Instituto Tecnológico de Costa Rica.

Área de Ingeniería en Computadores.

Lenguajes Intérpretes y Compiladores.

Profesor: Marco Rivera.

Grupo 02.

Tarea 3: BreakOutTEC.

Estudiantes:

- Santiago Brenes Torres (2019063875)
- Tomás Segura Monge (2018099729)

Documentación.

“Breakout es un videojuego arcade desarrollado por Atari, Inc. y lanzado al mercado el 13 de mayo de 1976. El juego consiste que en la parte inferior de la pantalla una rayita simulaba una raqueta de front-tenis que el jugador podía desplazar de izquierda a derecha.

En la parte superior se situaba una banda conformada por rectángulos que simulaban ser ladrillos. Una pelotita descendía de la nada y el jugador debía golpearla con la raqueta, entonces la pelota ascendía hasta pegar en el muro, los ladrillos tocados por la pelota desaparecían. La pelota volvía a descender y así sucesivamente. El objetivo del juego era terminar con la pared de ladrillos" (Wikipedia, 2020).

Objetivo General

Desarrollar una aplicación que permita reafirmar el conocimiento de los paradigmas de programación imperativo y orientado a objetos.

Objetivos Específicos

- Desarrollar una aplicación en el lenguaje de programación C y java.
- Aplicar los conceptos de programación imperativa y orientada a objetos.
- Crear y manipular listas como estructuras de datos.

A continuación se presenta la documentación externa del proyecto BreakOutTEC que se puede encontrar en GitHub. El esquema del documento es el siguiente:

- 1 - Documentación del código.
    - 1.1 - Manual de usuario: cómo ejecutar el programa.
    - 1.2 - Descripción de las estructuras de datos desarrolladas.
    - 1.3 - Descripción detallada de los algoritmos de solución.
    - 1.4 - Problemas sin solución.
    - 1.5 - Plan de Actividades.
    - 1.6 - Problemas encontrados.
    - 1.7 - Conclusiones y Recomendaciones del proyecto.
- 2 - Bitácoras.
    - 2.1 - Bitácora de reuniones.
    - 2.2 - Bitácora de Santiago.
    - 2.3 - Bitácora de Tomás.
- 3 - Referencias.

1.1 -Manual de usuario: cómo ejecutar el programa.

Hola y bienvenido a Break Out TEC, el juego imposible de sobrevivir en el primer nivel.

Antes de comenzar se debe agregar esta línea en el archivo `/etc/services` :

```
cpp_java 15557/tcp
```

Ahora sí, se puede comenzar con el manual:

1. Se debe descargar el ZIP de la branch master en el repositorio de Git.
2. Extraer el paquete en la carpeta de escogencia.
3. Abrir tu IDE de preferencia pero recomendamos usar IntelliJ ya que no podemos garantizar que trabaje con todos los IDEs de la misma forma.
4. Ejecutar el archivo `App.java` .
5. Otorgar los datos a la consola necesarios para comenzar.
6. Disfrutar del juego.
7. Se utilizan las flechas izquierda y derecha para jugar.

1.2 - Descripción de las estructuras de datos desarrolladas.

Ladrillos (lógica): `wall[rows][columns]` .

Los ladrillos se manejan a través de una matriz `(array[][])` de la siguiente forma:

```
{ {1, 1 ,1, 1, 1, 1, 1, 1, 1}, // rojo
  {1, 1 ,1, 1, 1, 1, 1, 1, 1},
  {2, 2, 2, 2, 2, 2, 2, 2, 2}, // naranja
  {2, 2, 2, 2, 2, 2, 2, 2, 2},
  {3, 3, 3, 3, 3, 3, 3, 3, 3}, // amarillo
  {3, 3, 3, 3, 3, 3, 3, 3, 3},
```

```
{4, 4, 4 ,4 ,4 ,4 ,4 ,4, 4}, // verde
{4, 4, 4 ,4 ,4 ,4 ,4 ,4, 4} }
```

En caso de que se destruya un ladrillo, este pasa a valer `0` de modo que se destruye de la interfaz también.

Ladrillos (interfaz): `wallOfBricks[][]`

Es un reflejo de la matriz lógica pero en vez de almacenar `integer`, guarda objetos `Brick` con sus respectivos atributos. Por ejemplo, un ladrillo podría tener una vida guardada por lo que su variable `life` pasa a ser `true`. Es importante decir que las habilidades o bonus que guarda cada ladrillo, se definen de forma aleatoria y no todos tendrán una habilidad o bonus.

```
{ {Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick},
  {Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick},
  {Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick},
  {Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick},
  {Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick},
  {Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick},
  {Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick},
  {Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick, Brick} }
```

Bolas en juego `balls`

Esta lista, a diferencia de las pasadas hace uso de `ArrayList<Ball>` de Java, principalmente porque tiene la capacidad de ser de tamaño dinámico. Esto es muy benificioso ya que la cantidad de bolas está constantemente variando, además de que las bolas en juego también suelen ser diferentes. Es decir, si tengo una bola en cierto del juego no necesariamente será la misma bola que tenía al inicio aunque la cantidad neta de bolas sea la misma.

