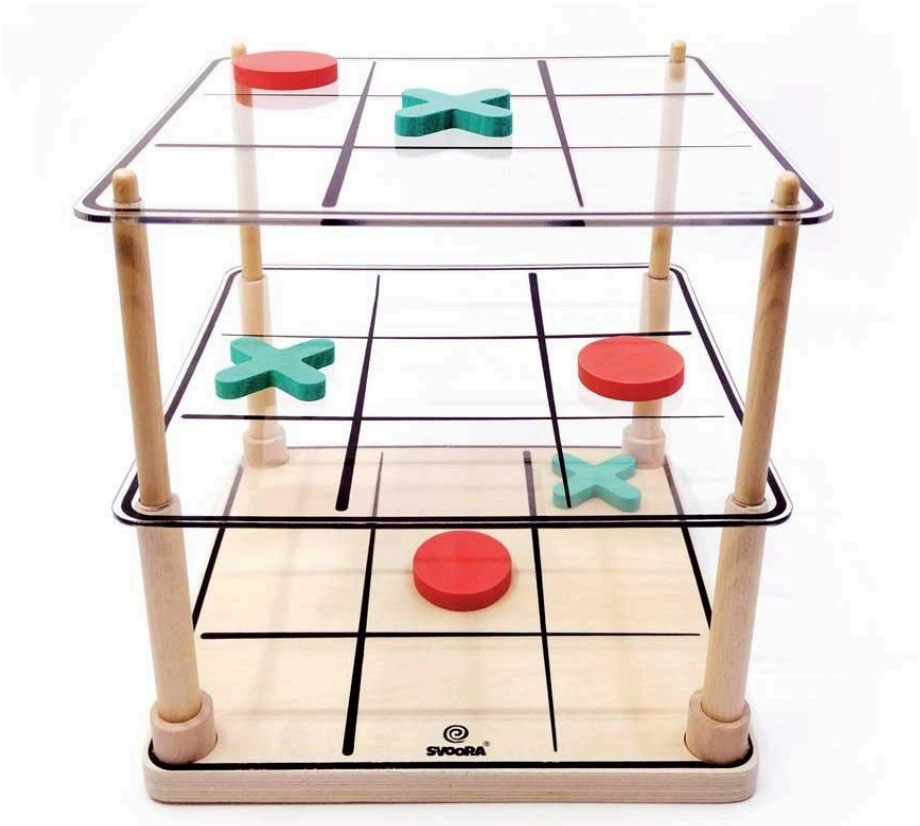
Algoritmos y Estructuras de datos - CB100

# TRABAJO PRÁCTICO N°2
# TATETI V2.0




Universidad de Buenos Aires

Facultad De Ingeniería

Año 2024- 2do Cuatrimestre

**PSEUDOEQUIPO**

Integrantes:

Axel Carlos Lopez - Santiago Cacace - Ulises Distefano - Leandro Julian Naveiro - Luz Marina Martinez Acuña.

# Índice:

# 1 - Objetivos

Generar una pieza de software que simule el funcionamiento del juego Tateti en su versión multi Jugador.

## Enunciado:

Tateti es un juego de mesa para varios jugadores en el que se introducen X fichas en un tablero de N por M por Z con el fin de alinear las fichas en una línea recta, ya sea horizontal, vertical o diagonal. Por cada turno cada jugador ingresa una ficha y luego de tener todas las fichas en el tablero se pueden mover en el sentido horizontal o vertical o profundo. Cada jugador podrá tener hasta C cartas en su mano, pudiendo jugarlas después de su turno. Las cartas se levantan al principio de cada turno de un mazo (cola de cartas). Los tipo de cartas disponibles son:

a) Hacer perder un turno a un jugador

b) Bloquear una ficha de otro jugador

c) Anular un casillero del tablero

d) Volver atrás una jugada del turno.

e) Cambiar de color una ficha.

El grupo debe añadir 2 tipos de cartas más, inventando su funcionalidad.

Al iniciar cada turno, se tira un dado que indica la cantidad de cartas a sacar, si se sobrepasa la cantidad que se pueden tener en mano, se devuelven al mazo.

La estructura principal debe estar hecha con listas. Interfaz de usuario Toda la interfaz de usuario debe estar basada en texto. El estado del tablero tiene que mostrarse usando un archivo de imagen BMP. Después de cada turno, el programa debe exportar el tablero en un archivo bitmap con el estado del tablero, de manera que quede una secuencia de imágenes.

