This page last changed on Nov 11, 2010 by scaplette.

Tic Tac Toe via TDD

For this TDD exercise, we will consider a Tic Tac Toe game of 2 players constituted of a 3x3 grid.

Formulating the first test

Let us formulate a first test.

```
A column of 3 item should win
```

We create the test using simple structures and objects in a Java test class with a generic name for the moment (ex: *TicTacToeGameTest.java*). As we write this first test, here is what we are thinking of

- creation of a simple main class (Game) to be able to call methods on it and to encapsulate the first generated logic
- creation of a simple method for the game called 'play' simulating a play
- the method 'play' should accept the position as parameter
- to start simply a position for the play will be represented by 2 int values
- the method 'play' will return a boolean with true for a win
- we should have a simple data structure to retain the state of the game
- a 2 dimensional array of boolean seems reasonable
- in our test we will do assertions for each 'play'
- in our test the first and second 'play' should return false. The third 'play' should return true (i.e. we have detected a win)

We notice that the conception is already taking place and is completely part of the creation of the test.

```
@Test
public void column_of_3_items_should_win() throws Exception {
    Game game = new Game();
    assertFalse(game.play(0,0));
    assertFalse(game.play(0,1));
    assertTrue(game.play(0,2));
}
```

A simple implementation to get the test to pass is

```
public class Game {

        private boolean[][] grid = new boolean[3][3];

        public boolean play(int i, int j) {
                grid[i][j] = true;
                return grid[0][0] && grid[0][1] && grid[0][2];
        }

}
```

Consolidating the implementation

We can see that the implementation only works for the first column. We want it to work for the middle column. We write the next failing test.

```
@Test
        public void middle_column_of_3_items_should_win() throws Exception {
                Game game = new Game();
                assertFalse(game.play(1,0));
```

```
                assertFalse(game.play(1,1));
                assertTrue(game.play(1,2));
        }
```

We get the green bar simply with

```
public boolean play(int i, int j) {
                grid[i][j] = true;
                return grid[0][0] && grid[0][1] && grid[0][2]
                      || grid[1][0] && grid[1][1] && grid[1][2];
        }
```

We can see the pattern and we complete with the last column to win.

```
@Test
        public void last_column_of_3_items_should_win() throws Exception {
                Game game = new Game();
                assertFalse(game.play(2,0));
                assertFalse(game.play(2,1));
                assertTrue(game.play(2,2));
        }
```

Back to the green pastures with

```
public boolean play(int i, int j) {
                grid[i][j] = true;
                return grid[0][0] && grid[0][1] && grid[0][2]
                      || grid[1][0] && grid[1][1] && grid[1][2]
                      || grid[2][0] && grid[2][1] && grid[2][2];

        }
```

Introducing a second player

Let us introduce now the concept of a second player.

With our conception so far, the grid can only contains 2 state for each position: true or false. For our Tictactoe of 2 player, each element in the grid needs to have at least 3 states. A state for an empty position in the grid and a state for each player to differentiate them.

Let us just change the grid to contain int values instead of boolean values. The 'true' boolean value will be replaced by the int value '1'.

The new implementation is

```
public class Game {

        private int[][] grid = new int[3][3];

        public boolean play(int i, int j) {
                grid[i][j] = 1;
                return (grid[0][0] == 1 && grid[0][1] == 1 && grid[0][2] == 1)
                      || (grid[1][0] == 1  && grid[1][1] == 1 && grid[1][2] == 1)
                      || (grid[2][0] == 1  && grid[2][1] == 1  && grid[2][2] == 1);

        }

}
```

... the tests have not changed and they are all still in the green.

Now a second player is introduced with the int value 2. Using the *Change Method Signature* refactoring in our IDE we introduce a third int parameter for the 'play' method. During the refactoring, we choose the default value of '1' for this new parameter so that all the regression passes. Still green!

```
public boolean play(int i, int j, int player) {
                grid[i][j] = player;
                return (grid[0][0] == 1 && grid[0][1] == 1 && grid[0][2] == 1)
                      || (grid[1][0] == 1  && grid[1][1] == 1 && grid[1][2] == 1)
                      || (grid[2][0] == 1  && grid[2][1] == 1  && grid[2][2] == 1);

        }
```

Now we are ready to write a failing test for the second player. The second player is represented by the int value of '2'.

```
...
        @Test
        public void player2_column_of_3_items_should_win() throws Exception {
                Game game = new Game();
                assertFalse(game.play(0,0,2));
                assertFalse(game.play(0,1,2));
                assertTrue(game.play(0,2,2));
        }
...
```

