Tarea 2 - Breakout

Profesor: Alexandre Bergel
Auxiliar: Juan-Pablo Silva

Breakout

Breakout es un juego de *arcade* que consiste en utilizar una barra horizontal y una bola que rebota en las paredes de la pantalla, para golpear ladrillos posicionados en la parte superior de la pantalla. Cada vez que la bola choca con un ladrillo, esta rebota y puede destruirlo, atribuyendo un puntaje al jugador acorde a la dificultad de destruir dicho ladrillo. Si la bola toca la parte inferior de la pantalla, esta se considera una bola perdida y el jugador pierde una vida. Durante esta tarea 2 y siguiente tarea 3 implementaremos este juego.

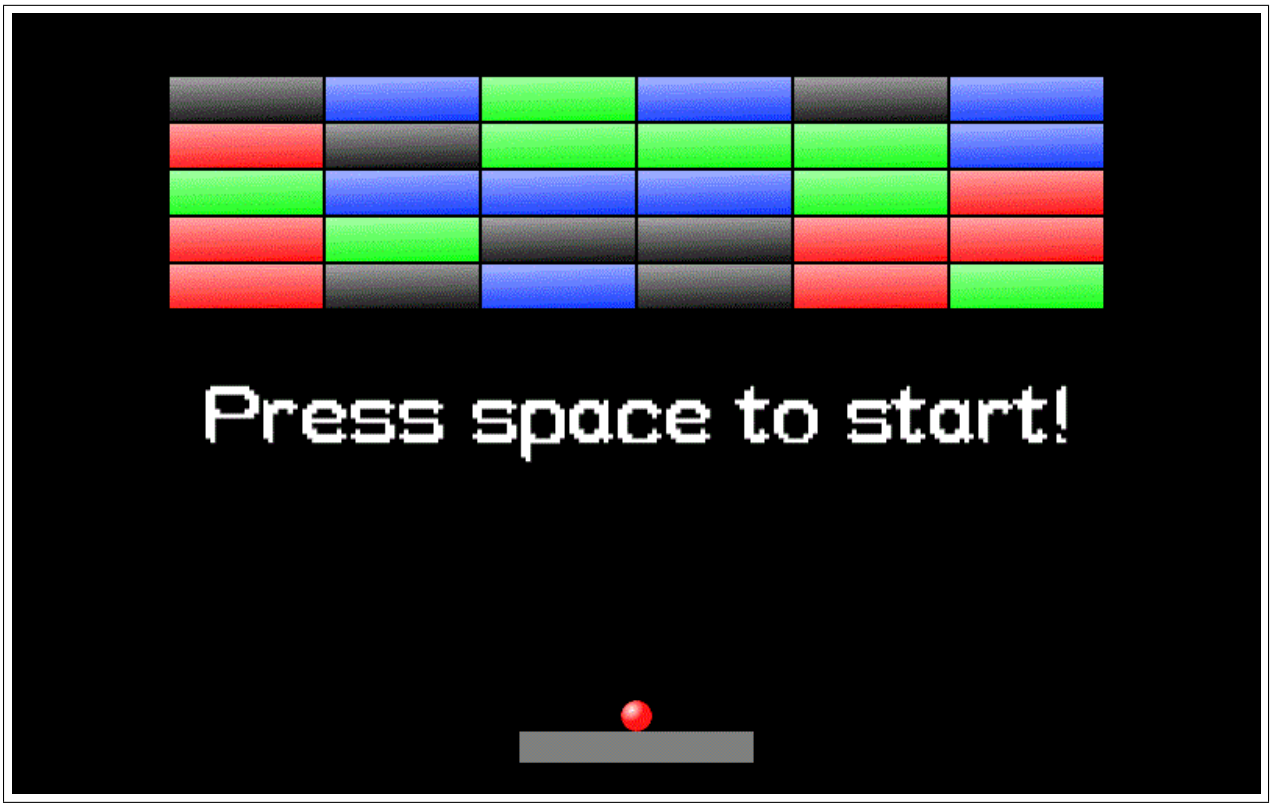

Figura 1: Ejemplo de un juego de Breakout.

