Fiuba
Facultad de Ingeniería
Universidad de Buenos Aires

U.B.A. FACULTAD DE INGENIERÍA

Departamento de Informática

75.59 – Técnicas de Programación Concurrentes I

Primer Proyecto: ConcuPig

Curso: 2013 – 1er Cuatrimestre

APELLIDO, Nombres	N° PADRÓN
Rodriguez, Sebastian	90202
Schenkelman, Damián	90728
Servetto, Matías	91363

Fecha de Aprobación :
Calificación :
Firma de Aprobación :

Observaciones:

Índice

Análisis del problema

Nos encontramos ante el problema de implementar la simulación del juego "**Chancho va**".

Este juego posee la dinámica de varios jugadores interactuando entre ellos al mismo tiempo, donde la interacción se da ente jugadores vecinos. En esta interacción, un jugador envía una carta a un vecino y recibe la carta del otro.

El juego impone la regla de que esta acción la deben realizar al mismo tiempo todos los jugadores e inmediatamente levantadas las cartas los jugadores deben comprobar si su mano es ganadora.

En el caso que un jugador gana, debe poner su mano (no las cartas) en el centro de la mesa y, ante este evento, el resto de los jugadores debe colocar la mano sobre la del ganador sin importar lo que estaban realizando en ese momento, ni el juego de cartas que tenga en la mano.

Para entrar en este estado de juego, existe un momento previo donde se deben repartir cartas a todos los jugadores. La baraja del juego varía según la cantidad de jugadores y debe ser elegida al comienzo. No hay una regla que indique si las cartas las reparte un "**Dealer**" ajeno al juego o los jugadores deben tomar el rol.

Este proceso de acciones "**Repartir**" luego "**Jugar**" se debe repetir tantas veces como sea necesario para llegar al estado en que un jugador pierda 7 veces (por cada vez que pierde se le asigna de puntaje una letra de la palabra "**CHANCHO**", hasta que completa la palabra).

Dado el previo análisis y las técnicas con las que se requiere que se resuelvan nos encontramos con los siguientes problemas del punto de vista de la concurrencia:

- Debemos implementar la simulación de manera que todos los jugadores realicen las interacciones jugador-jugador "**al mismo tiempo**".
- Realizar la acción del propio jugador sin un orden especifico, o sea, que el enviar y recibir no ocurran de manera secuencial siempre en el mismo orden, deben ocurrir "**al mismo tiempo**".
- Debemos buscar un mecanismo para sincronizar el reparto de cartas y tanto el comienzo como el fin de cada ronda.
- Implementar una manera de informar a todos los procesos del fin de la partida para que cada uno libere de manera correcta los recursos utilizados.

Hipótesis

- No delegamos la responsabilidad de "**Dealer**" a los jugadores, contaremos con una entidad encargada de repartir las cartas.
- No se usa el escenario donde se elimina este jugador y la partida continúa, hasta que queda un solo jugador siendo este el "ganador". La ejecución de la simulación termina cuando un jugador consigue las 7 letras, siendo este el "perdedor" del juego.
- Dado que la baraja se mezcla de manera puramente aleatoria ya que no tenemos el problema real de que la baraja posee un determinado orden y que depende de la manera que el "**Dealer**" baraje se mezclen "**bien**" o "**mal**", el "**Dealer**" de la simulación repartirá las cartas siempre empezando con el jugador 0 y no realizara la rotación de jugador como en una partida con personas reales. Si se alternará entre jugador y jugador la carta repartida.

Casos de uso

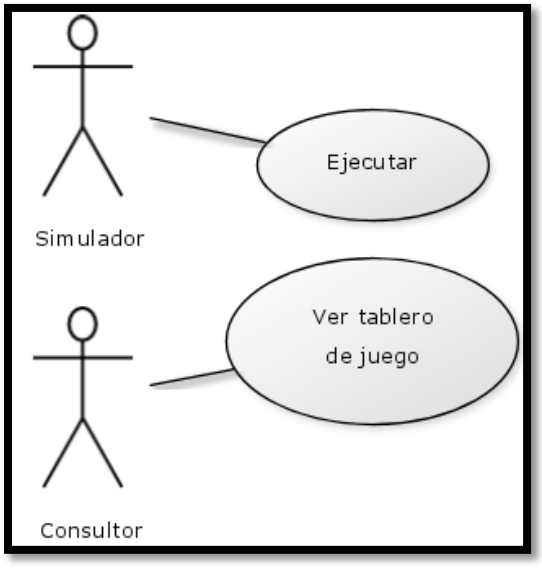
Actores

- Simulador: Persona encargada de lanzar la simulación del sistema.
- Consultor: Persona que consulta el tablero de puntaje de la simulación.

Casos de uso

- Ejecutar: El programa es iniciado con las opciones requeridas por el mismo (cantidad de jugadores y modo de ejecución para realizar o no el loggeo de las acciones). Es lanzado por el usuario con el rol de "Simulador".
- Ver tablero de puntaje: El programa muestra por consola el tablero de puntaje de los jugadores al momento del pedido. Es lanzado por el usuario con el rol de "Consultor".