Esta lista puede verse de la siguiente manera:

```
{Ball, Ball, ...}
```

Como dije, varía constantemente de cantidad de bolas por lo que la forma de la lista cambia dinámicamente.

Funciones usadas para la parte en C.

```
void compactaClaves (int *tabla, int *n)
```

Busca en array todas las posiciones con -1 y las elimina, copiando encima las posiciones siguientes.

Ejemplo, si la entrada es (3, -1, 2, -1, 4) con *n=5
A la salida tendremos (3, 2, 4) con *n=3

```
int dameMaximo (int *tabla, int n)
```

Función que devuelve el valor máximo en la tabla.

Supone que los valores válidos de la tabla son positivos y mayores que 0.

Devuelve 0 si n es 0 o la tabla es NULL

```
void nuevoCliente (int servidor, int *clientes, int *nClientes, struct datosTableroInicial tab)
```

Crea un nuevo socket cliente.
Se le pasa el socket servidor y el array de clientes, con el número de clientes ya conectados.

```
int main()
```

Programa principal. Crea un socket servidor y se mete en un select() a la espera de clientes. Cuando un cliente se conecta, le atiende y lo añade al select() y vuelta a empezar.

1.3 - Descripción detallada del algoritmo general de solución.

Lógica de la interfaz.

Primero, la clase `GameLoop` se encarga de manejar todo el ciclo del juego del modo que se crea la percepción de que los objetos están animados. El constructor de la clase crea todos

los objetos esenciales y la interfaz.

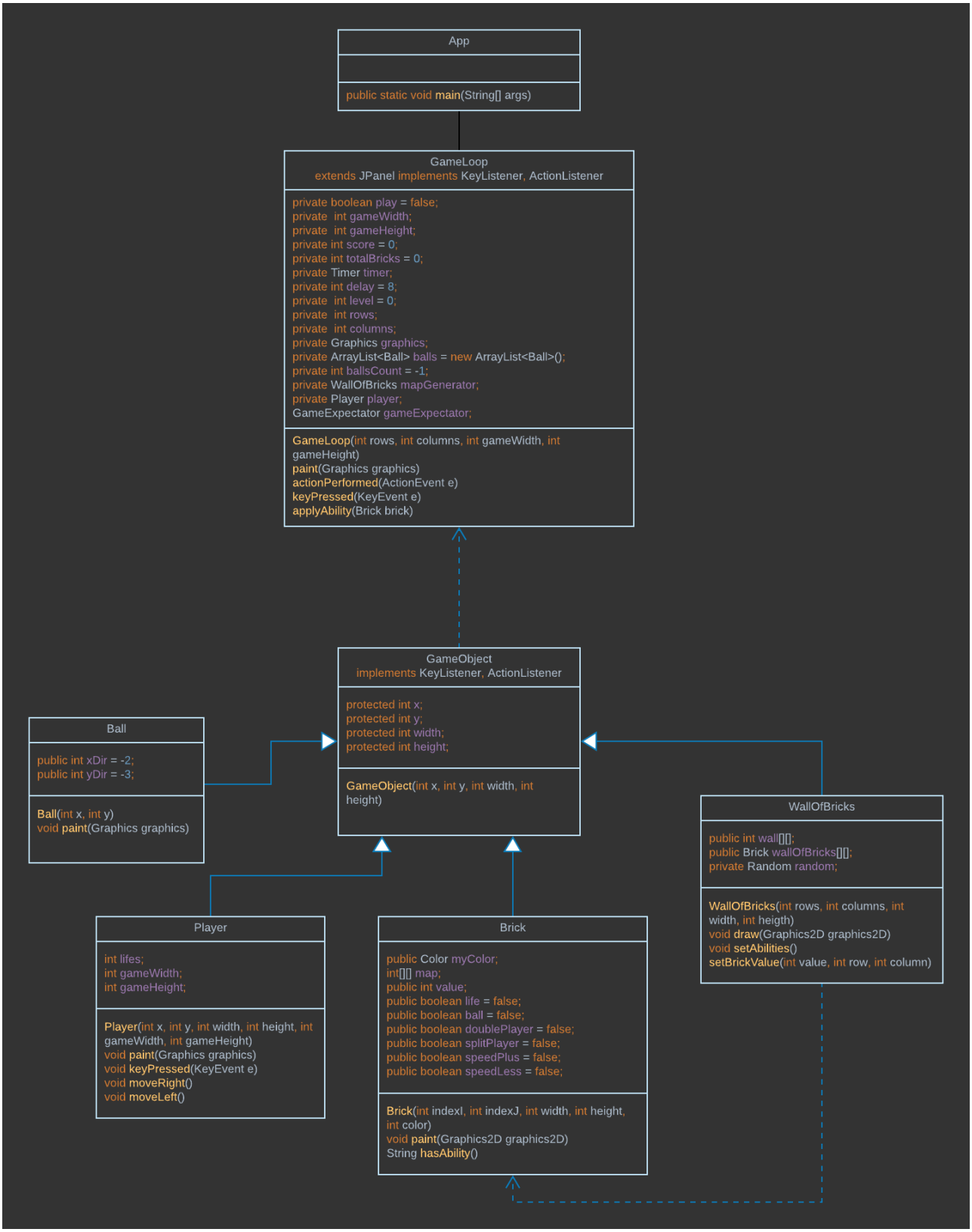
Segundo, se inicia el ciclo de modo que mientras que la variable `play = true` entonces se encarga de actualizar las posición y dirección tanto de las bolas como del jugador.

