Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC3745 - Testing

Tarea 1

Documentación y Análisis

Javier López A.

Lunes, 25 de septiembre de 2017

Documentación

En esta sección se presentarán los tests realizados, explicando el propósito de cada uno y los errores que ayudaron a encontrar.

NOTA: para ejecutar correctamente estos tests se deben seguir los siguientes pasos.

1. En el archivo `HighLifeBoard.java` agregar al comienzo del archivo la siguiente línea:

   ```
   package src;
   ```

2. En la terminal ejecutar lo siguiente:

   ```
   javac TestRunner.java
   java TestRunner
   ```

Test: `Test1`

El propósito de este test es probar que el método `setCell` efectivamente setee el valor de una celda del tablero a `true` cuando se llame.

En `setUp` se instanció un `HighLifeBoard(10, 10, false)`. La idea de usar este tablero inicial es comprobar que las celdas a testear cambien sus estados de `false` a `true`.

Para este test se usaron los siguientes inputs:

- Coordenadas de celdas en posiciones internas del tablero (excluyendo bordes).
  - `(3, 7, true)`
  - `(8, 4, true)`
  - `(2, 2, true)`

- Coordenadas de celdas en esquinas y bordes del tablero.
    - `(0, 5, true)`
    - `(5, 0, true)`
    - `(9, 5, true)`
    - `(5, 9, true)`
    - `(0, 0, true)`
    - `(0, 9, true)`
    - `(9, 0, true)`
    - `(9, 9, true)`

- Coordenadas de celdas fuera del tablero.
    - `(-1, -1, true)`
    - `(10, 10, true)`
    - `(-1, 10, true)`
    - `(10, -1, true)`
    - `(5, -1, true)`
    - `(-1, 5, true)`
    - `(5, 10, true)`
    - `(10, 5, true)`

Se consideraron inputs donde se cambie el valor original de la celda (`false`) a `true`, para probar que efectivamente se cambie.

Errores encontrados

➔ En el método `setCell`, para setear el nuevo valor a las coordenadas `(i, j)`, como última instrucción se debe realizar la asignación al elemento del arreglo `board[i][j]`, pero erróneamente esta asignación no se realiza.
Este error se evidencia cuando el test retorna que el valor obtenido en la celda `(3, 7)` es `false`, pero se esperaba que fuera `true`. Ocurre de manera equivalente en los demás casos de input.

ERROR	ARREGLADO
`// No se realiza asignación`	`this.board[i][j] = value;`

➔ En el método `setCell`, cuando para un par de coordenadas `(i, j)` ocurre que `j` es mayor o igual al `width` del tablero, entonces `j` se debe setear al valor de `width - 1` (borde del tablero). Erróneamente esta reasignación se hace a la variable `i`.
Este error se evidencia cuando el test retorna que se lanzó una excepción del tipo `ArrayIndexOutOfBoundsException` con las coordenadas `(5, 10)`.

ERROR	ARREGLADO
`else if(j >= this.width)` `  i = this.width - 1;`	`else if(j >= this.width)` `  j = this.width - 1;`

Test: `Test2`

El propósito de este test es probar que el método `isAlive` efectivamente retorne `true` si la celda en las coordenadas entregadas está viva, y `false` en cualquier otro caso (muerta o fuera del tablero).

En `setUp` se instanció un `HighLifeBoard(10, 10, false)` y se utilizó el método `setCell` sin errores (arreglado) para setear en `true` las siguientes coordenadas:

- Coordenadas de celdas en posiciones internas del tablero (excluyendo bordes).
  - `(3, 7, true)`
  - `(8, 4, true)`
  - `(2, 2, true)`
- Coordenadas de celdas en esquinas y bordes del tablero.
  - `(0, 5, true)`
  - `(5, 0, true)`
  - `(9, 5, true)`
  - `(5, 9, true)`
  - `(0, 0, true)`
  - `(0, 9, true)`
  - `(9, 0, true)`
  - `(9, 9, true)`
- Coordenadas de celdas fuera del tablero.
  - `(-1, -1, true)`
  - `(10, 10, true)`
  - `(-1, 10, true)`
  - `(10, -1, true)`
  - `(5, -1, true)`
  - `(-1, 5, true)`
  - `(5, 10, true)`
  - `(10, 5, true)`

La idea de usar este tablero inicial es cubrir todos los casos posibles de coordenadas en el tablero: esquinas, bordes, posiciones interiores y posiciones exteriores.