After a bit of cleaning with an *Extract Method* refactoring, the implementation becomes

```
public class Game {

        private int[][] grid = new int[3][3];

        public boolean play(int i, int j, int player) {
                grid[i][j] = player;
                return playerOneHasColumn() || playerTwoHasColumn();

        }

        private boolean playerOneHasColumn() {
                return (grid[0][0] == 1 && grid[0][1] == 1 && grid[0][2] == 1)
                        || (grid[1][0] == 1  && grid[1][1] == 1 && grid[1][2] == 1)
                        || (grid[2][0] == 1  && grid[2][1] == 1 && grid[2][2] == 1);
        }
        private boolean playerTwoHasColumn() {
                return (grid[0][0] == 2 && grid[0][1] == 2 && grid[0][2] == 2)
                        || (grid[1][0] == 2  && grid[1][1] == 2 && grid[1][2] == 2)
                        || (grid[2][0] == 2  && grid[2][1] == 2 && grid[2][2] == 2);
        }

}
```

We can now implement the winning row (and winning diagonals) for the 2 players without difficulty

```
public boolean play(int i, int j, int player) {
                grid[i][j] = player;
                return playerOneHasColumn() || playerTwoHasColumn() ||
                           playerOneHasRow() || playerTwoHasRow() ||
                           playerOneHasDiagonal() || playerTwoHasDiagonal();

        }
```

We have reached at this point an basic implementation that can detect a win for 2 distinct players.

We notice as well that at the moment, we do not prevent a player to play in the same spot as the previous player.

Obvious refactoring

Before continuing implementing business logic let us do a bit of cleaning.

We can see in the current implementation a lot of redundancy. Using *Change Method Signature* refactoring again on *playerOneHasColumn* and *playerOneHasRow* we introduce an int parameterto indicate the player. During the refactoring in our IDE, we default the value of the parameter to 1 (instead of 0) and name it 'player'.

Here is the code we get

```
public class Game {

	private int[][] grid = new int[3][3];

	public boolean play(int i, int j, int player) {
		grid[i][j] = player;
		return hasColumn(1) || hasColumn(2) ||
				hasRow(1) || hasRow(2) ||
				hasDiagonal(1) || hasDiagonal(2);

	}

	private boolean hasColumn(int player) {
		return (grid[0][0] == player && grid[0][1] == player && grid[0][2] == player)
			|| (grid[1][0] == player  && grid[1][1] == player && grid[1][2] == player)
			|| (grid[2][0] == player  && grid[2][1] == player && grid[2][2] == player);
	}
	private boolean hasRow(int player) {
		return (grid[0][0] == player && grid[1][0] == player && grid[2][0] == player)
		|| (grid[0][1] == player  && grid[1][1] == player && grid[2][1] == player)
		|| (grid[0][2] == player  && grid[1][2] == player && grid[2][2] == player);
	}
	private boolean hasDiagonal(int player) {
		return (grid[0][0] == player && grid[1][1] == player && grid[2][2] == player)
		|| (grid[0][2] == player  && grid[1][1] == player && grid[2][0] == player);
	}
}
```

Occupied spot feature

As we have noticed before nothing prevent a player from playing where the previous player played.

I want the 'play' method to leave the grid as it is and to return -1 when a player plays in an occupied spot.
It means I have to change the return type of the 'play' method. It will be an int.

My 'play' method simple contract will be:

- return 1 when a win is detected
- return 0 when a play is in an empty spot and no win is detected
- return -1 when a play is in an occupied spot. The method does nothing in this case

A refactoring is needed.

We refactor the test first using the time saver *Find/Replace* function in our IDE.

- Find the expression (with case) *assertFalse(* and replace it by *assertEquals(0,*
- Find the expression (with case) *assertTrue(* and replace it by *assertEquals(1,*

As an example this is how a test would look like

```
@Test
        public void player1_column_of_3_items_should_win() throws Exception {
                Game game = new Game();
                assertEquals(0, game.play(0,0,1));
                assertEquals(0, game.play(0,1,1));
                assertEquals(1, game.play(0,2,1));
        }
```

Refactoring the implementation the tests become green with

```
...
        private int[][] grid = new int[3][3];

        public int play(int i, int j, int player) {
                grid[i][j] = player;
                if(hasColumn(player) || hasRow(player) || hasDiagonal(player)){
                    return 1;
                } else return 0;
        }
...
```

Now we are ready to write the next failing test for our occupied spot feature. The 'play' method return -1 when a player plays in an occupied spot.

```
@Test
        public void player_1_return_minus_one_if_token_is_play_in_occupied_spot() throws Exception
 {
                Game game = new Game();
                assertEquals(0, game.play(0,0,1));
                assertEquals(-1, game.play(0,0,1));
        }
```

Implementation

```
public int play(int i, int j, int player) {
                if(grid[i][j] != 0){
                        return -1;
                }
                grid[i][j] = player;
                if(hasColumn(player) || hasRow(player) || hasDiagonal(player)){
                     return 1;
                } else return 0;
        }
```

The game has a basic API and feels more complete. I will rename the Game class to be Tictactoe and rename the test TictactoeTest.

Gravitation feature

Currently, our implementation do not respect the law of gravity. The first play cannot be in the middle of the grid!

The feature is that a play must be positioned at the lowest level or must have a occupied position below it, otherwise we return -1.