Diagrama de casos de uso



Especificación de casos de uso

Especificación de caso de uso	
Nombre	Ejecutar
Descripción	El programa es iniciado con las opciones requeridas por el mismo (cantidad de jugadores y modo de ejecución para realizar o no el loggeo de las acciones)
Actor	Simulador
Pre condiciones	-
Post condiciones	• La simulación se encuentra en ejecución. • Se generan archivos semilla para las estructuras de concurrencia utilizadas en la carpeta donde se ejecuta la simulación. • Se genera un archivo de log en la carpeta donde se ejecuta la simulación si la ejecución es modo "debug". • Se ve en pantalla el mensaje para dar la orden de ver puntaje.

Flujo normal de eventos

1. El usuario envía la instrucción de ejecución ingresando las opciones de ejecución (cantidad de jugadores y modo de ejecución).
2. Si se ingresaron correctamente los parámetros **{A}**
3. Si no **{E}**

Flujos alternos

{A} Los parámetros se ingresaron correctamente

1. El sistema lee la cantidad de jugadores y genera los archivos y procesos correspondientes.
2. El sistema lanza los procesos de sincronización de jugadores.
3. Si se ingreso en modo debug, el sistema genera el log.
4. El sistema lanza el proceso del puntaje y este muestra por pantalla la opción de "ingresar una tecla para mostrar puntaje".
5. El sistema comienza a jugar.
6. Fin del caso de uso.

Flujos de excepción

{E} Se ingresaron parámetros incorrectos

1. El sistema muestra un texto de ayuda donde muestra como ejecutar el programa.
2. Termina la aplicación.
3. Fin del caso de uso.

Especificación de caso de uso	
Nombre	Ver tablero de puntaje
Descripción	El programa muestra por consola el tablero de puntaje de los jugadores al momento del pedido.
Actor	Consultor
Pre condiciones	-
Post condiciones	• Por consola se ve el puntaje • Se vuelve a mostrar el mensaje para dar la orden de mostrar el puntaje. • El proceso de consulta terminado si se ingresa la opción de terminarlo.
Flujo normal de eventos	
1. El consultor ingresa una tecla a la consola donde se muestra el mensaje con la instrucción para ver el puntaje. 2. El sistema lee la tecla. 3. Si la tecla es “q”, el sistema termina con el proceso de consulta. 4. Si no, muestro puntaje, luego muestro el mensaje de instrucción para pedir el mensaje. 5. Fin del caso de uso.	
Flujos alternos	
-	
Flujos de excepción	
-	

Resolución del problema

Procesos involucrados en la simulación

Se comenzara la descripción de la resolución del problema explicando la estructura de los procesos generados:

Se pueden dividir en dos grandes rangos:

- Los procesos del jugador.
- Los procesos de sincronización de la partida.

Los procesos del jugador

Se crearon 3 procesos que representan a la entidad de jugador en la simulación:

La cabeza, implementada en la clase "**PlayerHead**". Se encarga de crear los procesos hijos de envió y recepción. Tiene la lógica para la selección de la carta a enviar, saber si gano la partida, poner la mano en la mesa cuando gana o es informado que otro jugador puso la mano, y para sincronizar el envió y recepción de cartas.

El receptor de cartas, implementado en la clase "**PlayerCardReceiver**". Se encarga de recibir una carta de su vecino y de compartir la carta recibida con "*la cabeza*".

El remitente de cartas, implementado en la clase "**PlayerCardSender**". Se encarga de enviar la carta al otro vecino que *la cabeza* le comparte.

Los procesos de sincronización de la partida

Se crearon los siguientes procesos:

La mesa, implementada en la clase "**Table**" y en el "**main**" del programa. Este proceso, como "**main**" del programa, inicializa los archivos y estructuras que se utilizan en el programa. Lanza a los procesos *cabeza* de los jugadores, y el resto de los procesos que se explicaran posteriormente. También se encarga de la terminación de los procesos y el cierre de los recursos una vez terminada la simulación.

Como "**Table**", tiene la responsabilidad en el loop principal del juego: reparte las cartas y envía la señal de la mano en la mesa. El loop termina cuando se detecta que un jugador recibe las 7 letras de "**CHANCHO**".