Para este test se usaron los siguientes inputs:

- Coordenadas de celdas en posiciones internas del tablero (excluyendo bordes).
  - `(3, 7)`
  - `(8, 4)`
  - `(2, 2)`
  - `(7, 3)`
  - `(4, 8)`
  - `(5, 5)`

- Coordenadas de celdas en esquinas y bordes del tablero.
  - `(0, 5)`
  - `(5, 0)`
  - `(9, 5)`
  - `(5, 9)`
  - `(0, 0)`
  - `(0, 9)`
  - `(9, 0)`
  - `(9, 9)`
- Coordenadas de celdas fuera del tablero.
  - `(-1, -1)`
  - `(10, 10)`
  - `(-1, 10)`
  - `(10, -1)`
  - `(5, -1)`
  - `(-1, 5)`
  - `(5, 10)`
  - `(10, 5)`

Errores encontrados

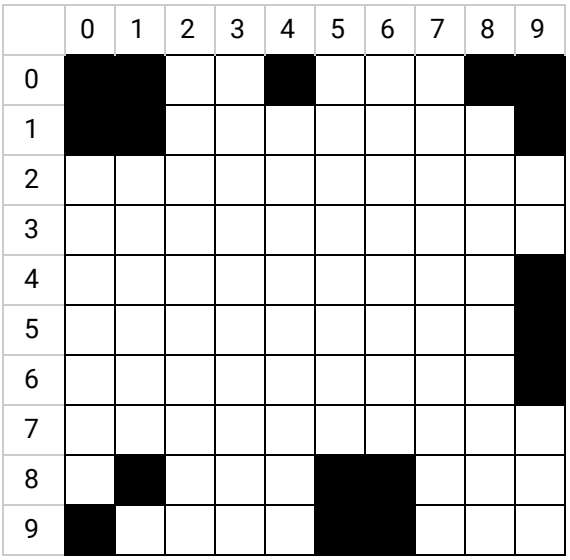
➔ En el método `isAlive`, si alguna de las coordenadas `i` o `j` escapa de las dimensiones del tablero, entonces se debe retornar `false`, ya que toda coordenada fuera del tablero está muerta por defecto. Erróneamente ocurre que cuando `j` escapa de las dimensiones del tablero se retorna `true`, cuando debería retornar `false`.
Este error se evidencia cuando el test retorna que el valor obtenido en la celda `(5, -1)` es `true`, pero se esperaba que fuera `false`.

ERROR	ARREGLADO
`else if (j < 0 || j >= width)` `  return true;`	`else if (j < 0 || j >= width)` `  return false;`

Test: `Test3`

El propósito de este test es probar que el método `countAliveNeighbors` efectivamente retorne el número de vecinos vivos de la celda ubicada en las coordenadas pasadas como parámetros.

En `setUp` se instanció un `HighLifeBoard(10, 10, false)` y se utilizó el método `setCell` sin errores (arreglado) para setear en `true` las coordenadas que se presentan gráficamente en el siguiente tablero para facilitar el entendimiento y visualización (en el código fuente del test se puede revisar la llamada a este método con estas coordenadas).