No es necesario que se limpie la pantalla, simplemente escribir el estado del tablero luego de cada jugada.

## Responder el siguiente Cuestionario:

1) ¿Qué es un svn?

2) ¿Que es una “Ruta absoluta” o una “Ruta relativa”?

3) ¿Qué es git?

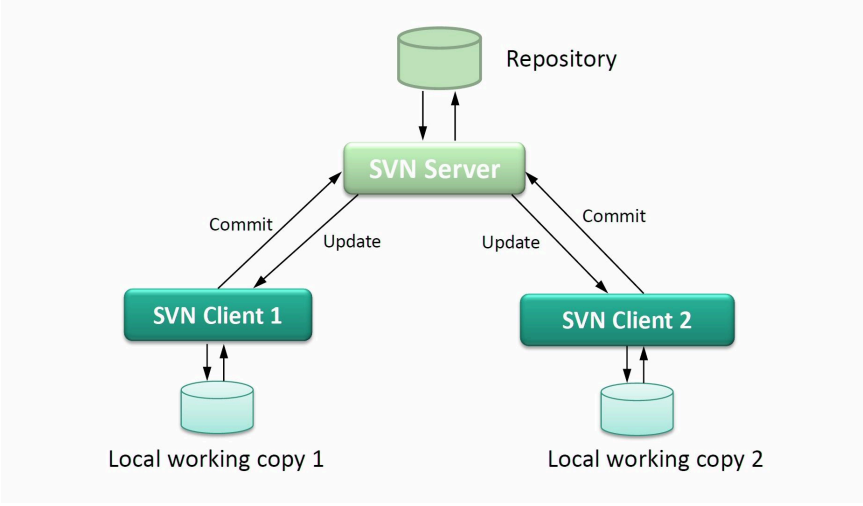
# 2 - Cuestionario

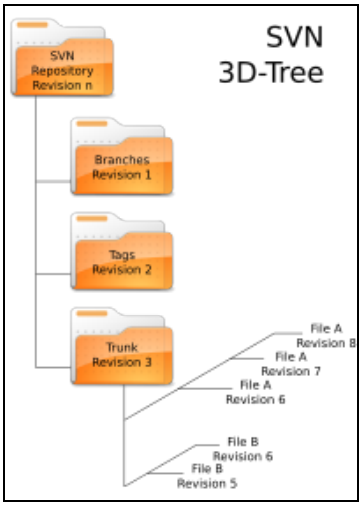
## 2.1 - ¿Qué es un SVN?

Apache Subversion (abreviado como SVN, por el comando svn) es una herramienta de control de versiones open source basada en un repositorio cuyo funcionamiento se asemeja enormemente al de un sistema de ficheros.

Utiliza el concepto de revisión para guardar los cambios producidos en el repositorio. Entre dos revisiones solo guarda el conjunto de modificaciones, optimizando así al máximo el uso de espacio en disco. SVN permite al usuario crear, copiar y borrar carpetas con la misma flexibilidad con la que lo haría si estuviese en su disco duro local. Dada su flexibilidad, es necesaria la aplicación de buenas prácticas para llevar a cabo una correcta gestión de las versiones del software generado.

SVN puede acceder al repositorio a través de redes, lo que le permite ser usado por personas que se encuentran en distintas computadoras. A cierto nivel, la posibilidad de que varias personas puedan modificar y administrar el mismo conjunto de datos desde sus respectivas ubicaciones fomenta la colaboración. Se puede progresar más rápidamente sin un único conducto por el cual deban pasar todas las modificaciones. Y puesto que el trabajo se encuentra bajo el control de versiones, no hay razón para temer porque la calidad del mismo vaya a verse afectada.