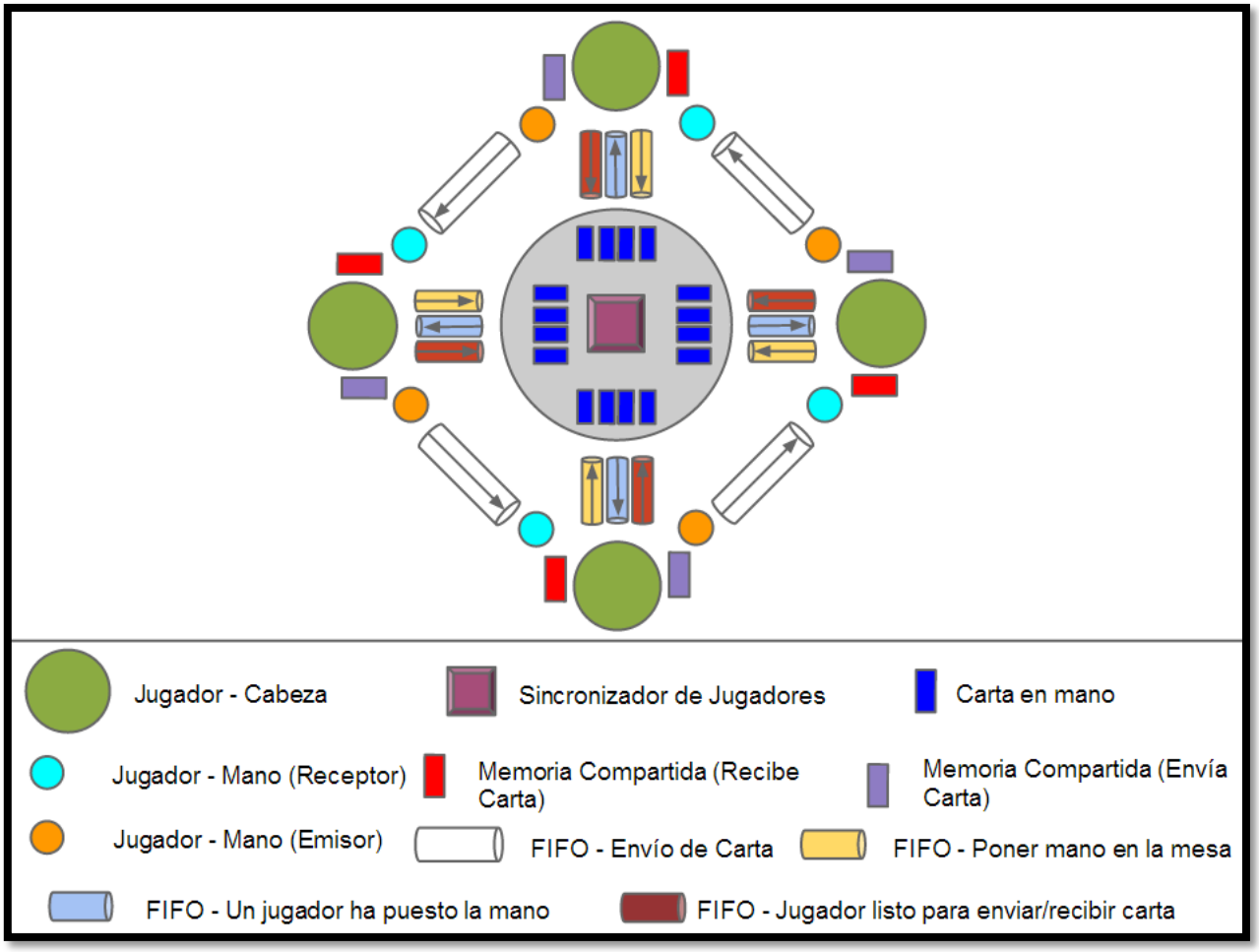
El tablero de puntaje, implementador en la clase "**ScoreBoardController**". Se encarga de leer la tecla de la pantalla y mostrar el puntaje ante ese evento o terminar, dependiendo esto último de si la tecla es "**Escape**" o no. También debe compartir los puntajes y la lógica de la condición de corte de juego con el proceso de *la mesa*.

El sincronizador de jugadores, implementado en la clase "**PlayerSynchronizer**". Se encarga de dar la señal de pasaje de cartas cuando los jugadores están listos para realizar la acción.

Aclaración: la frase "***dar la señal***" que se uso en la sección de "**procesos involucrados**" no hace referencia al mecanismo de señales de concurrencia.

Comunicación entre procesos

Diagrama



Intercambio de cartas

Para poder enviar y recibir las cartas en el juego, todos los jugadores deben estar sincronizados para hacerlo al mismo tiempo. En el juego real, esto se da mediante una señal dada por una persona; en este caso, para simularlo, lo que se realizó fue un esquema donde cada jugador (**PlayerHead**) debe comunicarle a un proceso sincronizador (**PlayerSynchronizer**) que ya está en estado de jugar. Para esto se utilizaron los siguientes mecanismos de concurrencia:

- **Semáforos**:
  - ReadyToSendReceive (1 por jugador)
- **Fifo**:
  - PlayersReady

La lógica de comunicación es la siguiente:

Una vez que **PlayerHead** ha hecho elección de qué carta desea enviar, escribe en el fifo **PlayersReady** su Id, y luego hace wait sobre el semáforo **ReadyToSendReceive**. Al mismo tiempo,

el proceso **PlayerSynchronizer** está leyendo del fifo **PlayersReady**, y cuando ha leído todos los Ids de los jugadores que están jugando, pasa a desbloquearlos haciendo signal sobre cada uno de los semáforos **ReadyToSendReceive**. A partir de aquí el jugador se encuentra desbloqueado y realizando el envío y recepción de cartas.

