IIC3745 Testing - Tarea 1

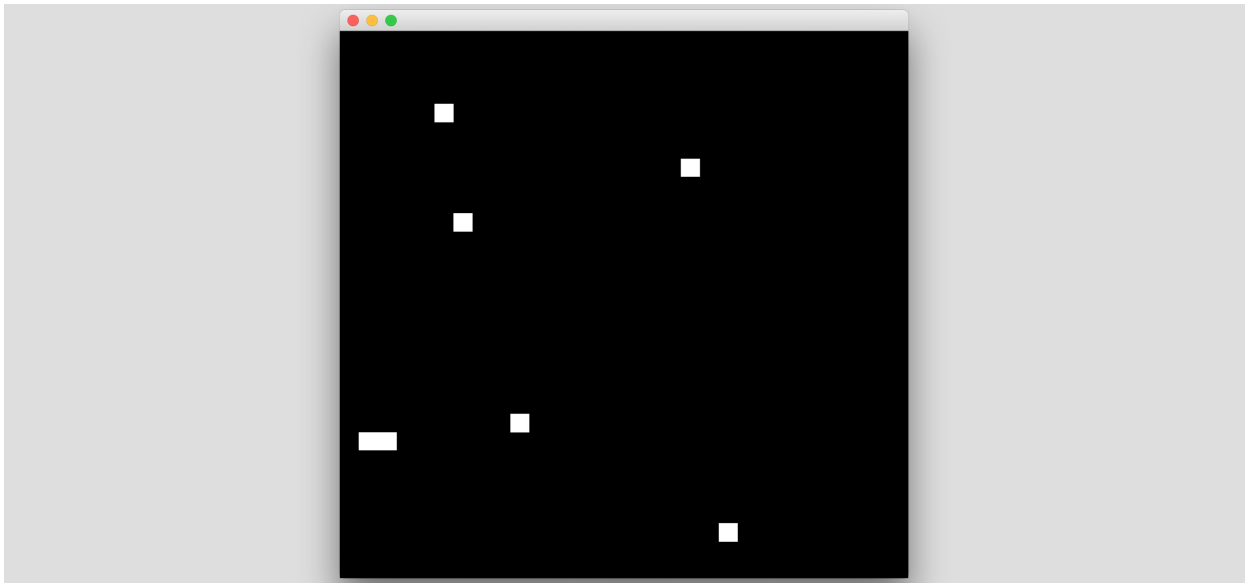
Repo negebauer

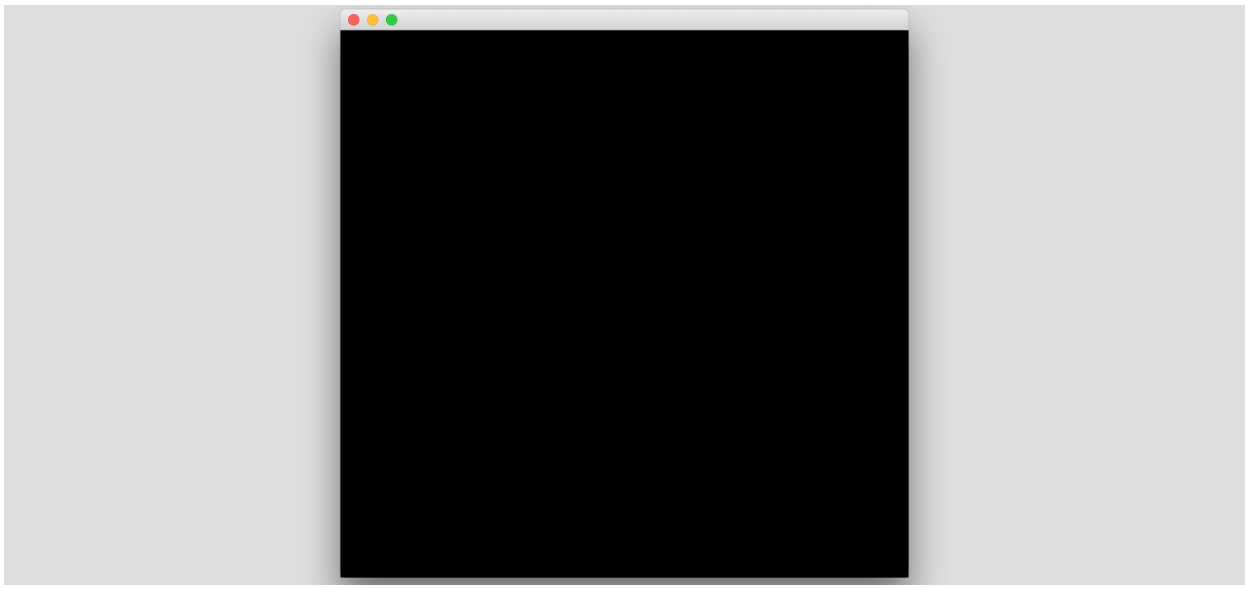
Preparando desarrollo de tests

El primer paso para desarrollar la tarea fue importar los archivos de la carpeta `HighLife/src` en un nuevo proyecto de Eclipse. Al hacer esto se mostraron *warnings* dentro del programa, los cuales fueron arreglados.

Description	Resource	Path	Location	Type
Warnings (4 items)				
The import java.awt.Color is never used	HighLifeGUI.java	/t1/src	line 1	Java Problem
The import java.util.Arrays is never used	HighLifeBoard...	/t1/src	line 1	Java Problem
The import javax.swing.JFrame is never used	HighLifeGUI.java	/t1/src	line 10	Java Problem
The serializable class HighLifeGUI does not declare a static final serialVersionUID field of type long	HighLifeGUI.java	/t1/src	line 13	Java Problem

Ahora con el proyecto sin *warnigs* se procede a correr el programa.





Luego de correrlo varias veces pareciera ser que ocurren las siguientes situaciones:

- No se respeta la regla de sobrevivencia. Es decir, si una célula viva tiene 2 o 3 vecinos muere, en vez de mantenerse viva
- Si se respeta la regla para crear nuevas células.

Para probar la primera hipótesis se aumentó la probabilidad de que una célula este viva al principio (estado inicial). Esto se realiza cambiando la línea donde se realiza `random` en el constructor de `HighLifeBoard`. Se deja la lína como `this.board[i][j] = Math.random() > 0.4 ? true : false;` y se prueba de nuevo. Se confirma que las células siempre mueren (aún cuando deberían sobrevivir). También a veces se crea una nueva. También se cambió el orden de la simulación para tener el primer `setDisplayData` antes de comenzar a simular y también `Thread.sleep(1000);` antes de `board.simulate();` para poder apreciar mejor lo que sucede.

Con esto en mente se procede a revisar el código en búsqueda de potenciales objetivos de

UN. Para ello solo se revisa el archivo `HighLife/src/HighLifeBoard.java`, dado que la funcionalidad del juego se encuentra contenida en dicho archivo.