## Ventajas:

- Se sigue la historia de los archivos y directorios a través de copias y documentos renombrados.
- Centralización: SVN utiliza un repositorio centralizado, lo que facilita el control y la gestión del código. Todos los desarrolladores trabajan sobre una versión única del repositorio, lo que simplifica la administración.
- Facilidad de configuración: Configurar y gestionar un servidor SVN es relativamente sencillo. Es una opción atractiva para equipos pequeños o medianos que no necesitan la complejidad de un sistema distribuido como Git.
- Control preciso de versiones: Subversion permite un control detallado de versiones. Puedes hacer "checkouts" y "updates" de versiones específicas o de ramas (branches) sin necesidad de descargar todo el historial de la base de código, lo que facilita su uso con proyectos grandes.
- Soporte de grandes archivos binarios:
- A diferencia de Git, SVN maneja mejor los archivos binarios grandes. Esto lo hace útil en proyectos que trabajan con medios como imágenes, audio, o video, donde los cambios son menos frecuentes y más grandes.
- Estructura de ramas clara: Subversion tiene una convención de estructura de carpetas de repositorios que facilita el trabajo con ramas y etiquetas. Normalmente, las carpetas de "trunk" (rama principal), "branches" (ramas) y "tags" (etiquetas) están bien definidas.
- Control de acceso más fácil: Como es un sistema centralizado, los permisos de acceso pueden ser gestionados de manera sencilla en el servidor, asegurando que solo ciertos usuarios puedan acceder o modificar determinadas partes del proyecto.
- Menos conflictos de fusión: Debido a su enfoque centralizado, los conflictos de fusión (merge conflicts) son menos comunes en comparación con sistemas distribuidos, como Git. Sin embargo, esto también depende del flujo de trabajo utilizado.
- Permite selectivamente el bloqueo de archivos. Se usa en archivos binarios que, al no poder fusionarse fácilmente, conviene que no sean editados por más de una persona a la vez.

## Desventajas:

- El manejo de cambio de nombres de archivos no es completo. Lo maneja como la suma de una operación de copia y una de borrado.
- No resuelve el problema de aplicar repetidamente parches entre ramas, no facilita llevar la cuenta de qué cambios se han realizado. Esto se resuelve siendo cuidadoso con los mensajes de commit.

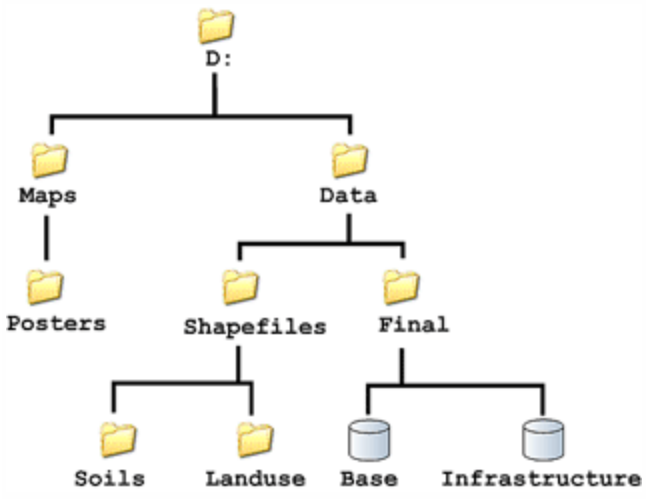
- Centralización y dependencia del servidor: Al ser un sistema centralizado, SVN depende completamente de un servidor. Si el servidor cae o se pierde, se pierde el acceso a todo el historial del proyecto.
- Falta de flexibilidad en el trabajo offline: A diferencia de Git, que permite a los desarrolladores trabajar de manera completamente offline y realizar "commits" locales, en SVN todo el trabajo depende de la conexión al servidor para hacer commit o actualización de archivos.
- Escalabilidad limitada: SVN tiene limitaciones a la hora de escalar en proyectos muy grandes o con equipos distribuidos.

## 2.2 - ¿Qué es una “Ruta absoluta” o una “Ruta relativa”?

Una ruta es simplemente el camino que se sigue para llegar a un archivo o directorio dentro del sistema de archivos de un sistema operativo. Es la dirección que se proporciona para acceder a un archivo o carpeta en el sistema.

Una ruta absoluta es aquella que especifica el camino completo desde el directorio raíz del sistema de archivos, sin importar el directorio en el que te encuentres. Te da la dirección completa desde el inicio del sistema hasta el archivo o directorio deseado. No depende de la ubicación actual del usuario. Si alguien te da una ruta absoluta, puedes llegar al destino sin importar tu ubicación actual.

Una ruta relativa es aquella cuyo camino para llegar a un archivo o directorio depende del directorio activo en el que te encuentras en ese momento. Osea que la ruta se calcula en relación al directorio actual.