El objetivo básico del juego es usar la bola para golpear *Brick*s y ganar puntos para pasar al

siguiente nivel. Los *Brick*s tienen la característica de que cuando la bola choca con ellos, esta rebota y, dependiendo del tipo de *Brick*, este puede ser destruido, y el jugador ganar puntos de ello. Una vez se hayan destruido todos los *Brick*s de una nivel, el juego continúa al nivel siguiente de forma automática. Cuando un jugador pierde todas las bolitas, este pierde el juego.

Requisitos

Adjunto a este enunciado se encuentra el código que usted debe completar e implementar. Para esta tarea 2 buscamos implementar la lógica y controlador lógico del juego, basado en las especificaciones que se darán a continuación.

En el paquete `facade` se encuentran las las abstracciones de las funcionalidades que usted debe implementar usando el patrón de diseño *facade*. Descripciones sobre cada método se encuentran en la misma clase. El objetivo de este patrón es crear un objeto que exponga y represente al proyecto, en forma de API, para luego se usar este objeto al conectar y crear la interfaz gráfica del juego para la tarea 3. Se dará una descripción de para qué sirve este patrón y cómo usarlo.

A continuación se explican las interfaces y los objetos que deben crear. Explicaciones sobre cada método están en el mismo código.

Brick

Esta interfaz representa a un ladrillo y se encuentra en el paquete `logic.brick`. En el juego existen 3 tipos de ladrillos. Los ladrillos implementan métodos tales como `hit`, para simular un golpe, y `getScore`, para saber cuantos puntos se obtienen al destruirlo. Además se tienen métodos de utilidad como `isDestroyed` y `remainingHits`, para saber si el `Brick` se encuentra destruido o no, y para saber el número de golpes restantes antes de que se destruya, respectivamente. Cada vez que la bola choca con estos objetos, el ladrillo disminuye en uno la cantidad de golpes requeridos para que sea destruido, y una vez se acaban estos puntos, este se destruye y entrega puntaje al jugador. Los tipos de `Brick` que incluiremos en el juego son los siguientes:

- **GlassBrick:** necesita de **1 golpe** para que se destruya y entrega **50 puntos** al hacerlo.
- **WoodenBrick:** necesita de **3 golpes** para que se destruya y entrega **200 puntos** al hacerlo.
- **MetalBrick:** necesita de **10 golpes** para que se destruya y entrega **0 puntos** al hacerlo. Este `Brick` tiene la característica de que cuando es destruido el jugador gana una bola extra. Lo interesante es que no es necesario destruir los `MetalBrick`s para pasar de nivel, ya que no entregan puntaje al romperlos.

Level

Level es una interfaz que se encuentra en el paquete `logic.level` que representa la abstracción de un nivel de juego. Contiene varios métodos para obtener información sobre este y agregar nuevos niveles a la lista de juegos. Una vez se comienza un juego, para poder pasar automáticamente al siguiente nivel, es necesario que cada nivel también sepa que nivel viene. Cuando se acaban los niveles, el juego termina y el juegador gana.

Cada nivel tiene un nombre y una lista de `Bricks`, además de una referencia al siguiente nivel. Un `Level` contiene métodos *getters* y *setters* como los que se muestran en la interfaz en el código adjunto, pero puede agregar más en caso de que lo requiera. También se tienen métodos de utilidad para el testeo de su código como `isPlayableLevel` y `hasNextLevel`. El método `addPlayingLevel` tiene la característica de que agrega un nivel a la lista, pero en la última posición, en vez de la primera, permitiendo que se siga jugando el nivel actual y aún así poder agregar otro.

El juego por defecto se inicializa con un nivel inválido, sin objetos dentro de él, es simplemente un *placeholder*, por lo que el método `isPlayableLevel` responde a si el nivel es jugable o no. Descripciones más específicas sobre cada método en la interfaz `Level` se encuentran en el código adjunto.

Game

Corresponde al controlador del juego. Esta clase se le entrega vacía y se espera que usted implemente toda la funcionalidad general de juego aquí, y luego complete la clase `HomeworkTwoFacade` para exponer una interfaz simplificada para la interfaz gráfica que haremos en la siguiente tarea. Tiene total libertad sobre cómo implementar esta clase y los métodos que contenga (puede poner cuantas variables y métodos quiera, con los nombres que quiera), siempre y cuando mantengan un buen diseño final y responda a la funcionalidad deseada en la clase `HomeworkTwoFacade`.