El mecanismo de concurrencia Fifo fue utilizado para que los jugadores permitan avisar de su estado listo a jugar. Los semáforos usados, tuvieron el fin de sincronizar a los jugadores para que nadie se adelante en el pase o recepción de cartas, sino que lo hiciesen todos al mismo tiempo como en el juego real.

Envío de Carta

Para el envío de cartas, la cabeza del jugador se comunica con la mano que se encarga de mandar las cartas. Para la comunicación entre "**PlayerHead**" y "**PlayerCardSender**", se utilizaron las siguientes estructuras:

- **Semáforos**:
  - SenderSemaphore
  - SentSemaphore
- **Memorias Compartidas**:
  - SharedCard

La lógica de comunicación es la siguiente:

PlayerHead comienza a jugar la ronda, y comienza a elegir carta a enviar. Al mismo tiempo, el **PlayerCardSender** se encuentra haciendo un wait sobre el **SenderSemaphore**. Una vez que **PlayerHead** elige carta a enviar, la escribe en la memoria compartida **SharedCard** y le avisa al proceso **PlayerCardSender** de dicha acción mediante un signal al **SenderSemaphore**. Aquí las acciones se invierten, dado que **PlayerHead** debe esperar ahora a que **PlayerCardSender** envíe la carta, con lo cúal hace un wait sobre el **SentSempahore**. Una vez que **PlayerCardSender** termina de enviar la carta (Esta situación se explica en otro apartado), hace un signal sobre el **SentSemaphore** para avisar que ha finalizado su acción de envío, y vuelve (si el juego no ha terminado) al wait del principio para enviar una nueva carta.

El uso de semáforos permite la sincronización entre ambos procesos, indicando cuando uno debe enviar la carta y el otro esperar, y cuando el otro proceso puede continuar una vez finalizado el envío de la carta. Asimismo, la memoria compartida permite el traspaso, de un proceso a otro, de información de la carta a mandar; otorgando un lugar común donde leer y escribir.

Recepción de Carta

Para la recepción de cartas, la cabeza del jugador se comunica con la mano que se encarga de recibir las cartas. Para la comunicación entre "**PlayerHead**" y "**PlayerCardReceiver**", se utilizaron las siguientes estructuras:

- **Semáforos**:
  - ReceiverSemaphore

    - ReceivedSemaphore
- **Memorias Compartidas**:
    - SharedCard

La lógica de comunicación es la siguiente:

PlayerHead ha elegido su carta a enviar, y se encuentra en estado de envío y recepción de cartas. **PlayerCardReceiver** se encuentra haciendo wait sobre **ReceiverSempahore**, esperando a que se le dé pase libre para comenzar el proceso de recepción de cartas. **PlayerHead** da signal sobre **ReceiverSemaphore** para recibir la nueva carta, y hace wait sobre **ReceivedSemaphore** con el fin de esperar que se le informe que la carta nueva ha sido recibida con éxito. **PlayerCardReceiver** obtiene el permiso de recibir carta a partir del signal de **PlayerHead**, con lo cual realiza dicha acción (Situación explicada en otro apartado) y escribe la nueva carta en la memoria compartida **SharedCard**. Al finalizar da signal sobre **ReceivedSemaphore** para indicar a **PlayerHead** que la carta se ha recibido, y si el juego no ha finalizado vuelve a hacer wait sobre el **ReceiverSempahore** para comenzar nuevamente el proceso.

El uso de semáforos permite la sincronización entre ambos procesos, indicando cuando uno debe recibir la carta y el otro esperar, y cuando el otro proceso puede continuar una vez finalizada la recepción de la carta. Asimismo, la memoria compartida permite el traspaso, de un proceso a otro, de información de la carta que se ha recibido; otorgando un lugar común donde leer y escribir.

Envío y Recepción de Carta entre 2 Jugadores.

Para la comunicación entre 2 jugadores, se encargan la mano que envía cartas de un jugador y la mano que recibe cartas del otro jugador. Se usaron las siguientes estructuras:

- **Fifo (***CardPassingFifo***)**

La lógica de comunicación es la siguiente:

Una vez que los jugadores han elegido las respectivas cartas a mandar, comienza el proceso de envío de dicha carta y recepción de una nueva. Para esto, el jugador que envía la carta utiliza la mano *PlayerCardSender;* la cuál obtiene la carta que se encuentra en la memoria compartida SharedCard(Donde *PlayerHead* ha escrito la carta que desea mandar), la serializa y la escribe sobre el *Fifo* que utilizan el jugador y el jugador vecino al cuál le llega la carta. Al mismo tiempo el jugador vecino se encuentra esperando la carta utiliza la mano **PlayerCardReceiver** que, de manera análoga al **PlayerCardSender**, se encuentra leyendo sobre el *Fifo*; al leer la nueva carta, la de-serializa y la escribe en la memoria compartida que tiene con su **PlayerHead** para que este la guarde en su mano.

