Informe Algoritmos y programación II
Trabajo Práctico N°2

Integrantes:

- Altieri Lamas, Franco Stefano Daniel
- Arispe, Mijail Braian.
- Balderrama, Carlos
- Cano, Ezequiel Martin
- De Bortoli, Rodrigo Gastón
- Saldaña Nigro, Franco

Docentes:

- Calvo, Patricia
- Schmidt, Gustavo Adolfo

Introducción:

El presente proyecto está dirigido a estudiar y comprender el uso de listas, pilas y colas, así como también el diseño e implementación de tipos de datos abstractos. Para ello se diseñó un juego denominado Tatetí V2.0 en donde se explican algunos ejemplos.
Tatetí 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. Una vez que todos los jugadores colocaron sus respectivas fichas, se comienzan a repartir cartas con distintas habilidades.

OBJETIVO:

Generar una pieza de software que simule el funcionamiento del juego Tatetí en su versión multijugador, así como también comprender el uso de las listas, pilas, colas y la utilización del concepto de dato abstracto.

MARCO TEÓRICO:

1) ¿Qué es un svn?
2) ¿Que es una "Ruta absoluta" o una "Ruta relativa"?

1) SVN es un sistema de control de versiones usado para que varios desarrolladores puedan trabajar en un mismo proyecto en forma ordenada. Tiene una arquitectura cliente servidor con controles de concurrencia para cuando varios desarrolladores están trabajando en el mismo archivo.

2) Una ruta o path es donde se localiza un fichero o archivo dentro de nuestro sistema de ficheros que es el equivalente a su dirección. Todos los comandos que hagamos sin especificar una ruta lo hará donde esté situado y existen dos tipos de ruta que debemos diferenciar:

- **Ruta absoluta:** se indica toda la ruta del archivo incluyendo el directorio raíz. Por ejemplo, C:\carpeta1\carpeta2\archivo1.doc
- **Ruta relativa:** se indica la ruta a partir de donde este en ese momento situado. No se incluye el directorio raíz. Si estamos en la ruta C:\carpeta1 y queremos acceder al archivo1 que está dentro de la carpeta2, seria carpeta2\archivo1.

Lista:
Una lista es una estructura de datos que contiene una colección de elementos homogéneos (del mismo tipo) de manera que se establece entre ellos un orden.
Nodos:
Una lista se compone de nodos, que son estructuras de datos que nos permiten registrar datos de interés. Para que estos nodos se conviertan en una lista, debe existir un enlace entre ellos, que en términos más propios se los conoce como punteros. Los punteros, son la

parte de los nodos que nos permiten recorrer la lista y acceder también a las direcciones de memoria donde se almacenan los elementos de la lista.

Pila:

Una pila es un contenedor de objetos que se insertan y se eliminan siguiendo el principio 'Último en entrar, primero en salir' (L.I.F.O.= 'Last In, First Out'). La característica más importante de las pilas es su forma de acceso. En ese sentido puede decirse que una pila es una clase especial de lista en la cual todas las inserciones (alta, apilar o push) y borrados (baja, desapilar o pop) tienen lugar en un extremo denominado cabeza o tope. El Tope de la pila corresponde al elemento que entró en último lugar, es decir que saldrá en la próxima baja.

Cola:

Una cola es un contenedor de objetos que se insertan y se eliminan siguiendo el principio 'Primero en entrar, primero en salir' (F.I.F.O.='First In, First Out'). En ese sentido puede decirse que una cola es una lista con restricciones en el alta (encolar o enqueue) y baja (desencolar, dequeue). El Frente (FRONT) de la cola corresponde al elemento que está en primer lugar, es decir que es el que estuvo más tiempo en espera. El Fondo (END) de la cola corresponde al último elemento ingresado a la misma.

MARCO PRÁCTICO:

La implementación del tablero de juego multidimensional se realizó con la utilización de listas y para la partida se consideraron las siguientes hipótesis:

- Cada jugador tendrá 3 cartas como máximo.
- Si el jugador ingresa una ficha fuera del rango del tablero no pierde su turno.
- Las cartas comienzan a repartirse una vez que todos los jugadores colocaron sus fichas
- El largo de la línea de fichas para lograr el tateti es especificada por el usuario
- Si se anula un casillero, el mismo permanecerá en ese estado por tres turnos.
- Si se bloquea una ficha, el mismo permanecerá en ese estado por **TRES** turnos.