`public boolean[][] getData()`: Dado que se puede crear un *board* a partir de una matriz fija, podemos usar este método para testear el constructor junto con este método. Para ello creamos un nuevo *HighLifeBoard* a partir de una matriz, luego revisamos que el retorno de `getData` coincida con este.

`public void setCell(int i, int j, boolean value)`: Este método setea el valor de una celda. Se espera que cuando se le de un valor para cierta coordenada dicha celda tenga ese valor. Luego, se puede testear dandole distintas coordenadas junto con distintos estados y luego revisar si `board[][]` tiene el valor que se le dio llamando a `setCell`.

`public HighLifeBoard(int length, int width, boolean random)`: Este método rellena el *board* de manera aleatoria (si se le pide). Si no, crea un *board* vacío. Se puede testear llamando con `random` `true` y `false`, luego revisar que si es `false` no haya ninguna célula viva. Si es `true` debería haber alguna célula viva. Dado que se llena de manera aleatoria no podemos definir un número fijo que debería cumplirse, por lo que testear porque haya por lo menos 1 debería ser suficiente.

`public HighLifeBoard(int length, int width)`: Este constructor puede ser testeado de la misma forma que el anterior, revisando que el *board* este vacío.

`public HighLifeBoard(boolean[][] board)`: Este método es testeado junto con `getData()`.

`public boolean isAlive(int i, int j)`: Importante testear este método para que retorne el valor correcto. Cabe destacar que se debería testear llamando con coordenadas válidas como no válidas (fuera del mapa) y luego probar que se cumplan las reglas (por ejemplo, que todo fuera del mapa sea considerado `false`).

`public int countAliveNeighbors(int i, int j)`: También importante de testear. Para ello podemos crear un *board* propio y revisar que los conteos de vecinos sean iguales a los esperados de acuerdo a las reglas del juego. Tomar en cuenta casos dentro del *board* como en el borde y fuera.