La utilización de un Fifo permite simular el envío y recepción de carta de un jugador a otro, al escribir sobre dicho fifo la información de la carta a pasar.

Jugador gana ronda – Poner mano en la mesa.

Cuando un jugador consigue 4 cartas de igual número, dicho jugador está en disposición para ganar la ronda que se está jugando. Para lograr ganar la ronda, debe poner la mano en la mesa. Con lo cual el jugador (**PlayerHead**) y la mesa (**Table**) deben estar comunicados.

Para representar esta acción, el jugador y la mesa se encuentran comunicados mediante un **Fifo** (**HandDownFifo**) en el cuál el jugador escribe su id para indicar dicho estado de ganador. La mesa una vez que repartió las cartas y desbloqueo los jugadores para que jueguen, se encuentra esperando en un ciclo a que algún jugador escriba sobre el *Fifo* su id indicando que puso la mano en la mesa y por ende que la ronda debe ser finalizada mediante los otros jugadores poniendo las respectivas manos en la mesa. De no obtener ningún id en la lectura del *Fifo*, indica que ningún jugador ha ganado esa ronda, por lo cual vuelve a leer del *Fifo* hasta que un jugador indique su condición de ganador.

En el caso de un jugador que no ha ganado la ronda, igualmente escribe en el *Fifo* su condición de *no* ganador para que la mesa pueda tener un sincronismo con todos los jugadores.

La utilización de un Fifo permite simular la puesta de mano en la mesa, mediante el envío de información al escribir en dicho fifo.

Hay ganador de ronda – Jugadores deben poner mano en la mesa

Mientras se está jugando la ronda, existe un momento en el cuál un jugador logra conseguir 4 cartas iguales y pone la mano en la mesa indicando su condición de ganador del juego. A partir de aquí, el resto de los jugadores deben poner sus respectivas manos en la mesa y quién quede último será el perdedor de la ronda.

Para llevar a cabo esta situación, una vez que el jugador le ha informado a la mesa que ha puesto la mano en la mesa, la mesa (*Table*) se debe encargar de avisarle a todos los jugadores que ya hay un jugador que ha puesto la mano en la mesa. Para lograr esto, escribe en un **Fifo** (*PlayerWonFifo*) un valor de 1 para indicar que un jugador ha ganado la ronda (0 en caso contrario, para que los jugadores sigan jugando una mano más dentro de la ronda), repitiéndolo por cada jugador.

A partir de aquí, el jugador recibe por el **Fifo** (*PlayerWonFifo*) la alerta de que alguien ha ganado y que debe poner su mano en la mesa. De tener que poner la mano, y no haber sido él quien ya ganó, el jugador escribe en el **Fifo***(HandDownFifo*) su id para indicar que pone la mano en la mesa.

La mesa comienza a leer del **Fifo***(HandDownFifo*) nuevamente, esperando que todos los jugadores hayan escrito su id. El último jugador en escribir su id será quién, análogamente, haya puesto último la mano en la mesa, y por ende pierde la ronda.

La utilización de los Fifos permite en ambos casos sincronizar a los procesos, indicando o bien que cada jugador ha puesto la mano en la mesa mediante la escritura de su id en el Fifo, o bien que existe un jugador que ha ganado mediante la escritura de un valor en el Fifo.

GAME OVER - Jugador coleccionó las 7 letras

Cuando un jugador llega a las 7 letras, esto indica que el jugador ha perdido el juego, y que por ende el juego ha llegado a su fin. Para simular este fin de juego, la mesa controla cada vez que termina una ronda si existe alguien en condición de perdedor. De existir alguien en condición de perdedor, la mesa entonces procede a terminar.

Para terminar el juego, la mesa utiliza una señal de notificación *GameOver*. La misma es mandada a cada uno de los procesos de los jugadores y al proceso de sincronización para que ejecuten los respectivos handlers de la señal, y generen un "*graceful quit*".

Tanto los **PlayerHead** como el **PlayerSinchronizer** utilizan el mismo mecanismo para manejar la señal, modificando una variable propia que indica si el juego está o no finalizado.

Utilizar una señal para indicar que el juego ha finalizado, permite que un proceso corriendo pueda prestar atención a dicho acontecimiento y modificar su estado para poder actuar en consecuencia (Ejemplo: El jugador dándose cuenta que debe terminar y no seguir con la ejecución)

Tablero de puntajes – Table & ScoreboardController

El tablero de puntajes tiene una característica especial, que es que es utilizado por 2 procesos totalmente independientes. Por un lado **Table** se encarga del juego en sí mismo, mientras que **ScoreboardController** es el encargado de mostrar, cada vez que se lo solicita, el tablero de puntajes actual.

Sin embargo, ambos procesos utilizan un mismo tablero de puntajes **Scoreboard**. Dicho **Scoreboard** posee un vector de *Scores* (uno por cada jugador existente), que contienen cada uno un **SharedScore** donde almacenan el puntaje.

El uso de las memorias compartidas permite que los procesos **Table** y **ScoreboardController** puedan acceder al mismo recurso, haciendo el uso respectivo necesario.