A continuación, se enlistan y detallarán los métodos utilizados para desarrollar el juego:

TDA FICHA:

Post: crea una ficha vacia.

Ficha();

Post: crea una ficha y le asigna como símbolo el caracter

Ficha(char carácter**);**

Pre: la ficha existe

Post: devuelve el valor de la ficha

char **getSimboloFicha();**

Pre: la ficha fue creada

Post: asigna un valor al símbolo de la ficha

void **setSimboloFicha(**char nuevoSimbolo**);**

Pre:la ficha existe

Post: devuelve TRUE si la ficha está bloqueada o FALSE en caso contrario

bool **estaBloqueadaFicha();**

Pre: la ficha existe

Post: bloquea una ficha

void **bloquearFicha();**

Pre:la ficha existe

Post: desbloquea una ficha

void **desbloquearFicha();**

Pre: la ficha existe

Post: asigna un valor a cantidadTurnosRestantesDesbloqueo de la ficha.

void **setTurnosRestantesDesbloqueo(**unsigned int cantidadTurnos**);**

Pre:la ficha existe

Post: devuelve la cantidad de turnos que tiene TurnosRestantesDesbloqueo.

Unsigned int **getTurnosRestantesDesbloqueo();**

Pre:la ficha existe

Post: resta en uno la cantidad de TurnosRestantesDesbloqueo.

void **decrementarTurnosRestantesDesbloqueo();**

TDA JUGADOR:

Pre:el jugador existe

Post: crea un jugador con un nombre y una ficha asignada.

Jugador(string nombre,char simboloFicha**);**

Pre:el jugador existe

Post: libera la memoria utilizada para el jugador

~Jugador();

Pre:el jugador existe

Post: asigna un nombre al jugador

void **setNombreJugador(**string nombre);

Pre:el jugador existe

Post: devuelve el nombre del jugador

string **obtenerNombreJugador();**

Pre:el jugador existe

Post: devuelve el valor de la ficha del jugador

char **obtenerSimboloFichaJugador();**

Pre:el jugador existe

Post: asigna una ficha al jugador

void **setFichaJugador(**Ficha * nuevaFicha);

Pre:el jugador existe

Post: libera la memoria de la ficha del jugador

void **destruirFichaJugador();**

Pre: el jugador existe

Post: devuelve el ID del jugador

size_t **obtenerIdJugador();**

Pre:el jugador existe

Post: asigna un ID al jugador

void **setearIdJugador(**size_t id**);**

Pre:el jugador existe

Post: asigna el mazo del jugador a nuevoMazo

void **setMazoJugador(**Mazo* nuevoMazo**);**

Pre:el jugador existe

Post: devuelve el mazo de un jugador.

Mazo * **getMazoJugador();**

TDA CARTA:

Post: crea una carta sin ninguna habilidad

Carta();

Post: crea una carta con la habilidad "**efectoCarta**"

Carta(habilidadCarta_t efectoCarta**);**

Pre: existe la carta

Post: asigna la habilidad **"numero"** a una carta

void **setHabilidad(**habilidadCarta_t numero**);**

Pre: existe la carta

Post: devuelve la habilidad de la carta

habilidadCarta_t **getHabilidad();**

Post: devuelve un efecto aleatorio de los disponibles:

habilidadCarta_t **generarEfectoAleatorio();**

Pre: existe la carta

Post: imprime por pantalla la habilidad de la carta.

habilidadCarta_t **imprimirHabilidadCarta();**

TDA MAZO:

Post: Crea un mazo con una cantidad de cartas sin efecto

Mazo(int cantidadCartas**);**

Post: Crea un mazo

Mazo();

Pre:

Post: libera la memoria del mazo

~Mazo();

Pre:

Post: agrega al mazo la carta recibida al final del mismo

void **agregarCarta(**Carta * nuevaCarta**);**

Pre: existe el mazo y posee almenos una carta

Post: devuelve la carta superior del mazo

Carta * **obtenerCartaSuperior();**

Pre: existe el mazo

Post: asigna de manera aleatoria efectos a las cartas de un mazo

void **barajarMazo();**

Pre:existe el mazo

Post: libera la memoria dinámica del mazo

void **destruirMazo();**

Pre: existe el mazo

Post: muestra por pantalla un mazo.

void **imprimirMazo();**

Pre: existe la carta

Post: elimina una carta en la posición pedida

void **eliminarCarta(**unsigned int pos);

Pre: existe el mazo

Post: devuelve la cantidad de cartas que tiene un mazo.