`public boolean shouldSurvive(int i, int j)`: Testear con un *board* predefinido y revisar que retorne correctamente en base a las reglas. Importante probar también con casos borde. Con una mirada rápida se puede apreciar que tiene un error ya que hace `numAliveNeighbors == 2 && numAliveNeighbors == 3` en vez de `numAliveNeighbors == 2 || numAliveNeighbors == 3`. Esto va de la mano con las primeras hipótesis sobre el programa (ver más arriba) y quedará en evidencia en los tests. Tampoco se revisa que la célula esté viva en primer lugar.

`public boolean shouldBeBorn(int i, int j)`: Mismo que el anterior. También se aprecia un error ya que solo revisa si se tienen 6 vecinos, cuando tener 3 también resulta en un nacimiento. Tampoco se revisa que la célula esté muerta en primer lugar.

`public boolean calculateNextState(int i, int j)`: Este método es más complejo de probar ya que para ello tenemos que tener un *board* predefinido y luego probar que el futuro estado de las células coincida con el calculado a mano.

`public void simulate()`: Lo mismo que el método anterior pero se debe tomar en cuenta todo el board, no solo casos en particular.

`public String toString()`: Este método también podría ser testeado, pero no tiene tanto valor. Esto dado que si los métodos anteriores están correctos se podrá apreciar facilmente en consola si este tiene un problema.

Con esta información en mente se proceden a crear los distintos tests, los cuales se pueden encontrar en `HighLife/src/HighLifeBoardTests.java`.

Creando y ejecutando tests. Bug fixes

Primero se crean tests para probar la creación del *board*, estos son:

- shouldCreateBoardWithSizeTest
- shouldCreateBoardWithSizeNoRandomTest
- shouldCreateBoardWithSizeAndRandomTest
- shouldCreateBoardWithBoardTest

Estos pasan todos, por lo que no hay que hacer modificaciones ni a los contructores ni a `getData`.

Luego se crean tests para probar el seteo de celdas (`setCell`), este test es `shouldSetCellTest`. Para ello se prueba la celda (0,0), cambiando su valor a `false`, checkear que sea `false` y luego a `true` y checkear que sea `true`. También la implementación se encarga de que uno no escape de los bordes, cambiando los indices al correspondiente al borde. Por ello, se prueba el (0,0) usando (-1,-1) y el (length, width) usando (length+1, width+1). Este test falla, primero por que no se setean los valores correctamente y luego por salir del bound del array. Por lo que se modifica el código:

```
// Código con error, no pasa test
public void setCell(int i, int j, boolean value) {
    if(i < 0)
        i = 0;
    else if(i >= this.length)
```

```
        i = this.length - 1;

    if(j < 0)
        j = 0;
    else if(j >= this.width)
        i = this.width - 1;
}
```

```
// Código arreglado, pasa test
public void setCell(int i, int j, boolean value) {
    if(i < 0)
        i = 0;
    else if(i >= this.length)
        i = this.length - 1;

    if(j < 0)
        j = 0;
    else if(j >= this.width)
        j = this.width - 1; // Aquí se estaba cambiando i en vez de j

    board[i][j] = value; // No se hacía la asignación
}
```

Los errores fueron porque primero faltaba la asignación del valor. Luego de arreglar esto se comprueba, al revisar fuera de los bordes, que hay una asignación de borde mala, ya que el test sigue fallando.

Despues se checkea la revisión de si una celda está vacía. La regla para esto es:

1. Si se está dentro del *board*, obtener el valor directamente de este
2. Si se está fuera del *board*, entonces se considera como muerta

Esto implica que las celdas en las orillas solo pueden tener un máximo de 5 vecinos. Para checkear esto se crea el test `shouldCheckCellAlive`. Este falla, por lo que se cambia el código para que cumpla con las reglas mencionadas.

```
// Código con error, no pasa test
public boolean isAlive(int i, int j) {
  if(i < 0 || i >= length)
    return false;
  else if (j < 0 || j >= width)
    return true;
  else
    return board[i][j];
}
```

```
// Código arreglado, pasa test
public boolean isAlive(int i, int j) {
  if(i < 0 || i >= length || j < 0 || j >= width) // Fuera del borde es muerto
    return false;
  return board[i][j];
}
```