Aclaración: Las distintas memorias compartidas utilizadas a lo largo del programa, utilizan el mecanismo de concurrencia de Lock con cada acción que realizan sobre dicha memoria. Esto permite evitar problemas de acceso sobre la misma memoria por parte de procesos distintos.

Paquete Mecanismos de Concurrencia

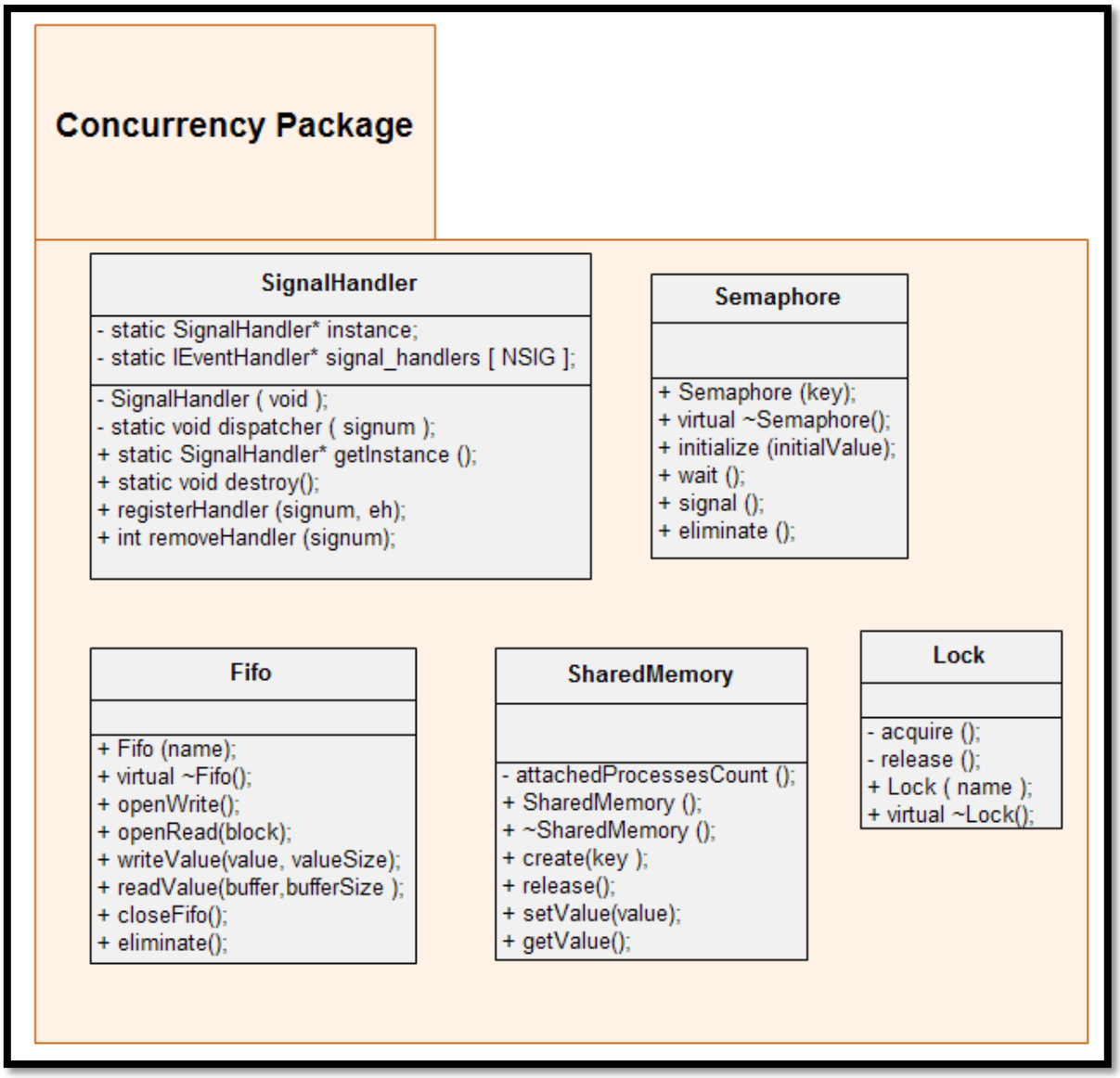


Diagrama de clases

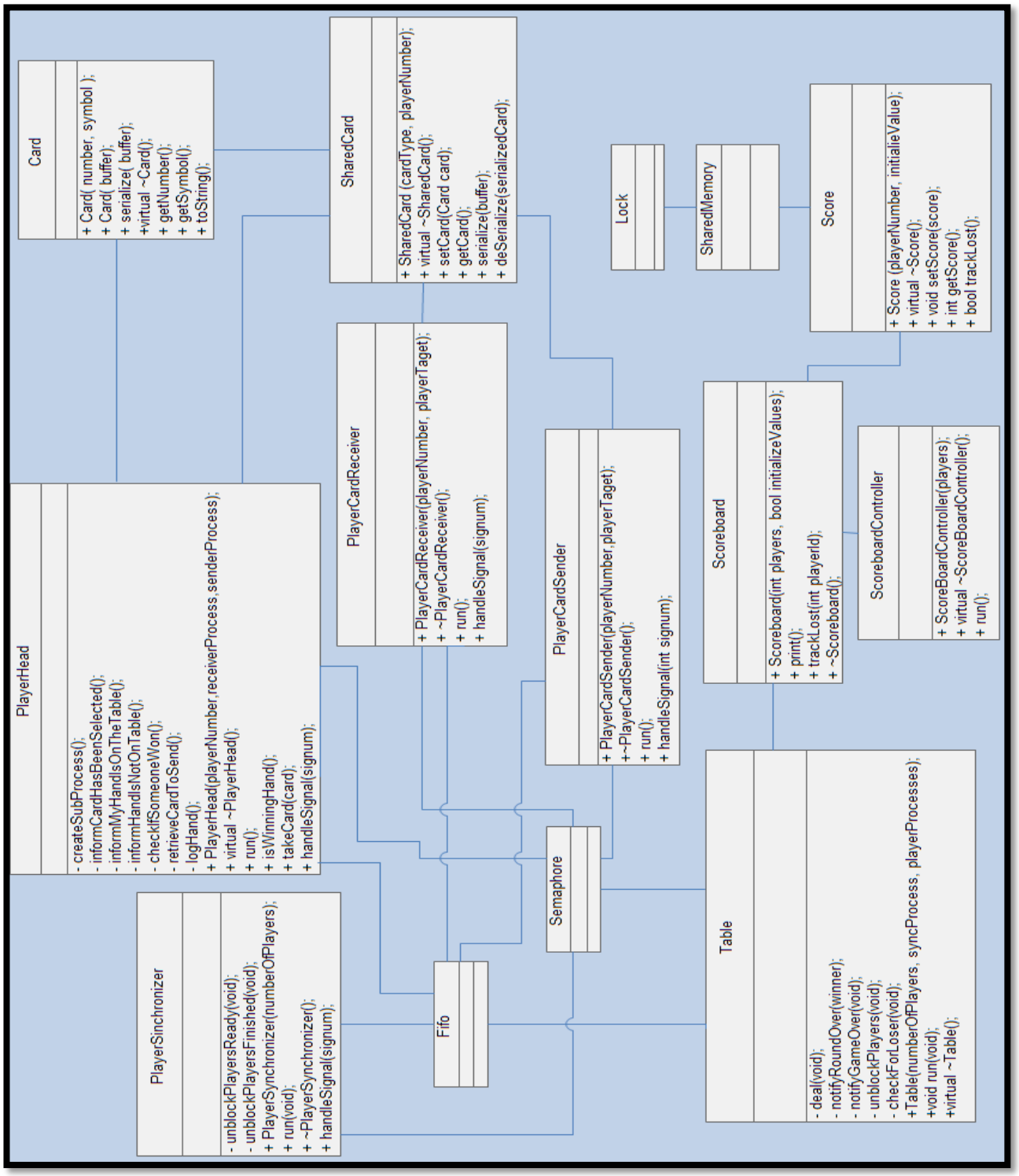
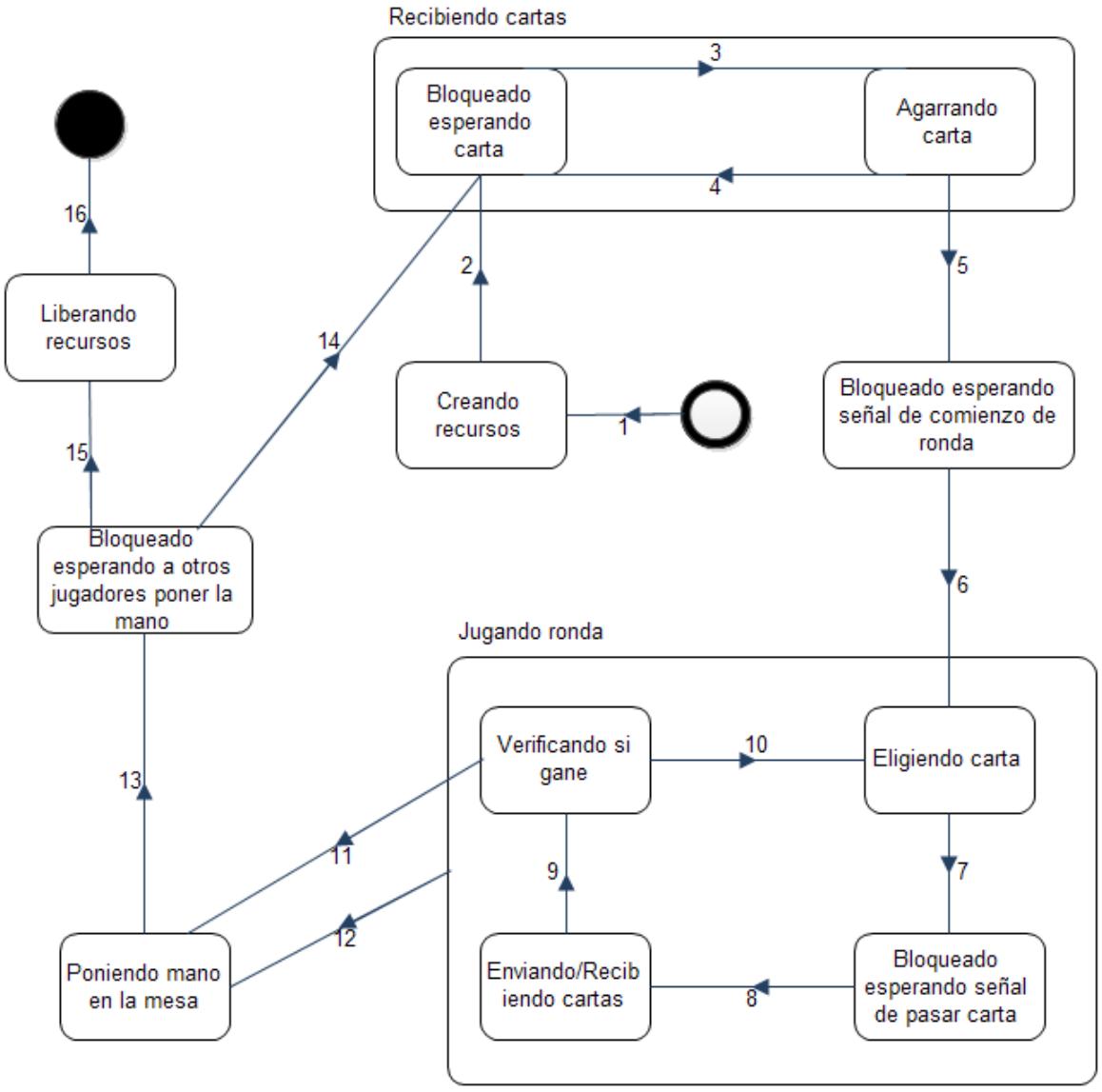


