Universidad San Carlos de Guatemala

Facultad de ingeniería

Escuela de Ciencias y Sistemas

Sistemas Operativos 2

Manual Técnico

ID	Nombre
201701133	Selvin Lisandro Aragón Pérez
201700529	Cristian Manases Juárez Juárez

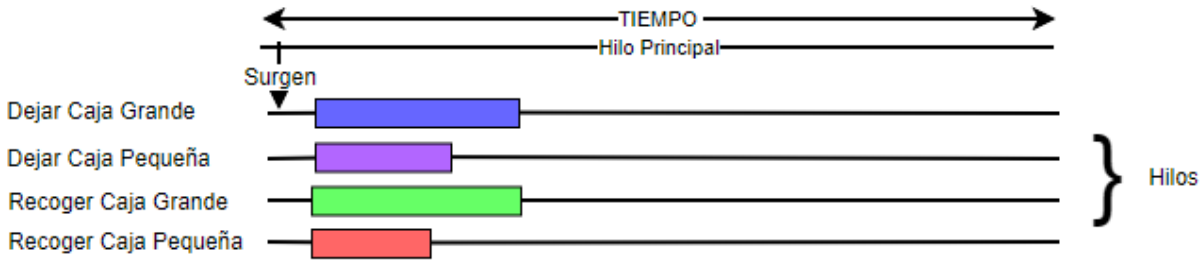
Manual técnico

Problema 1: Centro de Acopio

Para el problema 1 es de un centro en el cual se reciben y se entregan cajas con productos, el centro tiene una estantería con una capacidad máxima de 20 espacios. Las personas nos capaces de dejar cajas y llegar a recogerlas al mismo tiempo.

En los hilos que se pudieron identificar son los siguientes:

- Dejar caja grande
- Dejar caja pequeña
- Recoger caja grande
- Recoger caja pequeña



En la parte de código se ve reflejada de la siguiente manera:

```
//ENTREGA CAJAS GRANDES
Runnable entregas_CJgrande;
entregas_CJgrande = new Runnable() {
    @Override
    public void run() {
        while(true){
            try {
                while(pausa){
                    Thread.sleep(1000);
                }
                int prod_cjgrande = Integer.parseInt(vista.jTextField1.getText());
                int num_grad = (int)Math.floor(Math.random()*prod_cjgrande+1);
                Thread.sleep(num_grad);
                TipoCaja caja = new TipoCaja(1, estante, vista);
                caja.start();
            }catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
};
Thread hiloCJEntregagrande = new Thread(entregas_CJgrande);
hiloCJEntregagrande.start();
```

Se crea un runnable para crear el hilo donde se estará trabajando uno de los diferentes hilos definidos anteriormente. La función while es para que se repita el proceso varias veces,

siendo controlado con un **sleep** ingresado por el usuario. La clase que es heredada de un hilo se debe colocar **extends Thread**, ese fue la clase de la clase TipoCaja.

Recursos Compartidos

Todos los hilos comparten la información de espacios de disponibles en la estantería y las cantidades de cajas que fueron dejadas y recogidas. Esta información está definida en el hilo principal.

```
public class Estanteria {
    private HashMap<Integer, String> ids;
    private int libres;
    private int DejarCajaGrande;
    private int DejarCajaPequenia;
    private int RecogerCajaGrande;
    private int RecogerCajaPequenia;
```

Comunicación y sincronización entre procesos.

Para la sincronización entre los diferentes hilos ya mencionados se usó **synchronized**. Esto se usó para sincronizar los datos con el hilo principal. Un ejemplo de cómo se usó en el código es el siguiente:

```
public synchronized boolean dejar_cajaGrande(){
    if(this.libres <= 1){
        return false;
    }else{
        Pintar dibujo = new Pintar(1,ids, 20-(this.libres - 2), this.DejarCajaGrande + 1,
                                   this.DejarCajaPequenia, this.RecogerCajaGrande,
                                   this.RecogerCajaPequenia);
        this.libres -= 2;
        this.DejarCajaGrande++;
        dibujo.start();

        return true;
    }
}
```

En esta función se llamaba la función de dejar caja grande, se actualizaban los contadores y se llamaba pintar.

Situaciones donde suele surgir problemas de concurrencia y como se solucionaron en caso de que existieran.