El juego, como es el controlador, también es quien pasa los niveles cuando corresponda. Tenga especial atención que un nivel se completa, termina, gana, etc, cuando se alcanza el máximo número de puntos obtenibles en ese nivel, no necesariamente cuando se destruyan todos los `Bricks`, ya que los `MetalBricks` deben ser considerados como una especie de *bonus* no requerida para ganar el nivel.

Facade

La clase `HomeworkTwoFacade` corresponde a un patrón de diseño llamado *facade*, que en español se traduce como *fachada*. Lo que intenta solucionar este patrón es, en vez de mostrar toda la complejidad que tiene `Game` y las otras clases del juego a un cliente, usuario, o interfaz gráfica, esta clase expone (muestra) una simplificación, con nombres de métodos claros, documentación, etc,

del programa en sí. La funcionalidad expuesta por el *facade* responde a todo lo que el programa puede hacer y a todas las configuraciones necesarias, por lo que con solo un nivel de indirección, escondemos toda la complejidad de nuestra aplicación y mostramos una interfaz limpia y clara de lo que puede hacer nuestro programa.

Los cuerpos de los métodos que se encuentran en un *facade* son, en general, de una línea ya que la lógica se encuentra en otro lado. La mayoría de los métodos que se encuentran en `HomeworkTwoFacade` se solucionan haciendo una llamada con el mismo nombre a la clase `Game`. Usted podría solucionarlo de otra forma, pero recuerde que no hay lógica en el *facade*, solo llamadas a otros métodos que pueden estar enlazados de la forma `objeto.metodo1().metodo2().metodo3()` en caso de que los métodos retornen otros objetos. Se le entrega el cuerpo del método `winner` como ejemplo de lo que se espera que complete en esta clase. Para testear el funcionamiento de la tarea se utilizará esta clase, ya que es la interfaz expuesta estandar que usaremos.

Los métodos `newLevelWithBricksFull` y `newLevelWithBricksNoMetal` contienen como último parámetro `int seed`. Un *seed* o semilla corresponde a un número del cual se parted generando número pseude-aleatórios. Para generar los `Brick`s, tendrán que utilizar la clase `Random`, que genera números pseudo-aleatórios. Para poder testear la funcionalidad, y que los tests sean replicables, es necesario que estos números "random" sean siempre los mismos. De hecho, la forma en que normalmente funciona esta clase es asignando siempre una semilla distinta, utilizando funciones como `System.currentTimeMillis()`, que no importa el momento que se llame, debería ser distinta.

Testing de funcionalidad

Después de publicada la tarea, se publicará el conjunto de pruebas a las que se someterá su tarea para verificar su funcionamiento. No está mal mirar las pruebas para ver en dónde puede estar fallando su tarea, pero en caso de que alguien implemente su código para solo pasar los tests (por ejemplo un test que pida el puntaje obtenido hasta ese momento, digamos 100, y el código implementado por el alumno retorna 100 de manera directa) será evaluado de forma muy negativa, asignandose un gran descuento.

Cómo se juega y aclaraciones

Un juego comienza con un nivel vacío o inválido, para crear niveles se utilizan los métodos `newLevelWithBricksFull` y `newLevelWithBricksNoMetal`, y para comenzar a jugar un juego oficial se requiere llamar el método `setCurrentLevel` con el nivel creado por alguno de los métodos anteriores. Luego para agregar niveles al juego, se utiliza el método `addPlayingLevel`.

La creación de un nivel recibe los siguientes parámetros: `(String name, int numberOfBricks, double probOfGlass, double probOfMetal, int seed)`. Estos corresponden al nombre del nivel, el número total de `Brick`s, la probabilidad de que un `Brick` sea un `GlassBrick` (`probOfGlass`

probabilidad de ser `GlassBrick`, y por lo tanto 1 − `probOfGlass` de ser `WoodenBrick`), la probabilidad `probOfMetal` de que aparezca un `MetalBrick` y la semilla `seed` explicada en una sección anterior. El número indicado en `numberOfBricks` corresponde solo a la suma entre `GlassBrick`s y `WoodenBrick`s, los `MetalBrick`s no entran en este número, por lo que finalmente existe la posibilidad de que hayan más ladrillos que los originalmente indicados, pero para pasar de nivel solo se necesitan destruir `numberOfBricks` ya que son estos los que asignan puntaje.