Unsigned int **getCantidadCartas();**

Pre: existe el mazo

Post: devuelve un mazo de cartas.

Lista<Carta*> **& getMazoCartas();**

TDA CASILLERO:

Pre:

Post: Inicializa un casillero y le asigna una ficha con el simbo especificado

Casillero(char simboloFicha){

Pre:

Post: libera la memoria del casillero

~Casillero();

Pre: existe el casillero

Post: devuelve TRUE si el casiilero está vacio o FALSE en caso contrario

bool **estaCasilleroVacio();**

Pre: existe el casillero

Post: devuelte TRUE si el casillero esta anulado o FALSE en caso contrario

bool **estaCasilleroAnulado();**

Pre: existe el casillero de destino

Post: copia el contenido de un casillero a otro casillero

void **copiarCasillero(**Casillero * dest**);**

Pre:existe el casillero

Post: inserta la cantidad de turnos restantes por los cuales el casillero está bajo el efecto de una carta

void **setTurnosRestantesDesbloqueo(**size_t cantidadTurnos**);**

Pre:

Post: devuelve la cantidad de turnos restantes por los cuales el casillero está bloqueado

size_t **getTurnosRestantesDesbloqueo();**

Pre:

Post: decrementa en uno la cantidad de **'turnosRestantesDesbloqueo'** de un casillero.

void **decrementarTurnosRestantesDesbloqueo();**

Pre: existet el casillero

Post: bloquea la ficha del casillero

void **bloquearFichaDelCasillero();**

Pre:existe el casillero

Post: asigna una ficha al casillero

void **setSimboloFichaDelCasillero(**char ficha**);**

Pre::existe el casillero

Post: anula un casillero

void **anularCasillero();**

Pre:

Post: devuelve la ficha que tiene el casillero

char **obtenerSimboloFichaDelCasillero();**

Pre::existe el casillero

Post: devuelve el contenido del casillero.

Ficha * **obtenerContenidoCasillero();**

Pre::existe el casillero

Post: desbloquea un casillero

void **desbloquearCasillero();**

Pre::existe el casillero

Post: crea la matriz de vecinos vacia

void **crearMatrizVecinosVacia(**int cantFilas,int cantCol,int cantProf);

Pre::existe el casillero

Post: libera la memoria de la matriz de vecinos

void **destruirMatrizVecinos(**int cantFilas,int cantCol,int cantProf**);**

Pre::existe el casillero

Post: devuelve la matriz de vecinos del casillero

Lista<Lista<Lista<Casillero*>*>*> * **obtenerMatrizDeVecinos**();

Pre: existe el casillero y la matriz de vecinos

Post: asigna un casillero en la matriz de vecinos

void **setCasillaMatrizVecinos(**size_t cantFilas,size_t cantCol,size_t cantProf,Casillero * punteroCasillero**);**

Pre:existe la matriz de vecinos del casillero

Post: devuelve un casillero adyacente de la matriz de vecinos ubicado en i,j,k

Casillero * **getAdyacente(**int i,int j,int k**)**;

Pre:

Post: devuelve TRUE si la casilla adyacente esta vacia

bool * **estaCasillaAdyacenteVacia(**int i,int j,int k**)**;

pre:existe la matriz de vecinos del casillero

Post: devuelve TRUE si la casilla adyacente existe

bool * **existeCasillaAdyacente(**int i,int j,int k**)**;

Pre:existe la matriz de vecinos

Post: Devuelve la cantidad de fichas iguales de la matriz de vecinos que hay una dirección (Es recursiva)

Size_t * **getLongitud(**int i,int j,int k,Casillero* casilleroOrigen**)**;

Pre: existe la matriz de vecinos

pos:devuelve true si la casilla i,j,k de la matriz de vecino posee la ficha vacia

estaCasillaAdayacenteVacia(int i , int j, int k)

TDA TATETI:

Post:crea una partida tatetí inicializando valores por defecto.