Imaginemos que estamos en Linux y que nuestro directorio de trabajo actual es /home/Pseudoequipo, si queremos acceder al archivo localizado en /home/Pseudoequipo/tp2/informe.pdf podemos hacerlo de dos formas:

Usando la ruta absoluta: /home/Pseudoequipo/tp2/informe.pdf

Usando la ruta relativa: tp2/informe.pdf

Las rutas relativas no tienen un separador al inicio de la ruta.

## 2.3 - ¿Qué es Git?

Git es un software de control de versiones pensando en la eficiencia, la confiabilidad y compatibilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora incluyendo coordinar el trabajo que varias personas realizan sobre archivos compartidos en un repositorio de código.

Al principio, Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario o front end. Sin embargo, Git se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena. Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux.

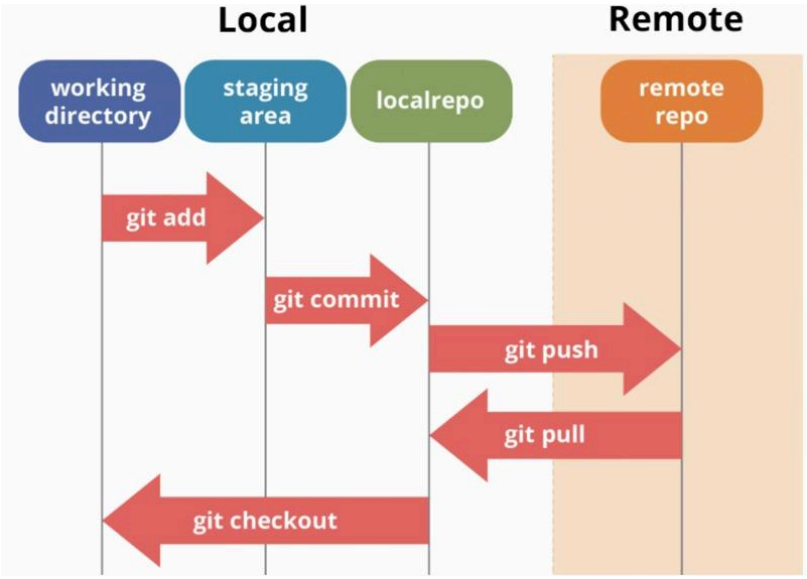
**Ventajas**:

- Distribuido y rápido, permitiendo trabajo offline y una alta eficiencia.
- Gestión de ramas flexible para trabajar en paralelo.
- Historial completo que permite una gran visibilidad de los cambios.
- Ideal para equipos grandes.

**Desventajas**:

- Curva de aprendizaje pronunciada para principiantes.
- No es ideal para archivos grandes o repositorios con muchos archivos binarios.
- Conflictos de fusión pueden ser problemáticos en equipos grandes.
- Puede volverse lento con repositorios muy grandes o con un largo historial.

A pesar de sus desventajas, Git sigue siendo una de las mejores herramientas de control de versiones disponibles debido a su flexibilidad, potencia y adopción generalizada en la comunidad de desarrollo de software.




## 3 - Manual del usuario

El juego consiste en un ta-te-ti de tres dimensiones.

Comienzo eligiendo el ancho, el alto y la profundidad del tablero con el que se va a jugar. Luego se elige la cantidad de jugadores para la partida, a tener en cuenta: no hay una cantidad máxima de jugadores por lo tanto elegir adecuadamente las proporciones del tablero y los jugadores.

Cada Jugador elige su nombre, número que representa un color en particular(1 al 7).

El mazo se genera automáticamente al inicio del juego y consta de 50 cartas con una cantidad aleatoria de cada tipo de carta con una jugada diferente cada una.

Paso a detallar las mismas:

- Carta anular casillero:  Carta que anula un casillero del tablero.
- Carta bloquear ficha: Si se aplica esta carta se bloquea la ficha elegida y la misma no se puede cambiar de su lugar
- Carta cambiar color de ficha: Al utilizar esta carta cambia el color de una ficha a elección por el color del jugador que usó la carta
- Carta doble turno: Al usar la misma, el jugador tiene dos turnos seguidos
- Cartas eliminar cartas del jugador: Elimina las cartas en la mano del jugador que se elija
- Cartas pierde turno: El jugador elige a algún otro jugador del juego para perder un turno
- Carta volver jugada anterior: Carta que retrocede el tiempo una jugada atrás.