Inconsistencia de datos:

Deadlocks: Cuando la estantería esta llena se genera una cola, esta bloquea el proceso dejar_caja o recoger_caja, por lo cual cuando otro cliente que desee dejar o recoger una caja no podrá acceder al recurso. Entonces existirá un punto muerto.

Inconsistencia de datos.

Problema 2: El Barbero Dormilón

En una barbería consta de una sala de espera con n sillas (para este caso 20) y una sala de barbero con una silla. Si no hay clientes para atender el barbero se va a dormir. Si un cliente entra a la barbería y todas las sillas están ocupadas, el cliente abandona el lugar. Si el barbero está ocupado y hay sillas disponibles, el cliente se sienta a esperar. Si el barbero está dormido, el cliente despierta al barbero.

Solución

Para la solución al problema se propone el diagrama de la **figura 1** para manejar los hilos.

La barbería será el centro de los recursos para el manejo de la lógica del problema. Este tendrá una lista para manejar los procesos clientes que se van creando. Estos se crean por medio del Gestor de Clientes que instara un nuevo proceso cada cierto tiempo.

El barbero será un hilo que estará chequeando la lista de procesos cliente para realizar el método cortar cabello.

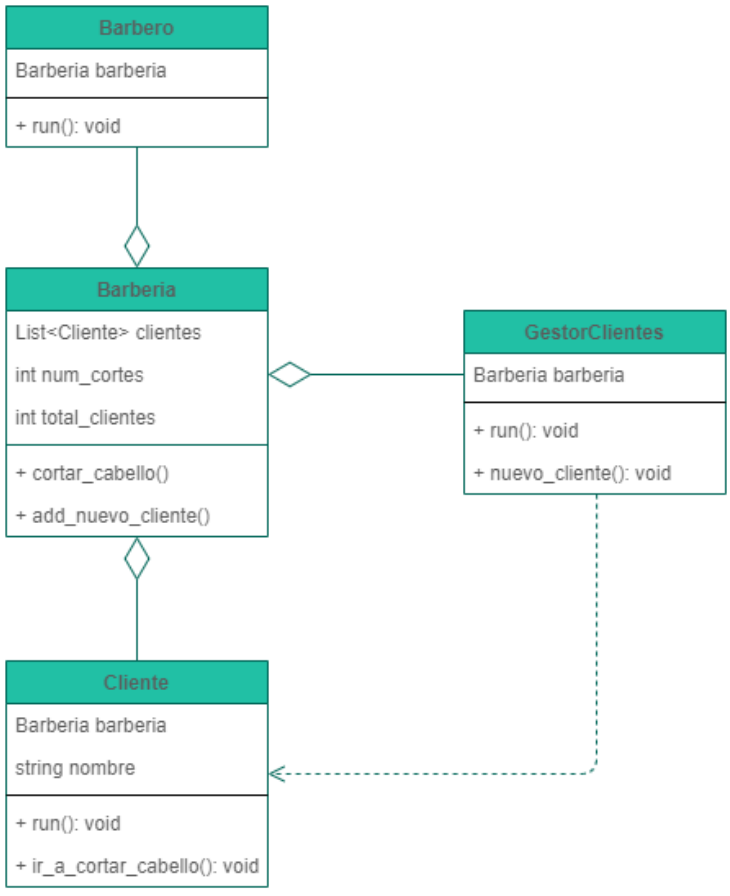


Ilustración 1 Diagrama de clases

En la **figura 2** se muestra el flujo que lleva a la creación de los hilos. El hilo Gestor Clientes se crea al inicio y persiste hasta que la barbería cierre.

El barbero se iniciará cuando la barbería abra y persistirá hasta que cierre.

El cliente se genera cada cierto tiempo implementado por el Gestor de Clientes y este finalizará cuando termine el método cortar cabello.