Para jugar en esta tarea, todo debe hacerse de manera un poco manual. Cuando los tests sean publicados usted podrá ver que se crea un nivel, se asigna al juego, y luego para golpear cada uno de los ladrillos, se le pide el nivel al juego, luego los ladrillos al nivel y se les envía el mensaje `hit` a cada uno de ellos, destruyendolos eventualmente y finalmente obteniendo puntaje a partir de ello.

El juego comienza con 3 bolitas como se muestra en el construtor de `Game` en la clase `HomeworkTwoFacade` y el juego tiene 0 puntos acumulados inicialmente. A medida que se van destruyendo ladrillos, el puntaje acumulado aumenta, manteniendose de nivel en nivel y acumulandose progresivamente.

Recuerde que el juego se gana solo cuando se termina el último nivel en la lista de niveles. Si se pierden las 3 bolitas el juego se pierde.

Usted tiene total libertad de introducir cuantas clases e interfaces quiera, sin embargo la estructura que se le provee debe mantenerse, para tener una estructura definida sobre la cual podamos probar. Usted es libre de agregar métodos a las interfaces `Brick` y `Level`, implementarlas las veces que necesite, hacer que las interfaces extiendan de otras interfaces, etc, pero recuerde que esos métodos no serán considerados "estándar" y no serán usados por nosotros (ustedes pueden usarlos para lógica interna de su tarea). La clase `Game` tiene libre implementación. No puede modificar la clase `HomeworkTwoFacade`, solo completar el cuerpo de los métodos pedidos, agregar un constructor, y agregar variables, *fields*, si lo necesita.

Como puede ver, esta tarea no contiene parte gráfica ya que esta será desarrollada en la tarea 3 usando como base el código que implemente en esta tarea. Por lo mismo es muy importante que la haga a consciencia.

Requerimientos adicionales

Además de una implementación basada en las buenas prácticas y patrones de diseño vistos en clases, usted además debe considerar:

- **Cobertura:** Cree los tests unitarios, usando JUnit 4, que sean necesarios para tener al menos un coverage del 90 % de las líneas por paquete. Todos los tests de su proyecto deben estar dentro del paquete `test.java` del proyecto *Maven*. Sus tests deben verificar funcionalidad y

testear casos de borde que encuentre, no sirve que solo pruebe los *getters* y *setters* y pase por sobre los demás métodos para ganar líneas recorridas.

- **Javadoc:** Cada interfaz, clase pública y método público deben estar debidamente documentados siguiendo las convenciones de Javadoc[1]. En particular necesita `@author` y una pequeña descripción para su clase e interfaz, y `@param`, `@return` (si aplica) y una descripción para los métodos. Cláramente la documentación entregada en el código base no cuenta como documentación para usted, por lo que debe documentar sus implementaciones concretas.
- **Resumen:** Debe entregar un archivo **pdf** que contenga su nombre, rut, su usuario de Github, un link a su repositorio y un diagrama UML que muestre las clases, interfaces y métodos que usted definió en su proyecto. **No debe incluir los tests** en su diagrama.
- **Git:** Debe hacer uso de git para el versionamiento de su proyecto. Luego, esta historia de versionamiento debe ser subida a Github. Además deberá crear un *tag* "tarea2" en el último commit de su tarea, cómo hacer esto se encuentra en los detalles de entrega. Esto es un requerimiento ya que utilizará el mismo repositorio para la tarea 3.
- **Readme:** En la raíz de su proyecto debe crear un archivo `README.md` con una descripción de su implementación, cómo compilarlo (cómo se usa) y dónde se encuentra cada módulo. Más instrucciones en los detalles de entrega.
- **Maven:** Su implementación debe estar basada en un proyecto *Maven*, dado como código adjunto a este enunciado. Esto nos facilitará la importación de librerías para la tarea 3.

Evaluación

- **Código fuente (4 puntos):** Se analizará que su código provea la funcionalidad pedida utilizando un buen diseño basado en las buenas prácticas.
- **Coverage (1 punto):** Sus casos de prueba deben crear diversos escenarios y contar con un coverage de al menos 90 % por paquete. No está de más decir que sus tests deben testear algo (es decir, no ser tests vacíos o sin asserts).
- **Javadoc (0.5 puntos):** Cada clase, interfaz y método público debe ser debidamente documentado. Se descontará por cada falta.
- **Resumen y readme (0.5 puntos):** El resumen y readme mencionado en la sección anterior. Se evaluará que haya incluido el diagrama UML de su proyecto. **En caso de no enviarse el resumen con el link a su repositorio, su tarea no será corregida**[2]. Se le hará un

[1] `http://www.oracle.com/technetwork/articles/java/index-137868.html`
[2] porque no tenemos su código.

descuento de 0.5 puntos en código fuente si no incluye un readme, asumiendose que no utilizó patrones de diseño ni buenas prácticas de programación.