Al terminar el ingreso de los jugadores y la creación del mazo y el tablero, el juego comienza.

El primer jugador es el primer usuario que agregó sus datos, así sucesivamente.

Los turno corresponden:

1- Tiramos el dado, dependiendo del número del mismo se levantan ese número de cartas

2- Luego el jugador decide si mover o agregar una ficha al tablero, si corresponde.

3- El jugador decide jugar una carta (las mismas están explicadas arriba)

4- Dependiendo de la carta, se decide la jugada

Estos 4 pasos se repiten hasta que hay un ganador, el mismo se decide cuando hay 3 fichas en línea, ya sean horizontalmente, verticalmente o por profundidad.

# 4 - Manual del programador

**TDAs Estructuras:**

Nodo: Clase Nodo genérica que se utiliza para representar un nodo de una estructura de datos enlazada, como una lista enlazada, una pila o una cola. Cada nodo almacena un dato y un enlace al siguiente nodo. Contiene método para setear y obtener el dato y su siguiente.

Vector: Define un TDA Vector genérico que implementa una estructura de datos similar a un arreglo dinámico. Permite almacenar elementos en un vector de longitud variable, con métodos para agregar, obtener, y remover elementos.

Lista: Lista simplemente enlazada donde cada nodo contiene un dato. Contiene un cursor que permite agregar, eliminar, y acceder a los nodos de la lista.

Cola: Genera una Cola genérica, con métodos como acolar, desacolar, obtener el dato en el nodo, comprobar si está vacía y contar la cantidad de elementos.

Pila: Genera una Pila genérica, con métodos como apilar, desapilar, obtener el dato en el nodo, comprobar si está vacía y contar la cantidad de elementos.

TemplateDeClase: TDA con todos los templates utilizados en las otras clases del paquete.

**TDAs Cartas:**

Carta: Clase de Carta abstracta hecha para ser extendida por las cartas de cada tipo. Contiene un id único para cada carta que se genera de forma automática junto con un título que es específico de cada tipo de carta.

CartaAnularCasillero: Carta hija de Carta abstracta, que sirve para anular casillero de otro jugador. Al obtener su jugada llama a JugadaAnularCasillero.

CartaBloquearFicha: Carta hija de Carta abstracta, que sirve para bloquear la ficha de otro jugador. Al obtener su jugada llama a JugadaBloquearFicha.

CartaCambiarColorFicha: Carta hija de Carta abstracta, que sirve para cambiar el color de la ficha de otro jugador por el del jugador que esté jugando. Al obtener su jugada llama a JugadaCambiarColorFicha.

CartaDobleTurno: Carta hija de Carta abstracta, que sirve para jugar un turno extra. Al obtener su jugada llama a JugadaDobleTurno.

CartaEliminarCartasDelJugador: Carta hija de Carta abstracta, que sirve para eliminar las cartas del mazo de otro jugador. Al obtener su jugada llama a JugadaEliminarCartasDelJugador.

CartaPierdeTurno: Carta hija de Carta abstracta, que sirve para hacer perder un turno a otro jugador. Al obtener su jugada llama a JugadaEliminarCartasDelJugador.

CartaVolverJugadaAnterior: completar (no está hecha)

**TDAs Jugadas:**

Jugada: Clase de jugada abstracta, hecha para ser extendida por jugadas de un tipo en concreto. Contiene un metodo para jugar la jugada pasando por parámetro la partida y el turno actual.l

JugadaAnularCasillero: Jugada hija de Jugada abstracta, que sirve para efectuar el efecto de CartaAnularCasillero.

JugadaBloquearFicha: Jugada hija de Jugada abstracta, que sirve para efectuar el efecto de CartaBloquearFicha.

JugabaCambiarColorFicha: Jugada hija de Jugada abstracta, que sirve para efectuar el efecto de CartaCambiarColorFicha.

jugadaDobleTurno: Jugada hija de Jugada abstracta, que sirve para efectuar el efecto de CartaDobleTurno.

JugadaEliminarCartasDelJugador:Jugada hija de Jugada abstracta, que sirve para efectuar el efecto de CartaEliminarCartasDelJugador.