Luego se pasa a testear el método `countAliveNeighbors` a través del test `shouldCountAliveNeighborsTest`. Primero se prueba con el punto (0,1), el cual no tiene vecinos, y el test falla. Esto ya que dice que tiene 1 vecino. Inspeccionando el código se ve que se está contando el punto mismo (i,j) como vecino, lo cual es incorrecto. Se prueban también varios otros puntos, en particular casos bordes, pero no se encuentran más problemas. El cambio de código es el siguiente:

```
// Código con error, no pasa test
public int countAliveNeighbors(int i, int j) {

  int total = 0;

  total += this.isAlive(i - 1, j - 1) ? 1 : 0;
  total += this.isAlive(i - 1, j) ? 1 : 0;
  total += this.isAlive(i - 1, j + 1) ? 1 : 0;
  total += this.isAlive(i, j - 1) ? 1 : 0;
  total += this.isAlive(i, j) ? 1 : 0;
  total += this.isAlive(i, j + 1) ? 1 : 0;
  total += this.isAlive(i + 1, j - 1) ? 1 : 0;
  total += this.isAlive(i + 1, j) ? 1 : 0;
  total += this.isAlive(i + 1, j + 1) ? 1 : 0;

  return total;
}
```

```
// Código arreglado, pasa test
public int countAliveNeighbors(int i, int j) {

  int total = 0;

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

```
    return total;
  }
```

Luego pasamos a probar el método `shouldSurvive` a través del test `shouldDecideIfSurviveTest`. El test falla para celdas que tienen 2 y 3 vecinos, pero no falla cuando la celda debería morir. Se revisa el código y el problema es que se está usando `&&` en vez de `||`. Esto dado que una célula no puede tener 2 y 3 vecinos al mismo tiempo. No se revisa que la célula esté vacía ya que esto se realiza al calcular el próximo estado, es decir, este método tiene como precondición que la célula esté viva. El cambio de código es el siguiente:

```
// Código con error, no pasa test
public boolean shouldSurvive(int i, int j) {

  int numAliveNeighbors = this.countAliveNeighbors(i, j);

  if(numAliveNeighbors == 2 && numAliveNeighbors == 3)
    return true;
  else
    return false;
}
```

```
// Código arreglado, pasa test
public boolean shouldSurvive(int i, int j) {
  int numAliveNeighbors = this.countAliveNeighbors(i, j);

  if(numAliveNeighbors == 2 || numAliveNeighbors == 3)  // Arreglar revisión de regla
    return true;
  return false;
}
```

Ahora pasamos a revisar el método `shouldBeBorn` a través del test `shouldDecideIfBornTest`. Se ve que el método falla solo cuando se tiene una célula con 3 vecinos, se revisa el código y se nota que el caso de 6 vecinos no está siendo tomando en cuenta. El cambio de código es el siguiente:

```
// Código con error, no pasa test
public boolean shouldBeBorn(int i, int j) {

  int numAliveNeighbors = this.countAliveNeighbors(i, j);

  if(numAliveNeighbors == 6)
    return true;
  else
```

```
    return false;
}
```

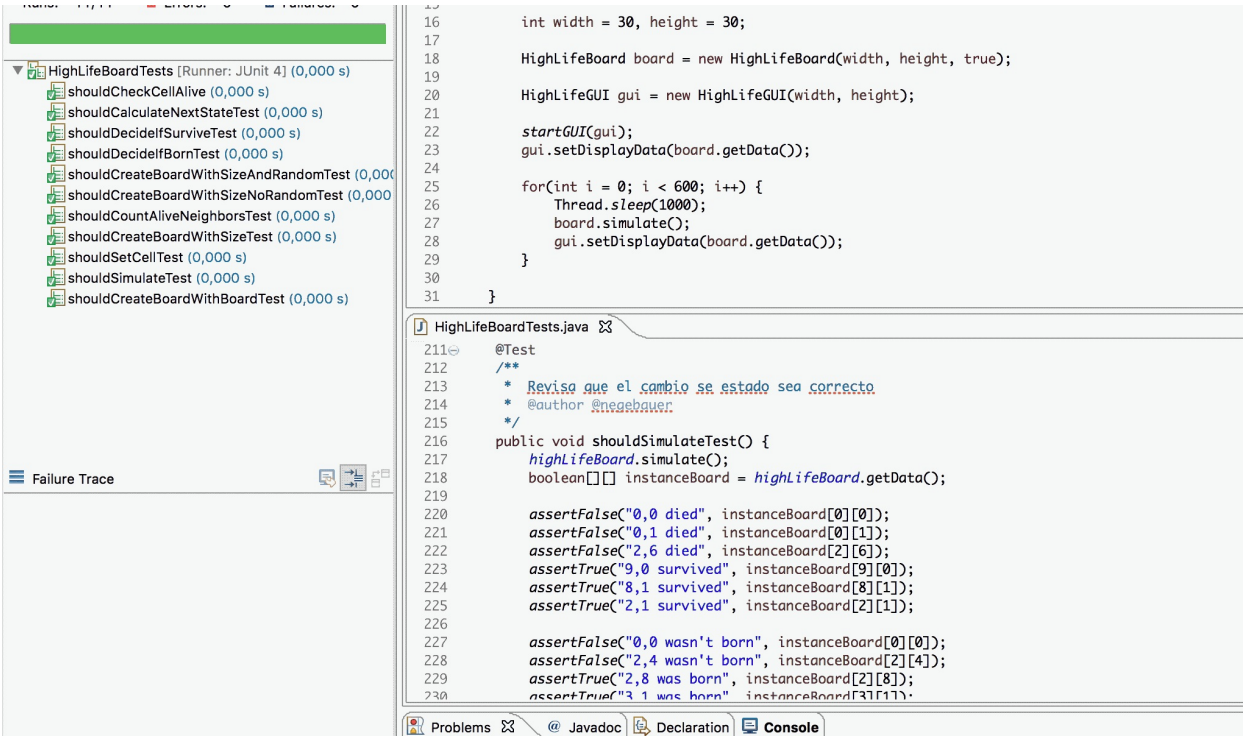
```
// Código arreglado, pasa test
public boolean shouldBeBorn(int i, int j) {
  int numAliveNeighbors = this.countAliveNeighbors(i, j);