Tateti();

Pre: existe la partida de tateti

Post: Libera la memoria asociada al tatetí

~ Tateti();

Post: inicializa la lista de jugadores del tatetí con sus nombres y fichas.

void **crearJugadores**();

Pre: - existe la lista de jugadores

Post: Muestra por pantalla la lista de jugadores y sus respectivas fichas.

void **imprimirJugadores**();

Pre: Existe la lista de jugadores

Post: inicializa la cola de turnos con los jugadores presentes en el orden que se ingresaron por interfaz.

void **inicializarTurnosJugadores**();

Pre: Existe la lista de jugadores de la partida

Post: libera la memoria de la lista de los jugadores y su contenido.

void **destruirJugadores**();

Pre: -existe la lista de jugadores

Post: devuelve TRUE si el símbolo elegido para la ficha le pertenece a otro jugador.

bool **FichaYaOCupadaPorOtroJugador**(char simboloFichaElegido);

Pre: existe el jugador

Post: devuelve TRUE si “simboloFichaElegido”es igual al símbolo de la ficha del jugador actual

void **PerteneceFichaAljugador**(char simboloFichaElegido);

Pre: -

Post: realiza la configuración inicial de los parámetros del tatetí mediante la interacción con el usuario.

void **iniciarJuego**();

Pre: el juego fue inicializado anteriormente

Post: comienza el juego

void **jugarJuego**();

Post: devuelve TRUE si existe un ganador o FALSE en caso contrario

bool **hayGanador**();

Post: crea el mazo principal.

void **crearMazoPrincipal**();

Pre: existe el mazo principal

Post: libera la memoria del mazo principal utilizada

void **destruirMazoPrincipal**();

Post: crea un mazo para cada jugador.

void **crearMazoJugadores**();

Pre: existe el mazo

Post: borra la memoria solicitada para cada mazo de cada jugador.

void **destruirMazoJugadores**();

Post: avanza el turno actual al siguiente jugador en la cola de turnos.

void **avanzarTurno**();

Post: solicita al usuario que ingrese tres coordenadas que serán asignadas en filas,columnas,profundidad respectivamente.

void **solicitarIngresoDeCordenadas**(int &filas,int & columnas,int &profundidad);

Post: devuelve la minima cantidad de jugadas que deben realizar los jugadores antes de poder hacer TATETI

Size_t **obtenerMinimaCantidadJugadasTateti**();

Pre: existe la lista de jugadores

Post: devuelve la cantidad de jugadores totales.

Size_t **obtenerCantidadJugadoresActuales**();

Pre: - existe el mazo principal y el mazo de los jugadores

Post: saca la última carta del mazoPrincipal y se la asigna al mazo del jugador correspondiente.

void **repartirCartaAlJugador**();

Post: devuelve TRUE en caso de poder mover la ficha o false en caso contrario.

bool **solicitarMoverFicha**(int &filas,int & columnas,int &profundidad);

Post: devuelve TRUE si la ficha ingresada en las coordenadas pedidas es valido o False en caso contrario.

bool **esFichaValida**(int &filas,int & columnas,int &profundidad);

Post: valida el casillero “casilleroOrigen

void **validarCasillero**(Casillero* & casilleroOrigen, int &filas,int & columnas,int &profundidad);

Post: crea una matriz de enteros inicializada en nulo

void **crearMatrizResultadosGanador**(int cantFilas,int cantColumnas,int CantProfundidad);

Pre: existe la martriz

Post: elimina la matriz

void **destruirMatrizResultadosGanador**();

Pre: existe la carta

Post: usa una carta

void **usarCarta**(unsigned int numero);

Pre: existe el casillero

Post: anula un casillero

void **anularCasillero**();

Pre: -existe la carta

Post: elimina una carta de un jugador.

void **eliminarCartaJugador**();

Post: roba una carta de un jugador.