Entrega

Repositorio: Su repositorio de Github debe ser **privado**, y debe llamarse **cc3002-breakout**, lo cual significa que si su usuario de Github es *juanpablos*, el link a su repositorio será `https://github.com/juanpablos/cc3002-breakout`. Además, deberán invitar a la cuenta del equipo docente (*CC3002EquipoDocente*) como colaboradores de su repositorio para que podamos acceder a él. Para hacer esto entran a su repositorio, luego a *Settings*, *Collaborators*, ingresan su contraseña y finalmente escriben *CC3002EquipoDocente* en el campo para buscar por usuario, seleccionan *Add collaborator* y nos habrán invitado exitosamente. No se asusten si su invitación no es aceptada, estas serán aceptadas oportunamente cuando su tarea sea revisada.

Entrega por U-Cursos: Usted debe subir a U-Cursos **solamente el resumen en pdf**, con su nombre, rut, usuario de Github, link a su repositorio y diagrama UML. Si su diagrama UML no cabe en el resumen, inclúyalo de todas formas y añada una imagen de él en su repositorio.

Código: El código de su tarea será bajado de Github directamente considerando como entrega el commit que contenga el *tag* "tarea2". Para hacer un *tag* abra su consola y tipee `git tag -a tarea2 -m "Este es el último commit de mi tarea 2 (ponga un comentario distinto a este por favor)"`, el comentario queda a su elección. Esto pondrá el *tag* "tarea2" en el último commit que haya hecho. Para subir su anotación tipee `git push tarea2`, o también `git push --tags`.

Readme: Dentro de su repositorio debe incluir un archivo de *markdown* llamado `README.md` que está incluido en el código adjunto. Usted debe dar una descripción de lo que implementó, cómo lo hizo, a grandes rasgos qué patrones de diseño usó, cómo correr su programa, y otras anotaciones que usted encuentre pertinentes. Aquí puede encontrar un *cheatsheet*[3] y un *template*[4]. El template incluye mucha más información que la solicitada, pero así es como debe presentarse un proyecto profesionalmente, puede inspirarse en él para agregar secciones al suyo.

Plazo: El plazo de entrega es hasta el jueves 22 de noviembre a las 23:59 hrs. Se bajará su código hasta su *tag* "tarea2" y se verificará que efectivamente la anotación "tarea2" se haya hecho antes de la fecha límite. No se aceptarán peticiones de extensión de plazo.

[3] `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`
[4] `https://gist.github.com/PurpleBooth/109311bb0361f32d87a2`

Recomendaciones

No comience su tarea a último momento. Esto es lo que se dice en todos los cursos, pero es particularmente importante en este. Si usted hace la tarea a último minuto lo más seguro es que no tenga tiempo para reflexionar sobre su diseño y termine entregando un diseño defectuoso o sin usar lo enseñado en el curso. Además esta tarea 2 es **incremental**, que significa que para la siguiente tarea 3 usted deberá usar el código que implemente en esta tarea como base.

Haga la documentación de su programa en inglés (no es necesario). La documentación de casi cualquier programa open-source se encuentra en este idioma. Considere esta oportunidad para practicar su inglés.

Les pedimos encarecidamente que las consultas referentes a la tarea las hagan por el **foro de U-Cursos**. En caso de no obtener respuesta en un tiempo razonable, pueden hacer llegar un correo al auxiliar.

Si tienen dudas, pregunten. Es importante que resuelvan sus dudas a tiempo para que no atrasen su diseño y/o implementación. Conversar con sus compañeros sobre la tarea está completamente permitido mientras no tomen código de tareas que no son suyas. Las reglas de colaboración las pueden encontrar en el siguiente post: `https://www.u-cursos.cl/ingenieria/2018/2/CC3002/1/foro/o/23192245`.

¡Suerte!