Diagrama de estados del jugador

Diagrama



Diccionario de transiciones

1. Creación del jugador.
2. Finalización de creación de recursos.
3. Recepción de carta.
4. Cantidad de cartas es menor a 4.
5. Cantidad de cartas es igual a 4.
6. Recepción de aviso de comienzo de ronda.
7. Carta elegida.
8. Recepción de aviso de envío/recepción de carta.

9. Envío/Recepción finalizada.
10. No se tiene mano ganadora.
11. Poniendo mano por cartas ganadoras.
12. Recibe aviso de que otro jugador pone la mano.
13. Mano ya puesta en la mesa.
14. Todos los jugadores pusieron la mano.
15. Game Over. Todos los jugadores pusieron la mano y hay un perdedor.
16. Recursos liberados.

Problemas conocidos

String Class memory leak

Al correr *Valgrind* sobre el programa, el resultado del análisis hecho da que existe pérdida de memoria ("*memory still reachable*"). Si uno investiga de donde provienen estos *memory leaks*, encuentra que es consecuencia directa de utilizar la clase *String*.

Investigando sobre este tipo de leaks, se encontró que es un "*problema común*" dentro del análisis de *Valgrind*, y que es independiente al código escrito. Este problema radica en el hecho de que las librerías de *C++* utilizan sus propios "*memory pool allocators*"; con lo cual la memoria no es liberada inmediatamente y devuelta al SO, sino que se mantiene en ese *pool* para ser reutilizada.

Por lo tanto, dado que estos "*memory pool allocators*" no son liberados cuando el programa llega a su fin, hace que el análisis de *Valgrind* reporte *memory leaks*.

Screenshots:

```
==4375== 29 bytes in 1 blocks are possibly lost in loss record 3 of 31
==4375==    at 0x4C2AF8E: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==4375==    by 0x4EEE3B8: std::string::_Rep::_S_create(unsigned long, unsigned long, std::allocator<char> const&) (in
/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.17)
==4375==    by 0x4EEFD94: char* std::string::_S_construct<char const*>(char const*, char const*, std::allocator<char> const&,
std::forward_iterator_tag) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.17)
==4375==    by 0x4EEFE72: std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string(char const*,
std::allocator<char> const&) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.17)
==4375==    by 0x4116C9: __static_initialization_and_destruction_0(int, int) (FifoNames.cpp:14)
==4375==    by 0x411850: _GLOBAL__sub_I__ZN9FifoNames11CardPassingE (FifoNames.cpp:17)
==4375==    by 0x41346C: __libc_csu_init (in /home/dschenkelman/workspace/concurrentes/ConcuPig/Debug/ConcuPig)
==4375==    by 0x536C6FF: (below main) (libc-start.c:185)
```

Fuente:

- "*Valgrind Frequently Asked Questions*" - Section 4.1
  http://valgrind.org/docs/manual/faq.html#faq.reports