void **robarCartaJugador**();

Pre: existe la ficha

Post: bloquea una ficha de un jugador.

void **bloquearFicha**();

Post: crea el tablero del juego

void **crearTablero**();

Pre: -existe el casillero

Post: devuelve TRUE si el casillero esta libre o FALSE en caso contrario.

bool **estaCasilleroLibre**(size_t fila,size_t columna, size_t profundidad);

Post: devuelve TRUE si las coordenadas pedidas están dentro del tablero o FALSE en caso contrario.

void **estaEnRangoValido**(int fila,int columna,int profundidad);

Pre: existe la ficha y fue colocada por algun jugador de la lista de jugadores

Post: devuelve el ID del jugador al que le pertenezca la ficha.

int **obtenerIdJugadorPropietarioFicha**(char ficha);

Pre:existe el tablero

Post: devuelve el tablero del juego actual.

Tablero * **crearTablero**();

Pre: existe la ficha

Post: devuelve TRUE si la ficha está bloqueada o FALSE en caso contrario.

bool **estaFichaBloqueada**(Casillero * casilleroOrigen);

Pre: -

Post: imprime por pantalla las fichas correspondientes a un jugador.

void **imprimirCordenadasFichaJugador**(char ficha);

Pre: existe un casillero con una ficha

Post: resta en uno la cantidad de turnos en los que un casillero o ficha permanecerán inhabilitados.

void **decrementarTurnosCasillerosYFichas**();

Pre: existe el casillero

Post: desbloquea el/los casilleros/s o la/s ficha/s que estaban anuladas si sus turnos bloqueados llegan a cero

void **desbloquearCasilleroYFicha**();

Pre: existe el casillero

Post: devuelve TRUE si el casillero esta anulado o FALSE en caso contrario.

bool **estaCasilleroAnulado**(size_t fila,size_t columna,size_t profundidad);

CONCLUSIÓN:

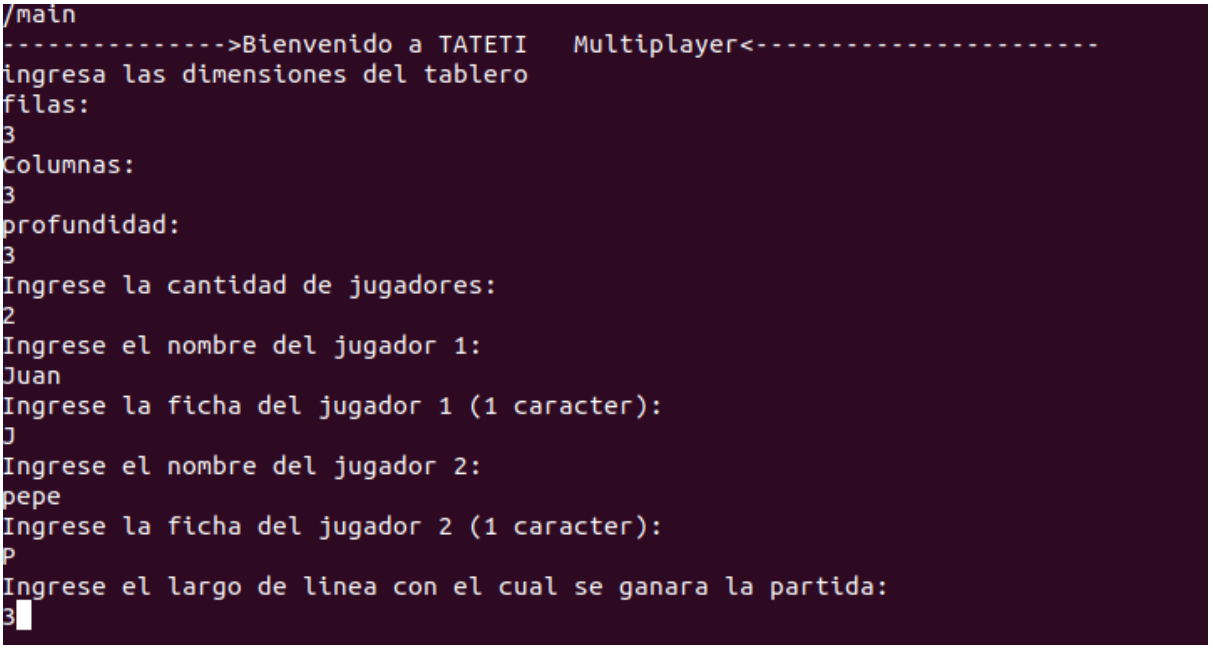
De acuerdo al proyecto realizado podemos concluir:

- Las listas son posiblemente las estructuras de datos más fáciles y sencillas de estudiar, aprender y entender.
- La resolución de una función relacionada con listas es fácil y rápida, en especial porque no se requieren demasiadas líneas de código, por ende, la solución es inmediata
- Las listas, así como también otro tipo de estructuras similares, son útiles a la hora de trabajar problemas como PILAS y COLAS, ya que se maneja la misma lógica de agregar, borrar o buscar elementos.
- La utilización de la programación orientada a objetos así como el concepto de tipo de dato abstracto permiten escribir un código mucho más legible para los programadores y también facilita la interacción de los miembros del equipo con el código.
- Es muy útil contar con un gestor de versiones, en nuestro caso utilizamos GitHub.

MANUAL DE USUARIO

Esta guía contiene la información que usted necesita para utilizar correctamente el producto.

1. Inicie el juego.
2. Una vez iniciado se le pedirá que ingrese el tamaño del tablero,tenga en cuenta que el mínimo es un tablero de 2x2x2.
3. Ingrese la cantidad de usuarios que jugaran la partida. (de 2 a 6 jugadores)
4. Ingrese el nombre y ficha de cada jugador, recordando que la ficha está representada por una única letra o carácter.
5. Ingrese el largo de la línea que necesita lograr un jugador para lograr tateti.
6. Luego respetando el turno del jugador indicado por pantalla, cada jugador debe ir ingresando la posición donde desea colocar su ficha en el tablero.
7. Una vez que los jugadores colocaron todas sus fichas, empezaran a recibir una carta por turno, la máxima cantidad de cartas que puede tener cada jugador es de 3.
8. Se le preguntará al jugador si desea hacer uso de su carta,en caso de que lo prefiera puede no utilizarla y se le pedirá que mueva una ficha a una posición que se encuentre vacía.



MANUAL DE PROGRAMADOR

El siguiente manual tiene como objetivo servir de referencia a futuros programadores que trabajen en el proyecto.
Entorno de desarrollo
Para trabajar con el proyecto se necesita tener instalados los siguientes programas y dependencias:

- Eclipse
- C++

Eclipse:
Eclipse es una plataforma de desarrollo, diseñada para ser extendida de forma indefinida a través de plug-ins. Proporciona herramientas para la gestión de espacios de trabajo, escribir, desplegar, ejecutar y depurar aplicaciones.
Se puede obtener desde https://www.eclipse.org/downloads.

Lenguaje C++:
C++ es un lenguaje de programación orientado a objetos muy potente que evolucionó de la extensión de lenguaje informático “C” y que hoy en día sigue usándose para realizar programación estructurada de alto nivel y rendimiento, como sistemas operativos, videojuegos y aplicaciones en la nube.

Importar proyecto en eclipse:
Para importar el proyecto en eclipse debemos seguir los siguientes pasos:

1. Abrir Eclipse.
2. File->Import->general->Existing projects into Workspace
3. Seleccionamos el directorio del proyecto y finalizamos.

Añadir nuevas características al proyecto:
Tras importar el proyecto en Eclipse, ya estamos en disposición de realizar modificaciones al juego.

Gráficos:

Los gráficos de la partida se realizan con la función imprimirTablero(), la misma imprime en un conjunto de bitmaps el estado actual del tablero. Por defecto la misma se encuentra habilitada pero puede desactivarse haciendo uso del método desactivarGraficarBmp() perteneciente a la clase Tateti.
Los bitmaps son graficados haciendo uso de la librería EasyBmp y de las imágenes que se encuentran anexadas al proyecto ‘casillero_vacio.bmp’ y ‘numbers.bmp’. Por lo tanto es importante que las mismas no se borren del proyecto para un óptimo funcionamiento,

A continuación se listaran los TDAS utilizados con sus miembros y sus métodos, una lista completa de los métodos fue brindada páginas más arriba, aquí solo se listaran los más importantes junto a todos los miembros que conforman el TDA.