Tercero, administra los muros de ladrillos de modo que si un ladrillo es impactado, entonces se verifica si tiene habilidad almacenada o no.

Cuarto, si el ladrillo tiene habilidad, entonces la manifiesta en el juego. Si no tiene, solamente destruye el ladrillo y suma el puntaje.

Quinto, si el jugador destruye todos los ladrillos o pierde sus vidas entonces el juego se detiene: `play = false`.

A continuación, un diagrama de clases:



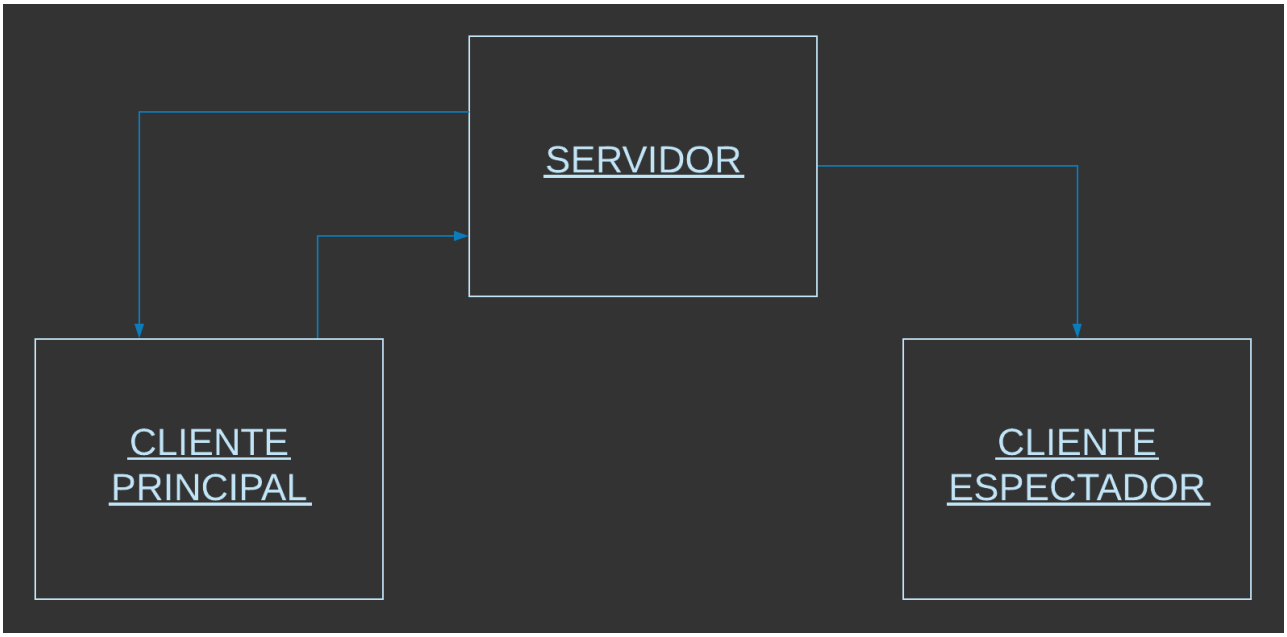
Arquitectura de la aplicación

La aplicación es cliente servidor sencilla. Cuenta con dos clientes:

1. Cliente principal: realiza toda la lógica anteriormente descrita y tiene la capacidad de recibir y enviar datos.
2. Cliente espectador: únicamente tiene la capacidad de recibir datos y no tiene control sobre lo expuesto.

El server se encarga de toda la comunicación y el primer seteo de la aplicación, es decir, las especificaciones de:

- Número de filas y columnas del muro de ladrillos.
- Score de cada color de ladrillos.