  if(numAliveNeighbors == 6 || numAliveNeighbors == 3)  // Revisión de regla faltante
    return true;
  return false;
}
```

Ahora que nuestros métodos que revisan sobrevivencia y nacimiento están correctos podemos pasar a testear el método que los utiliza a ambos, `calculateNextState`, a través del test `shoudlCalculateNextStateTest`. Para este test podemos reutilizar pruebas usadas en los 2 métodos anterios (`shouldSurvive` y `shouldBeBorn`). Este método no reporta ningún problema, lo cual es resultado de los arreglos realizados anteriormente.

Y finalmente testeamos el método `simulate` a través del test `shouldSimulate`, el cual revisa que el avance de un estado al siguiente sea correcto. No se encuentras errores en este método, nuevamente gracias a los arreglos anteriores.

Ahora se vuelve a correr el programa y se tiene un resultado mucho mejor, como se puede ver en el gif adjunto. (Se necesita un sistema compatible para verlo, o revisar el repo cuyo link está al principio)



Análisis Unit Testing

- ¿Cuáles son sus ventajas?
- ¿Cuáles son sus limitaciones?
- ~~¿Qué garantías entrega Unit Testing sobre un componente de software?~~

¿Cuáles son sus ventajas?

Tener un Unit Test de un componente de software sirve al equipo de desarrollo para identificar problemas. En particular, cuando el código cambia, un test unitario puede identificar problemas que nacen de dichos cambios. Esto facilita cambios del código, ya que ayuda a encontrar problemas y por ello arreglarlos antes. También facilitan la integración. Esto ya que si todas las unidades están funcionando correctamente se hace más fácil el proceso de probar la unión entre dichas unidades. Si se encuentran problemas, dado que cada unidad funciona, se puede encontrar más fácilmente el problema (probablemente en la unión de las unidades). Si no hay test unitarios, no hay como saber si el problema es de alguna de las unidades o de la comunicación entre ambas.

¿Cuáles son sus limitaciones?

Como su nombre lo indica, Unit Testing solo se dedica a testear unidades de código. Es decir, solo prueba ciertas partes del código y lo hace en base a ciertos *inputs* y *outputs*. Ayuda a capturar errores en dichas unidades de código. Pero no ayuda a capturar otros problemas como problemas de integración, de performance u otros relacionados al sistema completo. Por ejemplo, podemos tener un método que funciona perfectamente (en base a un unit test) pero en producción dicho método falla ya que toma demasiado tiempo en ejecutarse. O su ejecución en conjunto con otras partes del programa colapsan al sistema. También se puede tener código cuyos unit tests funcionan perfectamente, pero que cuando se comunican entre ellos (por ejemplo, distintas clases) comienzan a aparecer errores. Esto dado que no se esta cubriendo la integración de las unidades. Otro problema importante de Unit Testing es que crece rápidamente, es decir, se tiene que escribir mucho código. Por cada decisión booleana (`true` o `false`) se tienen que crear dos tests, para códigos más complejos pueden ser más aún. Esto puede traducirse en que los desarrolladores estén haciendo más líneas de test que de código.