cargadas las posiciones correctas donde puede tomar la pieza.

PawnCorrectMovements(): Verifica que estén cargadas las posiciones correctas de movimiento.

PawnCorrectTakes(): Verifica que estén cargadas las posiciones correctas donde puede tomar la pieza.

QueenCorrectMovements(): Verifica que estén cargadas las posiciones correctas de movimiento.

QueenCorrectTakes(): Verifica que estén cargadas las posiciones correctas donde puede tomar la pieza.

RookCorrectMovements(): Verifica que estén cargadas las posiciones correctas de movimiento.

RookCorrectTakes(): Verifica que estén cargadas las posiciones correctas donde puede tomar la pieza.

6.3.1 Board Tests:

OnlyOneBoard(): Comprobamos que solo exista un tablero.

BoardMovePiece(): Comprobamos que pueda moverse una pieza en el tablero.

CanFindKing(): Comprueba que el programa pueda encontrar el rey de ambos colores en el tablero.

CanUndoMovement(): Comprueba que el programa pueda deshacer un movimiento en el tablero.

6.3.2 Game Tests:

OnlyOneGame(): Comprueba que solo exista una instancia de juego

BoardInitializeCorrect(): Comprobamos que el tablero se inicializa con las piezas correctas en la posición correcta.

TurnChange(): Comprueba que pueda cambiarse el turno.

DetectIsCheck(): Comprueba que el programa detecte cuando se produce Jaque.

DetectIsCheckmate(): Comprueba que el programa detecte cuando se produce Jaquemate.

DetectisCastle(): Comprueba que el programa pueda determinar si la acción ingresada es enroque.

DetectCastle(): Comprueba que se produzca un enroque ante las inputs correctas.

DetectPieceMovementBlocked(): Comprueba que el programa pueda identificar que el movimiento legal de una pieza está siendo bloqueado por otra y le impide al usuario realizar esta acción.

DetectPieceIsAttacked(): Comprueba que el programa pueda identificar que una pieza está siendo atacada por alguna del contrincante.

6.3.3 GameService Tests:

OnlyOneGameService(): Comprueba que solo exista una instancia de GameService.

DetectEnemyPieceMoveAttempt(): Comprobamos que el programa impida al usuario mover una pieza enemiga.

DetectEatingOwnPiece(): Comprobamos que el programa impide al usuario comer una de sus propias piezas.

DetectNoPieceSelected(): Comprobamos que el programa forze al usuario a seleccionar una pieza.

DetectInvalidMovement(): Comprobamos que el programa compruebe que la pieza pueda moverse a esa posición.

DetectInvalidTake(): Comprobamos que el programa pueda detectar si la pieza seleccionada puede tomar la pieza enemiga de acuerdo a su patrón de movimientos.

DetectIsCastling(): Comprobamos que el programa detecte un intento incorrecto de enroque por el usuario y le informe al mismo del procedimiento correcto.

6.3.4 ConsoleUI Tests:

IsValidInput(): Comprobamos que el programa solo acepta inputs válidas del usuario por consola

DidUpdate(): Comprueba que ante la modificación del tablero se actualice la vista.

7 Diagramas

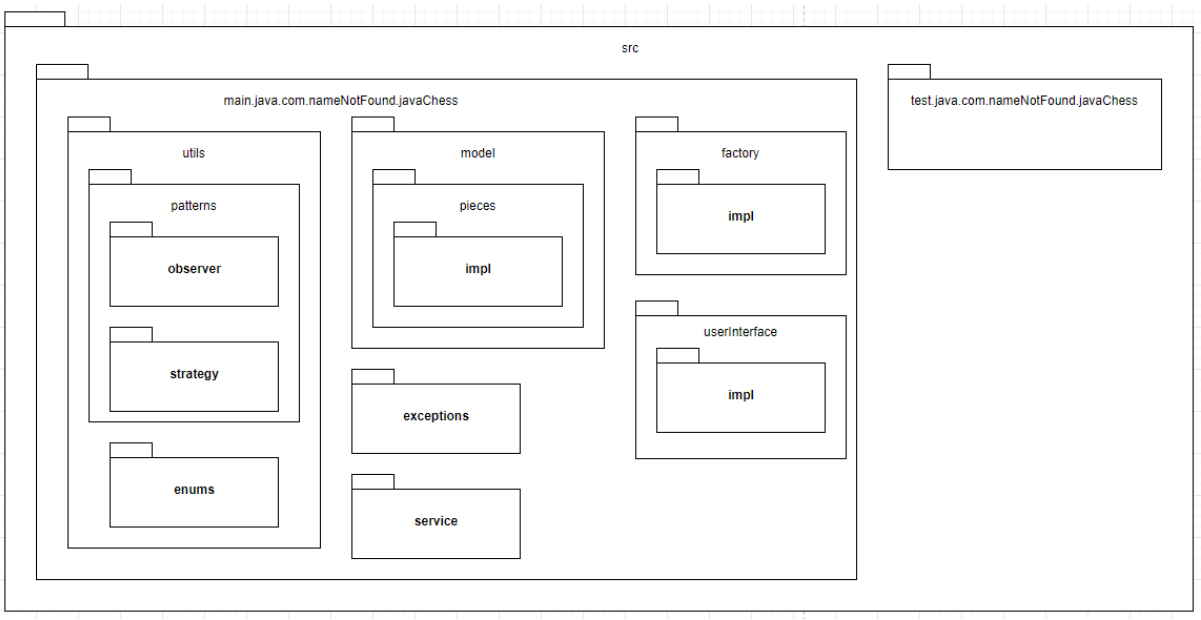
7.1 Diagrama de Clases

Debido a la extensión del diagrama de clases se incluye un link al archivo y a su vez el mismo está en el zip de los documentos.