Para este test se usaron los siguientes inputs:

- Internos.
  - `(5, 5)`
- Esquinas.
  - `(0, 0)`
  - `(9, 0)`
  - `(0, 9)`
  - `(9, 9)`
- Bordes.
  - `(5, 0)`
  - `(9, 5)`
  - `(5, 9)`
  - `(0, 5)`

Errores encontrados

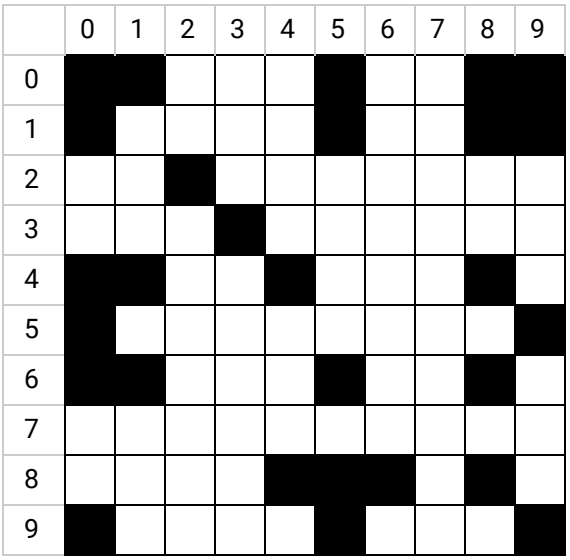
➔ En el método `countAliveNeighbors` se cuenta la cantidad de vecinos vivos que tiene una celda. Los vecinos de una celda se definen como aquellas celdas que comparten bordes o esquinas con ellas, excluyendo a la misma celda. Erróneamente el método cuenta a la misma celda como un vecino.
Este error se evidencia cuando el test retorna que el valor obtenido en la celda `(0, 0)` es `4`, pero se esperaba que fuera `3`.

ERROR	ARREGLADO
`total += this.isAlive(i, j) ? 1 : 0;`	`// Se elimina esta línea`

Test: `Test4`

El propósito de este test es probar que el método `shouldSurvive` efectivamente retorne `true` cuando una celda cumple con las condiciones para sobrevivir. El método asume correctamente que la celda está viva actualmente.

En `setUp` se instanció un `HighLifeBoard(10, 10, false)` y se utilizó el método `setCell` sin errores (arreglado) para setear en `true` las coordenadas que se presentan gráficamente en el siguiente tablero para facilitar el entendimiento y visualización (en el código fuente del test se puede revisar la llamada a este método con estas coordenadas).



Para este test se usaron los siguientes inputs:

- Internos.
  - `(3, 3)`
  - `(5, 6)`
- Esquinas.
  - `(0, 0)`
  - `(9, 0)`
  - `(0, 9)`
  - `(9, 9)`
- Bordes.
  - `(5, 0)`
  - `(9, 5)`
  - `(5, 9)`
  - `(0, 5)`

La idea de estos casos de inputs es cubrir todas los tipos de celdas existentes: celdas interiores, celdas en bordes y celdas en esquinas.

Errores encontrados

➔ En el método `shouldSurvive` se verifica si la celda debe sobrevivir para la siguiente iteración. El método asume correctamente que la celda está viva. Para que una celda sobreviva se debe cumplir que tenga 2 o 3 vecinos vivos. Erróneamente el método chequea que la celda tenga 2 y 3 vecinos vivos al mismo tiempo, lo que es imposible.
Este error se evidencia cuando el test retorna que el valor obtenido en la celda `(3, 3)` es `false`, pero se esperaba que fuera `true`.

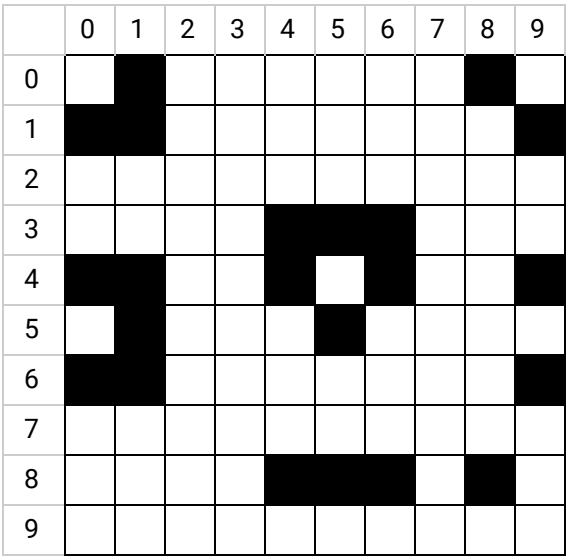
ERROR	ARREGLADO

`if(numAliveNeighbors == 2 &&`
`  numAliveNeighbors == 3)`
`  return true;`
	`if(numAliveNeighbors == 2 ||`
	`  numAliveNeighbors == 3)`
	`  return true;`

Test: `Test5`

El propósito de este test es probar que el método `shouldBeBorn` efectivamente retorne `true` cuando una celda cumple con las condiciones para nacer. El método asume correctamente que la celda está muerta actualmente.

