Schneide Blog – public business secrets

Softwareschneiderei

May 31

Game of Life: TDD style in Java

4

Posted by jenslukowski

I always got problems finding the right track with test driven development (TDD), going down the wrong track can get you stuck.
So here I document my experience with tdd-ing Conway's Game of Life in Java.

The most important part of a game of life implementation since the rules are simple is the datastructure to store the living cells.
So using TDD we should start with it.
One feature of our cells should be that they are equal according to their coordinates:

```
1  @Test
2  public void positionsShouldBeEqualByValue() {
3    assertEquals(at(0, 1), at(0, 1));
4  }
```

The JDK features a class holding two coordinates: java.awt.Point, so we can use it here:

```
1  public class Board {
2    public static Point at(int x, int y) {
3      return new Point(x, y);
4    }
5  }
```

You could create your own Position or Cell class and implementing equals/hashCode accordingly but I want to keep things simple so we stick with Point.
A board should holding the living cells and we need to compare two boards according to their living cells:

```
1  @Test
2  public void boardShouldBeEqualByCells() {
3    assertEquals(new Board(at(0, 1)), new Board(at(0, 1)));
4  }
```

Since we are only interested in living cells (all other cells are considered dead) we store only the living cells inside the board:

```
1  public class Board {
2    private final Set<Point> alives;
3
4    public Board(Point... points) {
```

```
5       alives = new HashSet<Point>(Arrays.asList(points));
6     }
7
8     @Override
9     public boolean equals(Object o) {
10      if (this == o) return true;
11      if (o == null || getClass() != o.getClass()) return false
12
13      Board board = (Board) o;
14
15      if (alives != null ? !alives.equals(board.alives) : board
16
17      return true;
18    }
19
20    @Override
21    public int hashCode() {
22      return alives != null ? alives.hashCode() : 0;
23    }
24  }
```

If you take a look at the rules you see that you need to have a way to count the neighbours of a cell:

```
1  @Test
2  public void neighbourCountShouldBeZeroWithoutNeighbours() {
3    assertEquals(0, new Board(at(0, 1)).neighbours(at(0, 1)));
4  }
```

Easy:

```
1  public int neighbours(Point p) {
2    return 0;
3  }
```

Neighbours are either vertically adjacent:

```
1  @Test
2  public void neighbourCountShouldCountVerticalOnes() {
3    assertEquals(1, new Board(at(0, 0), at(0, 1)).neighbours(at(
4  }
```

```
1  public int neighbours(Point p) {
2    int count = 0;
3    for (int yDelta = -1; yDelta <= 1; yDelta++) {
4      if (alives.contains(at(p.x, p.y + yDelta))) {
5        count++;
6      }
7    }
8    return count;
9  }
```

Hmm now both neighbour tests break, oh we forgot to not count the cell itself:
First the test…

```
1  @Test
2  public void neighbourCountShouldNotCountItself() {
3    assertEquals(0, new Board(at(0, 0)).neighbours(at(0, 0)));
4  }
```

Then the fix:

```
1  public int neighbours(Point p) {
2    int count = 0;
3    for (int yDelta = -1; yDelta <= 1; yDelta++) {
```

```
4       if (!(yDelta == 0) && alives.contains(at(p.x, p.y + yDelta
5         count++;
6       }
7     }
8     return count;
9   }
```

And the horizontal adjacent ones:

```
1   @Test
2   public void neighbourCountShouldCountHorizontalOnes() {
3     assertEquals(1, new Board(at(0, 1), at(1, 1)).neighbours(at(
4   }
```

```
1   public int neighbours(Point p) {
2     int count = 0;
3     for (int yDelta = -1; yDelta <= 1; yDelta++) {
4       for (int xDelta = -1; xDelta <= 1; xDelta++) {
5         if (!(xDelta == 0 && yDelta == 0) && alives.contains(at
6           count++;
7         }
8       }
9     }
10    return count;
11  }
```

And the diagonal ones are also included in our implementation:

```
1   @Test
2   public void neighbourCountShouldCountDiagonalOnes() {
3     assertEquals(2, new Board(at(-1, 1), at(1, 0), at(0, 1)).nei
4   }
```

So we set the stage for the rules. Rule 1: Cells with one neighbour should die:

```
1   @Test
2   public void cellWithOnlyOneNeighbourShouldDie() {
3     assertEquals(new Board(), new Board(at(0, 0), at(0, 1)).next
4   }
```

A simple implementation looks like this:

```
1   public Board next() {
2     return new Board();
3   }
```