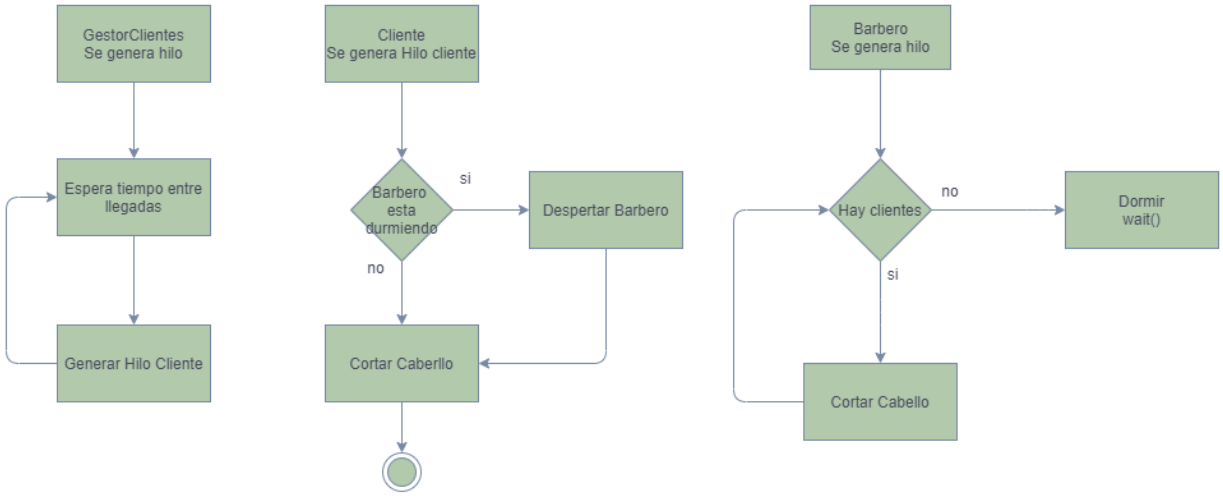


Ilustración 2 Diagrama de flujo

El principal problema que se tiene con estos hilos es la **inconsistencia de datos**. Estos datos son administrados por el objeto Barbería, específicamente por la lista de clientes.

Para dar solución a este problema se utiliza la función ***synchronized*** sobre la lista de clientes, esto lo que hace es bloquear el recurso, que no pueda ser accedido por otro hilo hasta que este termine.

```
synchronized(listClientes){
    while(listClientes.isEmpty()){
        System.out.println("Barbero durmiendo");
        this.x.barberoDuerme();
        try{
            listClientes.wait();
        }catch(InterruptedException ie){
            ie.printStackTrace();
        }
        System.out.println("Barbero se despierta");
        this.x.barberoDespierta();
    }
    cliente = (Cliente)((LinkedList<?>)listClientes).poll();
}
```

Ilustración 3 Bloque de recurso

Se utiliza la función ***wait*** para realizar una espera simulando que el barbero se duerme. Entonces nadie más podrá acceder al recurso hasta que sea devuelto el hilo con la función ***notify***.

```
public void add(Cliente cliente){
    System.out.println("Ingresa nuevo cliente" + cliente.getName());
    this.totalClientes++;
    this.total.setText(totalClientes+"");
    synchronized (listClientes){
        if(listClientes.size() >= 20){
            System.out.println("No hay sillas... se va");
            this.no_esperan++;
            this.se_van.setText(this.no_esperan+"");
            return ;
        }

        ((LinkedList<Cliente>)listClientes).offer(cliente);
        this.filaEspera.addElement(cliente.getName());
        this.espera.setText(this.listClientes.size()+"");

        if(listClientes.size() == 1)
            listClientes.notify();
    }
}
```

Ilustración 4 Aviso a recurso de continuación

En otra parte donde se implementó ***wait*** y ***notify*** es para pausar el sistema. Lo cual detiene los hilos hasta que sean devueltos, en esta ocasión se bloque la clase completa par que no puedan ser llamados sus métodos.

```
try{
    synchronized(this){
        while(VistaInicio.pausa){
            this.wait();
        }
    }
}catch(InterruptedException ex){
    Logger.getLogger(Barbero.class.getName()).log(Level.SEVERE, null, ex);
}
```

Ilustración 5 Hilo puesto en pausa

Problema 3: El Barbero Dormilón

El problema consiste en la realización del juego "space invaders" pero será para 2 jugadores. Básicamente se tendrá una pantalla en la cual en la parte superior irán apareciendo naves enemigas que irán descendiendo por la pantalla y los jugadores deberán eliminar.

Solución

Para la solución al problema se propone el diagrama de la **figura 6** para manejar los hilos.