En `setUp` se instanció un `HighLifeBoard(10, 10, false)` y se utilizó el método `setCell` sin errores (arreglado) para setear en `true` las coordenadas que se presentan gráficamente en el siguiente tablero para facilitar el entendimiento y visualización (en el código fuente del test se puede revisar la llamada a este método con estas coordenadas).



Para este test se usaron los siguientes inputs:

- Internos.
  - `(5, 1)`
  - `(5, 4)`
- Esquinas.
  - `(0, 0)`
  - `(9, 0)`
  - `(0, 9)`
  - `(9, 9)`
- Bordes.
  - `(5, 0)`
  - `(9, 5)`
  - `(5, 9)`
  - `(0, 5)`

Se utilizan estos conjuntos de input para cubrir todos los posibles tipos de celdas: celdas internas, celdas en bordes y celdas en esquinas del tablero.

Errores encontrados

➔ En el método `shouldBeBorn` se verifica si la celda debe nacer para la siguiente iteración. El método asume correctamente que la celda está muerta. Para que una celda nazca se debe cumplir que tenga 3 o 6 vecinos vivos. Erróneamente el método solamente chequea si la celda posee 6 vecinos vivos.
Este error se evidencia cuando el test retorna que el valor obtenido en la celda `(5, 9)` es `false`, pero se esperaba que fuera `true`.

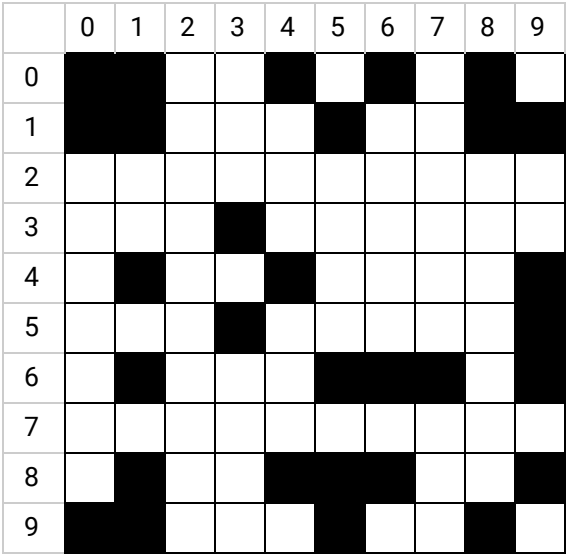
ERROR	ARREGLADO

`if(numAliveNeighbors == 6)`
`  return true;`
	`if(numAliveNeighbors == 3 ||`
	`  numAliveNeighbors == 6)`
	`  return true;`

Test: `Test6`

El propósito de este test es probar que el método `calculateNextState` efectivamente retorne `true` cuando una celda cumple con las condiciones para estar viva en la siguiente iteración.

En `setUp` se instanció un `HighLifeBoard(10, 10, false)` y se utilizó el método `setCell` sin errores (arreglado) para setear en `true` las coordenadas que se presentan gráficamente en el siguiente tablero para facilitar el entendimiento y visualización (en el código fuente del test se puede revisar la llamada a este método con estas coordenadas).



Para este test se usaron los siguientes inputs:

- Internos.
  - `(3, 4)`
  - `(6, 6)`
  - `(6, 5)`
  - `(7, 3)`
- Esquinas.
  - `(0, 0)`
  - `(9, 0)`
  - `(0, 9)`
  - `(9, 9)`
- Bordes.
  - `(5, 0)`
  - `(9, 5)`
  - `(5, 9)`
  - `(0, 5)`

Se utilizan estos conjuntos de input para cubrir todos los posibles tipos de celdas: celdas internas, celdas en bordes y celdas en esquinas del tablero.