JugadaPierdeTurno: Jugada hija de Jugada abstracta, que sirve para efectuar el efecto de CartaPierdeTurno.

JugadaVolverJugadaAnterior: Jugada hija de Jugada abstracta, que sirve para efectuar el efecto de CartaVolverJugadaAnterior.

**TDAs del Main:**

Main: Main principal donde se llama a los otros TDA del proyecto y se realiza el juego en sí. Esta es la clase que hay que correr para poder jugar al juego.

Bitmap: TDA que contiene todas las funciones para la generación de imágenes asociadas al juego.

*Métodos:*

*Void Inicializar (int, int, int):* inicializa el bitmap, con el alto, ancho, x, y, z, y las dimensiones (todas tienen que ser mayores a 1)

*Void CrearImagen():* crea la imagen de los tableros del juego, y la exporta al archivo de salida.

*Void colorMatrizCentral():* colorea la matriz central.

*Void escribirImagen():* escribe la matriz central a la imagen.

*void colorearMatrizTablero():* colorea los casilleros de los tableros del juego.

*void colorearLineasMatrizTablero():* colorea las lineas de los tableros del juego

*void colocarFicha(int, int, int, String rutaImagen):* coloca en las coordenadas x, y, z del tablero del juego, la ficha que se lee de la imagen rutaImagen

*BufferedImage cargarImagen(String rutaImagen):* lee la imagen en rutaImagen y retorna la imagen si corresponde

*void escribirTablero():* escribe en la imagen los tableros del juego

*void escribirArchivo():* escribe la imagen del juego al archivo NOMBRE_ARCHIVO_SALIDA

*void escribirFichaAlBitmap(Casillero<Ficha> casillero, Color color):* escribe al casillero la ficha con el color del jugador en la imagen salida

*String generarRutaRelativa(String nombreDeArchivo):* Retorna una cadena del formato RUTA_RELATIVA_IMAGENES/nombreDeArchivo

Casillero: TDA para las funciones de validacion y cambio en los casilleros del tablero.

*Metodos:*

*Casillero(int, int, int):* crea un casillero con las coordenadas pasadas por parametros y sus vecinos.

*void validarCoordenadas(int, int, int):* se valida que las coordenada sea mayor o igual a 1.

*void validarCoordenadasDeVecino(int, int, int):* valida que (x, y, z) sean coordenadas válidas, si alguna es inválida tira Exception

*void validarCoordenadaDeVecino(int):* valida que sea una coordenada válida, si es invalida tira Exception

*int invertirCoordenadaDeVecino(int i):* retorna el negativo del parametro int

*void vaciar():* elimina el dato del casillero

*boolean estaOcupado():* verdadero si hay un dato en casillero, falso si el casillero está vacío

*boolean tiene(T dato):* verdadero si el dato del casillero equivale al dato pasado por parámetro, falso no si equivale

*boolean existeElVecino(Movimiento movimiento):* verdadero si existe el casillero vecino que esté en la dirección 'movimiento', falso si no existe.

Color: TDA para el funcionamiento de las fichas con los colores elegidos por el jugador

*Métodos:*

*String getRutaDeImagen(Color color):* devuelve la ruta de la imagen según el enum.

*Color getColorJugador(int numero):* el color que corresponde al número.

Dado: TDA creado para que simule las funciones de un dado físico, el mismo genera números aleatorios desde el 1 al 6.

Dirección: Para indicar todas las direcciones alrededor de una ficha (por ejemplo para verificar si hay un ganador)

EstadoDeBloqueo: Enum para indicar si la ficha se encuentra bloqueada o no.

EstadoDePartida: TDA para la creación de un estado de juego

*Métodos*:

*EstadoDePartida(Tablero<Ficha> tablero, ListaSimple<Jugador> jugadores, Mazo mazo):* crea un estado de partida con los parametros dados

FIcha: TDA creado las características especiales de la ficha, como puede ser el color y si la misma se encuentra bloqueada o no.

*Métodos*:

*Ficha(Color color):* inicializa la ficha con el símbolo y color dados.

*void bloquear():* bloquear ficha.

Jugador: TDA para métodos asociados a las funciones del jugador.