https://drive.google.com/file/d/1d3L8jiO9miUALd7jJ6-1i6msBPgDxWdw/view?usp=sharing

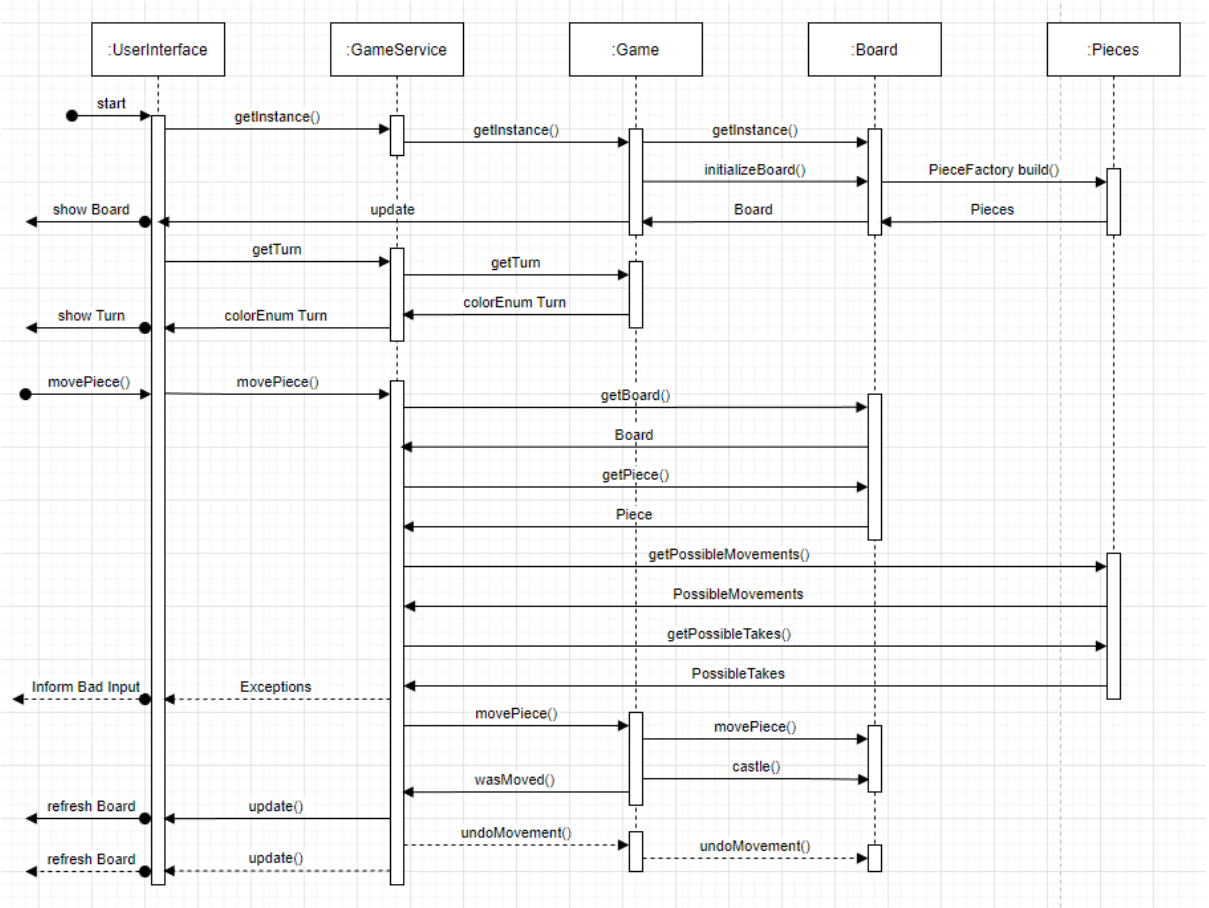
7.2 Diagrama de Paquetes

Utilizamos una estructura de paquetes estandarizada de acuerdo a la estructura usual de maven y spring.



7.2 Diagrama de Secuencia

Muestra la comunicación entre las distintas clases del sistema ante la interacción del usuario.


:UserInterface
:GameService
:Game
:Board
:Pieces
start
getInstance()
getInstance()
getInstance()
initializeBoard()
PieceFactory build()
show Board
update
Board
Pieces
getTurn
getTurn
colorEnum Turn
show Turn
colorEnum Turn
movePiece()
movePiece()
getBoard()
Board
getPiece()
Piece
getPossibleMovements()
PossibleMovements
getPossibleTakes()
PossibleTakes
Inform Bad Input
Exceptions
movePiece()
movePiece()
castle()
wasMoved()
refresh Board
update()
undoMovement()
undoMovement()
refresh Board
update()