Se tendrán varios hilos para darle vida al juego. Se tiene un objeto para los **Enemigos** lo cual será administrado por el **Gestor de enemigos**, este generará enemigos cada cierto periodo de tiempo. El enemigo se accederá a los recursos que están en el objeto Juego que es donde se manejará la lógica del juego y consistencia de los datos. Las balas estarán siendo generadas por el **Jugador.**

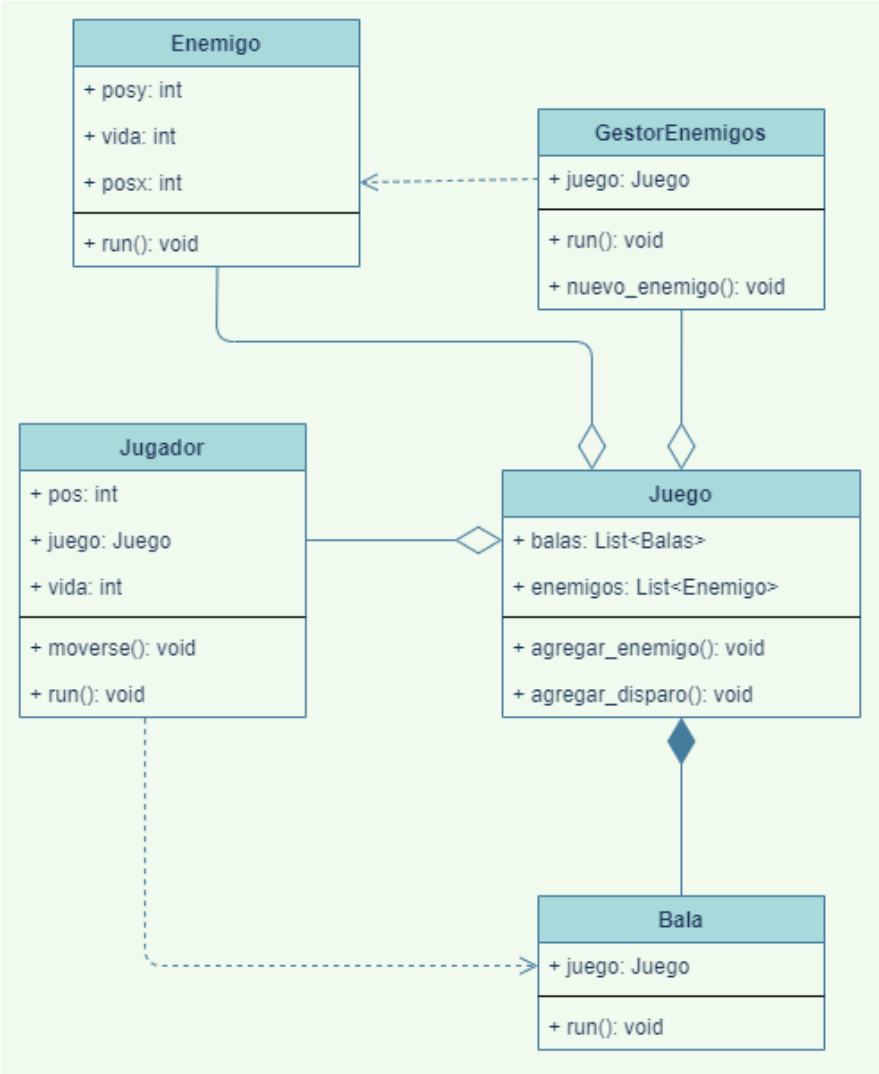


Ilustración 6 Diagrama de clases Space Invaders

Ciclo de vida de los procesos

A continuación, se detallará cada uno de los procesos para la solución de space invaders.

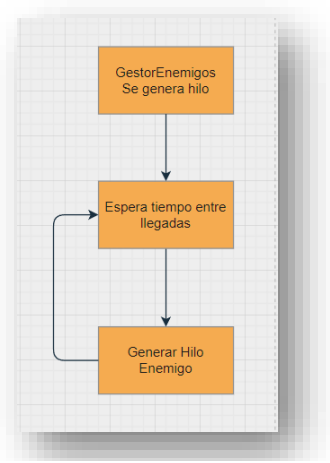


Ilustración 7 Flujo gestor enemigos

Como se puede observar en la ilustración 7 el gestor de enemigos se crea al iniciar el juego y persiste durante toda la vida del juego. Su función será generar los hilos de los enemigos cada cierto tiempo. Y se quedará en este bucle durante su vida.