OK, on to Rule 2: A living cell with 2 neighbours should stay alive:

```
1   @Test
2   public void livingCellWithTwoNeighboursShouldStayAlive() {
3     assertEquals(new Board(at(0, 0)), new Board(at(-1, -1), at(0
4   }
```

Now we need to iterate over each living cell and count its neighbours:

```
1   public class Board {
2     public Board(Point... points) {
3       this(new HashSet<Point>(Arrays.asList(points)));
4     }
5
6     private Board(Set<Point> points) {
7       alives = points;
8     }
9
10    public Board next() {
```

```
    Set<Point> aliveInNext = new HashSet<Point>();
    for (Point cell : alives) {
      if (neighbours(cell) == 2 {
        aliveInNext.add(cell);
      }
    }
    return new Board(aliveInNext);
  }
}
```

In this step we added a convenience constructor to pass a set instead of some cells.
The last Rule: a cell with 3 neighbours should be born or stay alive (the pattern is called blinker, so we name the test after it):

```
@Test
public void blinker() {
  assertEquals(new Board(at(-1, 1), at(0, 1), at(1, 1)), new B
}
```

For this we need to look at all the neighbours of the living cells:

```
public Board next() {
  Set<Point> aliveInNext = new HashSet<Point>();
  for (Point cell : alives) {
    for (int yDelta = -1; yDelta <= 1; yDelta++) {
      for (int xDelta = -1; xDelta <= 1; xDelta++) {
        Point testingCell = at(cell.x + xDelta, cell.y + yDel
        if (neighbours(testingCell) == 2 || neighbours(testin
          aliveInNext.add(testingCell);
        }
      }
    }
  }
  return new Board(aliveInNext);
}
```

Now our previous test breaks, why? Well the second rule says: a *living* cell with 2 neighbours should stay alive:

```
public Board next() {
  Set<Point> aliveInNext = new HashSet<Point>();
  for (Point cell : alives) {
    for (int yDelta = -1; yDelta <= 1; yDelta++) {
      for (int xDelta = -1; xDelta <= 1; xDelta++) {
        Point testingCell = at(cell.x + xDelta, cell.y + yDel
        if ((alives.contains(testingCell) && neighbours(testi
          aliveInNext.add(testingCell);
        }
      }
    }
  }
  return new Board(aliveInNext);
}
```

Done!
Now we can refactor and make the code cleaner like removing the logic duplication for iterating over the neighbours, adding methods like toString for output or better failing test messages, etc.

Related

TDD myths: the problems
In "2013"

Verbosity is not Java's fault
In "2010"

A more elegant way to equals in Java
In "2009"

Posted in *2012*, *Author: Jens Lukowski*, *Java*, *Testing*

Tagged *java*

Edit

4 THOUGHTS ON "GAME OF LIFE: TDD STYLE IN JAVA"

Pingback: Thoughts about TDD « Schneide Blog Edit

Christian Hujer

— JANUARY 20, 2015 AT 1:35 AM Edit

Using java.awt.Point here violates package principles. The class Board contains business logic, the class java.awt.Point is from the java.awt package and therefore the UI. It would be more convenient if the package that contains Board has zero dependencies on anything related to the UI, so that we could easily verify that our business logic does not wrongly depend on the UI. So we should use a Point class of our own here.

Apart from that, I really enjoyed reading this article and got inspired. I happen to be developing a Game of Life engine for fun just now, and was seeking inspiration about how to test it. You've inspired me, thank you!

Reply

daniel.lindner

— JANUARY 21, 2015 AT 11:49 AM Edit

Hi Christian, thank you for your comment. You are right with your observation that we should use a specific "Point" or "Coordinate" class. The usage of java.awt.Point is a shortcut that saves time only in the short run.

Good find.

Reply

Joachim Dietl

— SEPTEMBER 15, 2016 AT 1:04 PM Edit

use this:
class Point{

private int x;
private int y;
public Point(int x, int y)[
this.x = x;
this.y = y;
}

// no dependency on awt

Reply

Leave a Reply

Enter your comment here...

← *Summary of the Schneide Dev Brunch at 2012-05-27*

Your own CI-based RPM build farm, part 3 →

- Avoid switch! Use enum!
- Using PostgreSQL with Entity Framework

Our archive

Select Month

Our topics

Select Category

Recent Comments

carlo on Ten books that shaped me as a…

How to use CMake / C… on Modern CMake with target_link_…

Philippe BAUCOUR on Integrating catch2 with CMake…

Ten books that shape… on Ten books that shaped me as a…

Ten books that shape… on Book review: Clean Architectur…