Let us formulate a failing test.

```
should return -1 for a play that does not have an occupied position below
```

```
@Test
```

```
        public void should_return_minus_1_if_newly_play_token_has_no_token_below () throws
 Exception {
                Tictactoe tictactoe = new Tictactoe();
                assertEquals(-1, tictactoe.play(1,1,1));
        }
```

The simplest implementation is

```
public int play(int i, int j, int player) {
                if(grid[i][j] != 0){
                        return -1;
                }
                if(grid[i][j-1] == 0){
                        return -1;
                }
                grid[i][j] = player;
                if(hasColumn(player) || hasRow(player) || hasDiagonal(player)){
                    return 1;
                } else return 0;
        }
```

The tests break because of '*if(grid[i][j-1] == 0)*'. I need to check that *j>0*. This specification was part of my feature.

The implementation becomes after renaming the variables for clarity

```
public int play(int col, int row, int player) {
                if(grid[col][row] != 0){
                        return -1;
                }
                if(row > 0 && grid[col][row-1] == 0){
                        return -1;
                }
                grid[col][row] = player;
                if(hasColumn(player) || hasRow(player) || hasDiagonal(player)){
                    return 1;
                } else return 0;
        }
```

My tests are still failing! Of course, because the set up for some of my tests were not respecting the gravity law. Let us fix them. Ouch!! It is a long process and forces us to write complete game scenarios.

Example of a fixed test is

```
@Test
        public void player1_middle_row_of_3_items_should_win() throws Exception {
                Tictactoe tictactoe = new Tictactoe();
                assertEquals(0, tictactoe.play(0, 0, 2));
                assertEquals(0, tictactoe.play(1, 0, 1));
                assertEquals(0, tictactoe.play(2, 0, 2));

                assertEquals(0, tictactoe.play(0,1,1));
                assertEquals(0, tictactoe.play(1,1,1));
                assertEquals(1, tictactoe.play(2,1,1));
        }
```

More refactoring

By applying a few *Extract Method* refactorings we have

```
...
        public int play(int col, int row, int player) {
```

```
                if(occupied(col, row) || defyingGravity(col, row)){
                        return -1;
                }

                grid[col][row] = player;

                if(win(player)){
                        return 1;
                } else return 0;
        }

        private boolean win(int player) {
                return hasColumn(player) || hasRow(player) || hasDiagonal(player);
        }

        private boolean defyingGravity(int col, int row) {
                return row > 0 && grid[col][row-1] == 0;
        }

        private boolean occupied(int col, int row) {
                return grid[col][row] != 0;
        }
...
```

Introducing the Grid class

Here are the refactoring steps to introduce/extract the Grid class

- use *Extract Class* on instance fields grid
- adding a *setValue* and *valueAt* on grid
- making instance field grid private in Grid class
- pushing the methods in the Grid class

```
public class Grid {

        private int[][] grid;

        public Grid() {
                this.grid = new int[3][3];
        }

        public void setValue(int x, int y, int value) {
                grid[x][y] = value;
        }

        public int valueAt(int x, int y) {
                return grid[x][y];
        }

        public boolean invalidPosition(int col, int row) {
                return occupied(col,row) || defyingGravity(col,row);
        }

        public boolean defyingGravity(int col, int row) {
                return row > 0 && valueAt(col,row-1) == 0;
        }

        public boolean occupied(int col, int row) {
                return valueAt(col,row) != 0;
        }

        public boolean hasFullColumnOf(int value) {
                return ((valueAt(0,0) == value && valueAt(0,1) == value && valueAt(0,2) == value)
                                || (valueAt(1,0) == value  && valueAt(1,1) == value && valueAt(1,2)
 == value)
```

```
                                || (valueAt(2,0) == value  && valueAt(2,1) == value && valueAt(2,2)
 == value));
        }

        public boolean hasFullRowOf(int value) {
                return ((valueAt(0,0) == value && valueAt(1,0) == value && valueAt(2,0) == value)
                                || (valueAt(0,1) == value  && valueAt(1,1) == value && valueAt(2,1)
 == value)
                                || (valueAt(0,2) == value  && valueAt(1,2) == value && valueAt(2,2)
 == value));
        }

        public boolean hasFullDiagonalOf(int value) {
                return ((valueAt(0,0) == value && valueAt(1,1) == value && valueAt(2,2) == value)
                                || (valueAt(0,2) == value  && valueAt(1,1) == value && valueAt(2,0)
 == value));
        };
}
```

The Tictactoe class has become

```
public class Tictactoe {

        private Grid grid = new Grid();

        public int play(int col, int row, int player) {
                if(grid.invalidPosition(col,row)){
                        return -1;
                }

                grid.setValue(col, row, player);

                if(win(player)){
                        return 1;
                } else return 0;
        }

        private boolean win(int player) {
                return grid.hasFullColumnOf(player)
                                || grid.hasFullRowOf(player)
                                || grid.hasFullDiagonalOf(player);
        }
}
```