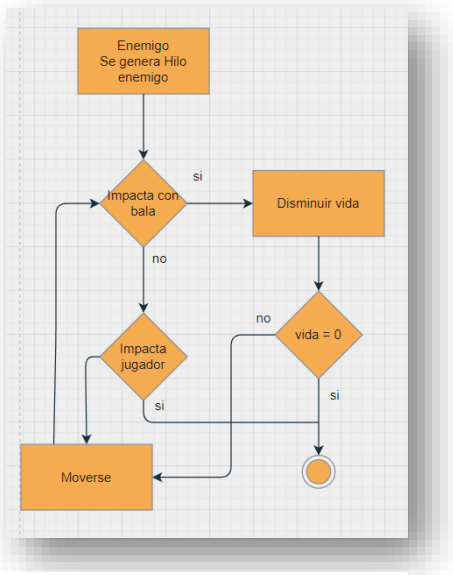


Ilustración 9 Ciclo de vida Enemigo

En la ilustración 8 se muestra el ciclo de vida del hilo Enemigo el cual estará actualizando los recursos del objeto Juego. Se origina por medio del hilo anterior (Gestor de enemigos). Principalmente estará verificando si ha colisionado con una bala o colisionado con un jugador.

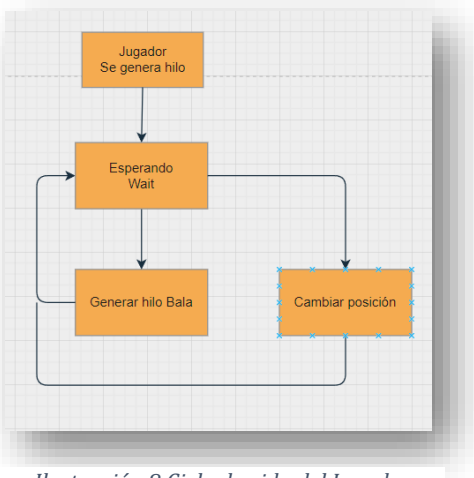


Ilustración 8 Ciclo de vida del Jugador

El jugador será el hilo que se genera al iniciar el juego es instanciarán dos objetos de esta clase. Se estará esperando a que el usuario generé un movimiento o generé un disparo.

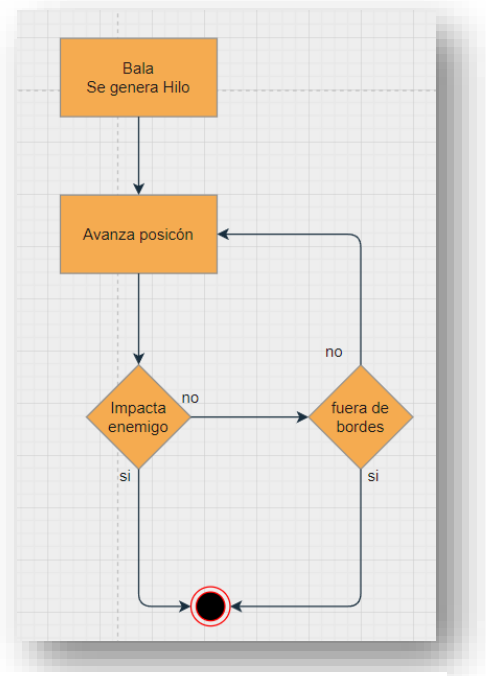


Ilustración 10 Ciclo de vida de la bala

La bala es generada por el hilo del jugador y su ciclo será avanzar en su eje y. Verificando si tiene un impacto con algún enemigo o si esta fuera de los bordes, si sucede alguna de estas acciones entonces terminará.

Problemas de concurrencia

Condiciones de carrera: Este problema ocurre cuando se quiere hacer la verificación en las listas del objeto Juego, existe la posibilidad de querer agregar y de querer eliminar un elemento al mismo tiempo. Por ejemplo, un enemigo nuevo que ingresa a la lista de enemigos y un enemigo que es eliminado de la lista porque según la lógica ha colisionado con una bala.

Inconsistencia de datos: Este problema surge a partir de una mala sincronización de datos. Esto ocurre cuando se desea modificar un valor o objeto dentro de las listas de enemigos disponibles o balas en el juego. Por lo cual se puede hacer una lectura errónea al momento de recorrer cada una de las listas de enemigos o de las balas que disparo cada usuario.

Para solucionar estos problemas se debe de bloquear el recurso entonces se utilizará la función ***synchronized*** para poder eliminar esas inconsistencias, ya que no se podrá acceder al recurso hasta que este sea desbloquedado.