No se encontraron errores con este test

Análisis

A continuación se responderán algunas de las preguntas para analizar *Unit Testing*.

¿Cuáles son sus ventajas?

Unit testing hace del proceso de desarrollo más ágil de lo que puede ser por sí solo. Cuando se agregan cada vez más funcionalidades a un software, a veces se tiene que modificar y rediseñar el código, sin embargo, cuando se cambia código que ya fue testeado puede ser muy riesgoso y costoso. Si en cambio tenemos unit tests implementados, podemos realizar refactoring sin problemas. Este método está muy asociado con el desarrollo ágil debido a que construye tests que permiten realizar cambios de manera simple y segura.

Los problemas en el software son hallados en etapas tempranas del desarrollo. Como el unit testing es llevado a cabo por desarrolladores que prueban piezas individuales de código antes de la integración, los problemas y errores pueden ser resueltos en ese momento, minimizando el riesgo de impacto en otros componentes del software.

Al identificar en el código los errores y defectos antes de realizar la integración, permite que se expongan los casos bordes y asegura que el código a ser integrado sea de una calidad superior. Permite también aumentar la confianza de los desarrolladores al mantener y refactorizar el código. Esto se produce porque el método verifica la correctitud de cada unidad. Dada la verificación de las unidades individuales, se facilita el testing de integración.

El tener que desarrollar código que satisfaga los tests unitarios conlleva a que el diseño del código, en cuanto a la forma de resolver la funcionalidad, sea de mejor calidad. Si la unidad es capaz de realizar la funcionalidad en pocas líneas, o de forma simple, significa que dicha unidad tiene alta cohesión y está bien definida.

Dado que los errores son encontrados en etapas tempranas del desarrollo, el test unitario permite reducir los costos de arreglar dichos errores. Los errores detectados en estas etapas son fáciles de detectar y más fáciles de arreglar, mientras que errores detectados en etapas tardías del desarrollo son generalmente resultado de múltiples errores con orígenes diferentes, provocando que el hallazgo y resolución de estos sea mucho más difícil.

Un ejemplo muy claro del beneficio de realizar unit testing es el siguiente. Si se tuvieran 2 unidades de código y se decidiera por integrarlas sin ser testeadas por separado. Luego de la integración podrían aparecer errores en múltiples partes, pero cómo podemos saber cuál es el origen (o los orígenes) de los errores. No podemos saber con claridad si el error está en la primera o en la segunda unidad, o si está en la interfaz entre ambas unidades. Si en cambio realizaramos unit testing antes de la integración, no podemos asegurar que los errores futuros

en la integración desaparecerán, pero sí podemos descartar múltiples fuentes posibles del error dado que cada unidad fue testeada individualmente y de forma exhaustiva.

¿Cuáles son sus limitaciones?

En testing en general, no podemos esperar que se encuentren absolutamente todos los errores. Esto ocurre, por ende, también en unit testing. Las pruebas unitarias pueden mostrar la existencia de errores, pero no la inexistencia absoluta de ellos.

Por definición, este método solamente prueba las unidades de código y, por lo tanto, podría no encontrar posibles errores de integración, problemas de performance u otros problemas a nivel de sistema. En este sentido, las pruebas unitarias no entregan indicios de cómo deben funcionar los componentes en conjunto, ni como debe ser el comportamiento del sistema en un nivel más superior.

Un ejemplo de esta principal limitación es que si un equipo de desarrollo está implementando una aplicación web de e-commerce, y por una parte un desarrollador implementa el login de usuarios y realiza las pruebas unitarias correspondientes sobre este componente, para posteriormente integrarlo al frontend de la aplicación, podría ocurrir que existan errores en este proceso de integración y los test unitarios no entregan información que ayude a resolver este problema. Este método de testing nunca permitirá encontrar el origen de errores provocados por integración entre unidades. Por otra parte, unit testing tampoco permitirá testear la performance del sistema ni muchos de sus atributos de calidad como la escalabilidad y la disponibilidad.