TATETI:

Este TDA contendrá la información de todo el juego (tablero, casillero, carta, mazo, jugador).

miembros:

- **.tableroDeJuego:Es el tablero de juego**
- **.jugadaAnterior : funciona como copia del tablero para poder restaurar un estado anterior**
- **.listaDeJugadores :Es la lista que almacena los jugadores actuales**
- **.colaDeTurnos :Es la cola que almacena el orden en el que se**
- **llevan a cabo los turnos**
- **.cantidadJugadasRealizadas :Lleva la cuenta de la cantidad**
- **de jugadas que se realizaron en la partida actual**
- **.mazoPrincipal: Es el mazo principal del juego**
- **.turnoActual : indica de que jugador es el turno actual**
- **.matrizResultadosChequeoGanador: matriz numerica utilizada para el calulo de longitudes de las lineas**
- **.largoLineaGanarTateti: Almacena el largo que debe tener una linea de fichas para ganar el tateti**
- **.hayTateti: es un booleano que indica si se logro el tateti o no**
- **.saltarJugadorSiguiente : Variable de estado para saltar un jugador**
- **.graficarBitmap: booleano que habilita o deshabilita la renderizacion del tablero en formato bitmap**

métodos:

Tateti(): esta función se encargará de crear una partida tatetí lista para cargarla con los datos de la partida.
~Tateti: esta función se encargará de liberar toda la memoria dinámica asociada al tatetí.
crearJugadores(): esta función se encargará de crear la lista de jugadores en memoria y de setearlea sus nombres y fichas correspondientes.
imprimirJugadores(): muestra por pantalla la lista de jugadores con sus fichas.
InicializarTurnosJugadores(): esta función se encargará de inicializar la cola de turnos con los jugadores presentes en el orden que se ingresaron por interfaz.
destruirJugadores(): esta función se encargará de liberar la memoria de la lista de los jugadores y su contenido.
iniciarJuego(): esta función se encargará de realizar la configuración inicial de los parámetros del tatetí mediante la interacción con el usuario.
jugarJuego/(: esta función se encargará de comenzar el juego y es la que contiene el bucle principal del mismo.
hayGanador(): esta función se encargará de devolver si existe un ganador del juego.
crearMazoPrincipal(): esta función se encargará de crear el mazo principal.
destruirMazoPrincipal(): esta función se encargará de liberar la memoria de la lista del mazo principal

avanzarTurno(): esta función se encargará de avanzar el turno actual al siguiente jugador en la cola de turnos.

TABLERO:

Este TDA contendrá las dimensiones del tablero junto con los casilleros correspondientes en función de la dimensión.

miembros:

- **.casilleros: es la lista de listas de lista de punteros a casillero**
- **.cantFilas : es la cantidad de filas que tiene el tablero**
- **.cantColumnas : es la cantidad de columnas que tiene el tablero**
- **.cantEnProfundidad : es la cantidad de profundo que tiene el tablero**

métodos:

Tablero: Esta función se encargará de crear un tablero de dimensiones X Y Z (los valores deben ser mayores a cero.
getCasilla: Recibe tres valores X, Y, Z (mayores a cero) y devuelve la ficha que se encuentra en la casilla elegida.
setCasilla: Recibe tres valores X, Y, Z y una ficha y la coloca en la casilla seleccionada.

CASILLERO:

Este TDA almacenará la información de una ficha y los posibles estados que pueda tener un casillero.

miembros:

- **.contenidoCasillero : Es la ficha almacenada por el casillero**
- **.turnosRestantesDesbloqueo: es la cantidad de turnos antes de poder desbloquearse**
- **.casilleroAnulado: indica si en ese casillero puede colocarse una ficha**
- **.matrizDeVecinos: posee la matriz de vecinos del casillero**

métodos:

casillero: Esta función se encargará de crear un casillero y una ficha sin asignar.
estaCasilleroVacio: Esta función se encargará de informar si un casillero está vacío.
estaCasilleroAnulado: Esta función se encargará de informar si un casillero esta anulado.
copiarCasillero: esta función se encargará de copiar el contenido de un casillero a otro (previamente le casillero debe existir).
setTurnosRestantesDesbloqueo: esta función se encargará de asignar una cantidad de turnos en los cuales un casillero está bajo los efectos de una carta.
getTurnosRestantesDesbloqueo: esta función se encargará de informar la cantidad de turnos restantes por los cuales el casillero está bloqueado.