*Métodos:*

*Jugador(String nombre,* ***int*** *numeroDeColor,* ***int*** *cantidadDeFichasMaximasPermitidas):* inicializa un jugador con el simbolo, nombre y color pasados por parámetro, y con fichas y mano vacías.

*boolean tieneTodasLasFichasEnElTablero():* verdadero si el jugador tiene todas las fichas en el tablero, falso sí no.

*void jugarFi*cha(): juega una ficha del jugador.

*void agregarCa*rtaALaMano(Carta carta): agrega la carta a la mano del jugador.

*void sumarFicha():* Metodo que suma una ficha cuando el jugador utiliza la carta de doble turno.

*void quitarCartaDeLaMano(Carta carta):* quita la carta de la mano del jugador.

Mazo: TDA para la realización de las tareas del mazo. Pueden ser: crear mazo de manera aleatoria, levantar carter, entre otras.

*Métodos:*

*Mazo(int maximoCartas):* crea un mazo de cartas con 50 cartas de todos los tipos. No hay un máximo de cartas por tipo

*Carta generarCartaAleatoria():* retorna una carta de tipo aleatoria.

*void levantarCartas(int cantidadDeCartasALevantar, Jugador jugador):* a la mano del jugador se agregan las cartas solicitadas.

*void agregarManoDelJugador(Jugador jugador):* devuelve todas las cartas de un jugador al mazo.

*void agregarCarta(Carta carta):* agrega la carta al mazo.

Movimiento: Enum para indicar los movimientos de la ficha.

Partida: TDA para la mayoría de las funciones que se necesitan para un juego.

*Métodos:*

*Partida(Tablero<Ficha> tablero, Lista<Jugador> jugadores, Mazo mazo):* inicializa la partida con el tablero, jugadores y mazo pasados por parámetro.

*Jugador iniciar():* inicia la partida, y retorna el ganador.

*Casillero<Ficha> jugarTurno(Turno turno):* juega el turno (si no tiene bloqueos), y retorna el casillero al que se movió o jugó una ficha.

*Casillero<Ficha> jugadaInicial(Tablero<Ficha> tablero, Jugador jugador):* juega la jugada inicial, le pregunta un casillero, y mueve una ficha del jugador ahí.

*Casillero<Ficha> mover(Tablero<Ficha> tablero, Jugador jugador):* mueve la ficha hacia el casillero en la dirección 'movimiento' ingresados por el jugador en el tablero, y retorna el casillero donde se movió la ficha.

*Ficha preguntarFicha():* le pregunta al jugador las coordenadas de un casillero, valida que exista ese casillero, y devuelve la ficha en ese casillero.

*Casillero<Ficha> preguntarCasillero():* le pregunta al jugador las coordenadas de un casillero, valida que exista ese casillero, y lo devuelve.

*boolean verificarGanador(Casillero<Ficha> casillero):* devuelve verdadero si hay ganador

*int contarFichasSeguidasEnDireccion(Casillero<Ficha> casillero, Direccion direccion, Direccion direccionContraria):* la cantidad de fichas seguidas del valor de la ficha, en la dirección respecto del casillero pasados por parámetro, y la dirección contraria.

Tablero: TDA para las tareas asociadas al tablero. cómo crear tablero con las especificaciones mencionadas por los jugadores. Además, de la validación de los casilleros del mismo para saber si existe un ganador o no, entre otras

Teclado: Scanner y diferentes print de pantalla.

TipoDeCarta: TDA de ENUM donde se distinguen los diferentes tipos de cartas y también se realiza la generación de cartas aleatorias para el mazo.

Turno: TDA para la inicialización de turno o otros métodos.

*Métodos:*

*Turno(Jugador jugador):* inicializa el turno con el jugador pasado por parámetro, con un sub turno y ningún bloqueo restante

*void incrementarBloqueosRestantes(int cantidadDeBloqueos):* le suma a la cantidad de bloques restantes del turno, la cantidad pasada por parámetro

ValidacionesUtiles: TDA para realizar validaciones y exportarlo a los otros TDAs y paquetes del proyecto, cuenta con métodos para validar posiciones, validar nulos, validar tipos de datos y validaciones para el juego.

# Agregar TDA EstadoDeTablero y lo que falte

## 6 - Bibliografía


https://es.wikipedia.org/

https://www.unc.edu.ar/

https://docs.aspose.com/drawing/java/create-image-bitmap/