UNIVERSITÄT
BERN

P2 - Lab 05

Assignment 04

Was your code testable? (did you change it?)

```java
@Test
public void moveDown3() {
    turtle.move(new Down(3), board);

    assertTrue(board[0][1]);
    assertTrue(board[0][2]);
    assertTrue(board[0][3]);
    assertFalse(board[0][4]);
}
```

what about board[0][5]?

```java
    @Test
    public void moveDown3() {
        turtle.move(new Down(3), board);

        assertTrue(board[0][1]);
        assertTrue(board[0][2]);
        assertTrue(board[0][3]);

        for(int i=4; i<100; i++)
           assertFalse(board[0][i]);
        }

        for(int i=1; i<100; i++)
          for(int j=0; j<100;j++) {
             assertFalse(board[i][j]);
        }

    }
```

Can we do better?

```
    @Test
    public void moveDown3() {
        turtle.move(new Down(3), board);

        expectedBoard = new boolean[100][100];
        expectedBoard[0][1] = true;
        expectedBoard[0][2] = true;
        expectedBoard[0][3] = true;

        assertArrayEquals(expectedBoard, board);
    }
```

```
boolean vector[];
```

```
boolean[] vector;
```

```
boolean board[][];
```

```
boolean[] board[];
```

```
boolean[][] board;
```

Keep the declaration of the type in one place

```java
int distance = Integer.parseInt("0");
```

X

```java
int distance = 0;
```

```java
int distance;
```

?

Variables of type ‘int’ are
by default initialized with 0

```
board = new boolean[size][size];
for (int i = 0; i < size; i++){
    for(int j = 0; j < size; j++){
        this.board[i][j] = false;
    }
}
```

X

```
board = new boolean[size][size];
```

Boolean variables default to false

All primitive types have a default value

```
if (direction.equals("") || steps == 0 ||
   (direction.equals("right") && (steps + colTurtle > 99)) ||
   (direction.equals("down") && (steps + rowTurtle > 99)) ||
   (direction.equals("left") && (colTurtle - steps < 0)) ||
   (direction.equals("up") && (rowTurtle - steps < 0)) )
       throw new ParserException();
```

X

```
if (direction.equals(""))
 throw new ParserException("Empty command");
if (steps == 0)
 throw new ParserException("Missing steps");
if (direction.equals("right") && (steps + colTurtle > 99))
 throw new TurtleOutOfBoardException("right", steps + colTurtle);
if (direction.equals("down") && (steps + rowTurtle > 99))
 throw new TurtleOutOfBoardException("down", steps + rowTurtle);
if (direction.equals("left") && (steps - colTurtle < 0))
 throw new TurtleOutOfBoardException("left", steps - colTurtle);
if (direction.equals("up") && (steps - rowTurtle < 0))
 throw new TurtleOutOfBoardException("up", steps - rowTurtle);
```

Complex boolean conditions are hard to follow

```
try {
    moveRelative(move);
} catch (Exception e) {}
```

X

```
try {
    moveRelative(move);
} catch (TurtleOutOfBoardException e) {
   e.printTrace();
}
```

It’s bad practice to catch ‘Exception’
It will catch everything (NullPointerException)

Do not swallow exceptions without
handling them

```
private void checkBounds(){
    if(x + dx < 0) xEnd = 0;
    else if(x + dx > 99) xEnd = 99;
    else xEnd = x + dx;

    if(y + dy < 0) yEnd = 0;
    else if(y + dy > 99) yEnd = 99;
    else yEnd = y + dy;
}
```

What's 99?

Avoid 'Magic numbers'

```
public final static int NINETY_NINE = 99;

private void checkBounds(){
    if(x + dx < 0) xEnd = 0;
    else if(x + dx > NINETY_NINE) xEnd = NINETY_NINE;
    else xEnd = x + dx;

    if(y + dy < 0) yEnd = 0;
    else if(y + dy > NINETY_NINE) yEnd = NINETY_NINE;
    else yEnd = y + dy;
}
```

Even Worse

Avoid ‘Magic constants’

```
public final static int NINETY_NINE = 100;
```

== vs. equals

```
String command1 = new String("down");
String command2 = new String("down");

System.out.println( command1 == command2 );
System.out.println( command1.equals(command2) );
```

false
true

Use ‘equals’ to compare String object.
‘equals’ tests equality.
‘==’ test the identity of objects

```
String command1 = "down";
String command2 = "down";

System.out.println( command1 == command2 );
System.out.println( command1.equals(command2) );
```

true
true

String literals are reused: command1 and command 2 represent the same object

```java
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```java
Point p1 = new Point(0, 0);
Point p2 = new Point(0, 0);
```

‘new’ always creates a new object

false

```java
p1 == p2
```

‘==’ tests if p1 and p2 represent the same object

false

```java
p1.equals(p2)
```

the default implementation of ‘equals’ tests if p1 and p2 are the same object

```java
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object anObject) {
        if (anObject == null)
            return false;
        if (getClass() != anObject.getClass())
            return false;

        Point other = (Point) anObject;
        return (x == other.x) && (y == other.y);
    }
}
```

```java
Point p1 = new Point(0, 0);
Point p2 = new Point(0, 0);
```

```java
p1 == p2
```

false

```java
p1.equals(p2)
```

true

Most collections from Java use equals to search for elements

```java
public class Point {
    private int x;
    private int y;
    private Color color;

    public Point(int x, int y, Color color) {
        this.x = x;
        this.y = y;
        this.color = color;
    }
    @Override
    public boolean equals(Object anObject) {
        if (anObject == null)
            return false;
        if (getClass() != anObject.getClass())
            return false;

        Point other = (Point) anObject;
        return (x == other.x) && (y == other.y);
    }
}
```

```
Point p1 = new Point(0, 0, Color.WHITE);
Point p2 = new Point(0, 0, Color.BLACK);
```

```
p1.equals(p2)
```

true

Depending on what the object represents we might not want to compare all instance variables