decrementarTurnosRestantesDesbloqueo: esta función se encargará de decrementar en uno la cantidad de 'turnosRestantesDesbloqueo' de un casillero.
bloquearFichaDelCasillero: esta función se encargará de bloquear la ficha de un casillero.
setFicha: esta función se encargará de asignar una ficha al casillero.
anularCasillero: esta función se encargará de anular un casillero.

FICHA:

Este TDA contendrá la información del valor de una ficha y el posible estado que puede tener.

miembros:

- **.simbolo : es el caracter representativo de la ficha**
- **.establoqueada: representa si la ficha puede moverse o no**
- **.turnosRestantesDesbloqueo : es la cantidad que restan de turnos antes de desbloquearse**

métodos:

Ficha: esta función se encargará de crear una ficha vacía.
Ficha (char carácter): esta función se encargará de crear una ficha con el valor ingresado por el jugador.
getSimboloFicha: esta función se encargará de devolver el valor de la ficha.
estaBloqueadaFicha: esta función se encargará de informar si una ficha está bloqueada.
bloquearFicha: esta función se encargará de bloquear una ficha.
desbloquear Ficha: esta función se encargará de desbloquear una ficha.

CARTA:

Este TDA contendrá la información del valor de la habilidad de una carta.

miembros:

- **.casilleros: es la lista de listas de lista de punteros a casillero**
- **.cantFilas : es la cantidad de filas que tiene el tablero**
- **.cantColumnas : es la cantidad de columnas que tiene el tablero**
- **.cantEnProfundidad : es la cantidad de profundo que tiene el tablero**

métodos:

Carta: esta función se encargará de crear una carta sin habilidad.
Carta (habilidadCarta_t efectoCarta): esta función se encargará de crear una carta con una habilidad obtenida aleatoriamente.
setHabilidad: esta función se encargará de asignar una habilidad a la carta.
getHabilidad: esta función se encargará de devolver la habilidad de la carta.
generarEfectoAleatorio: esta función se encargará de generar y devolver una habilidad aleatoria.
~carta: esta función se encargará de destruir una carta.

MAZO:

Este TDA contendrá una lista de cartas.

miembros:

- **.mazoCartas : es el conjunto de todas las cartas que posee el mazo**

métodos:

Mazo: esta función se encargará de crear un mazo con una cantidad de cartas sin efecto.
agregarCarta: esta función se encargará de agregar al mazo una carta al final del mismo.
obtenerCartaSuperior: esta función se encargará de devolver la carta superior del mazo.
barajarMazo: esta función se encargará de asignar de manera aleatoria efectos a las cartas un mazo.
~destruirMazo: esta función se encargará de liberar la memoria de un mazo.

JUGADOR:

Este TDA contendrá información del nombre del jugador, ficha, cartas y un id que lo identifique.

miembros:

- **.nombreJugador : Es el nombre del jugador**
- **.fichaJugador : Es la ficha del jugador**
- **.mazoJugador : Es el mazo propio del jugador o 'mano' del jugador**
- **.idJugador: Es un identificatorio del jugador**

métodos:

Jugador: esta función se encargará de crear un jugador con un nombre y una ficha asignada.
setNombreJugador: esta función se encargará de asignar un nombre al jugador.
obtenerNombreJugador: esta función se encargará de devolver el nombre del jugador.
obtenerSimboloFichaJugador: esta función se encargará de devolver el valor de la ficha del jugador.
setFichaJugador: esta función se encargará de asignar una ficha al jugador.
destruirFichaJugador: esta función se encargará de liberar la memoria de la ficha del jugador.
obtenerIdJugador: esta función se encargará de devolver el id del jugador.
setearIdJugador: esta función se encargará de asignar el id a un jugador.