1.5 - Plan de Actividades.

Tarea	Descripción	Tiempo estimado	Encargado
Investigar Java	Aprender sobre interfaces para juegos animados en Java.	1 día	Tomás
Investigar C	Aprender sobre sockets y conexión entre C y Java usandolos.	1 día	Santiago

Tarea	Descripción	Tiempo estimado	Encargado
Interfaz BOT	Montar la interfaz/cliente del juego usando Java	2 días	Tomás
Server BOT	Montar el server y los sockets de los clientes de Break Out TEC en C	2 días	Santiago
Conectar	Conexión entre cliente servidor y C / Java	1 día	Tomás y Santiago.

1.6 - Problemas solucionado y sin solución.

Patrones de diseño.

Nos habría gusta implementar más patrones de diseño pero por motivos de tiempo, no fue posible. Aplicamos un singleton para la creación del GameLoop de modo que solamente pueda existir una instancia de él a la vez.

Igualmente, dejo un par de ideas que teníamos para implementar:

- Factory para la creación de los ladrillos.
- Observer para el servidor y los clientes.
- Facade para los clientes, de modo que una sola central manejara todo de una manera muy limpia.

Reflexión del cliente espectador.

El cliente fue un reto interesante ya que al inicio creímos que tendríamos que pasar muchos datos y muy complicados (como la matriz de los ladrillos). Tras sentarnos a pensar un momento, nos dimos cuenta que no hacía falta ya que ese cliente únicamente debía “mostrar” y no hacer lógica.

De ese modo, redujimos a unas simples variables las que se debían enviar por estos sockets.

Finalmente, enviamos la posición y ancho de la paleta del jugador, la posición y dirección de la bola. La única lógica que hace este cliente es la de destruir ladrillos.

Conectar Java con C.

Conectar los sockets resultó más complicado de lo que esperábamos. Debimos actualizar el código para soportar varios clientes y constante comunicación.

1.7 - Conclusiones y Recomendaciones del proyecto.

Conclusiones.

1. Dejar los patrones de diseño para el final implica que se debe reescribir el código para poder implementar un patrón por lo que quita mucho tiempo.
2. Las conexiones cliente/servidor deben ser analizadas con cuidado de modo que los mensajes no sean demasiado complicados y no se vea afectado la eficiencia del sistema.
3. Conectar dos lenguajes implica muchos retos, usar sockets simplifica el proceso ya que un string o un buffer es algo que comparten muchos lenguajes entre sí.

Recomendaciones.

1. Aplicar los patrones de diseño desde que se comienza a escribir el código por primera vez. Esto se puede lograr con un buen planeamiento del proyecto.
2. Continuar analizando los problemas y hacer lluvias de ideas antes de lanzarse a programar. Resultó muy bien para el reflejo del clientes espectador.
3. Conectar dos lenguajes usando tecnologias que tengan en común o algún otra que pueda servir como un puente. Por ejemplo, un rest API.

2 - Bitácoras.

2.1 - Bitácora de reuniones.

`22 de octubre`

Nos organizamos. Decidimos que yo me encargaría de Java y la interfaz y Santiago de C y el server.### 2.2 - Bitácora de Santiago.

`26 de octubre`

Casi no nos hemos reunido ya que el trabajo ha sido bastante individual pero en estos días estaremos trabajando en la conexión de los datos de los clientes con el server.

2.2 - Bitácora de Santiago.

`22 de octubre`

Nos organizamos. Decidimos que yo me encargaría de Java y la interfaz y Santiago de C y el server.

`24 de octubre`

He estado leyendo sobre la conexión de Java con C usando el link del profesor.

`26 de octubre`

El servidor me ha dado muchos problemas. Debo actualizar el código para que soporte varios clientes y comunicación bilateral.

`30 de octubre`

Dando toques finales, haciendo la docu e intentando aplicar algun patrón de diseño.

2.3 - Bitácora de Tomás.

`22 de octubre`

Nos organizamos. Decidimos que yo me encargaría de Java y la interfaz y Santiago de C y el server.

`23 de octubre`

Vi un tutorial para hacer un Break out en java usando swing. Me pareció sencillo pero se debe mejorar el código para implementar más objetos.

`25 de octubre`

Implementé los objetos adicionales, también herencia para que cumpliera con las

características de OO.

`30 de octubre`

Dando toques finales, haciendo la docu e intentando aplicar algun patrón de diseño.

3 - Referencias.

chuidiang.org. (2020, 3 21). Socket entre C y java. Retrieved from Socket entre C y java: http://www.chuidiang.org/java/sockets/cpp_java/cpp_java.php

Wikipedia. (2020, 10 10). BreakOut (videojuego). Retrieved from https://es.wikipedia.org/wiki/Breakout_(videojuego)