



AARHUS UNIVERSITET

# **Software Engineering and Architecture**

A few comments on Mandatory



# What is TDD???

- Traditional tests = Quality Assurance Technique
  - Success:
    - Tests are constructed to catch defects
- TDD tests = ***Implementation Technique***
  - Success: test cases that *drive implementation*
  - Perhaps a few more to show absence of defects
- Not a comprehensive quality assurance technique



- **Evident Tests**

```
@Test
public void plainsEveryWhereBut2_2And1_0And0_1()
{
    //Iterate through all tiles in world
    for(int row = GameConstants.WORLDSIZE-1; row>=0 ; row--) {
        for(int column = GameConstants.WORLDSIZE-1; column >= 0 ; column--) {
            if(!(row == 1 && column == 0 || row == 2 && column == 2 || row == 0 && column == 1)) {
                assertEquals("There should be plains at " + row + ", " + column,
                    game.getTileAt(new Position(row,column)).getTypeString(), GameConstants.PLAINS);
            }
        }
    }
}
```

- Exercise: What is focus here?
  - Test that everything works? Or
  - Drive production code into existence?
- And – is it Evident ?



# How I will do it...

AARHUS UNIVERSITET

- Iteration 1: Fake it – intro 'fake' StandardTile, (return plain)
- Iteration 2: Intro datastructure, fill **all entries** with fake

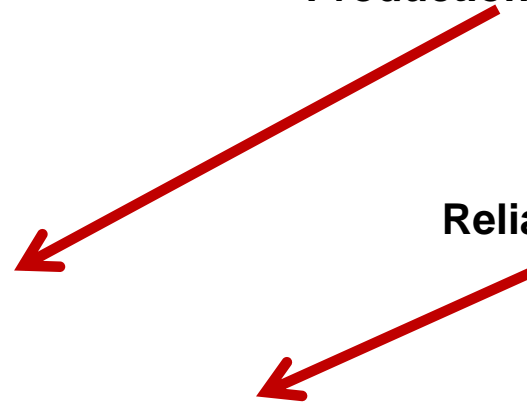
```
@Test
public void shouldMostlyBePlainsInWorld() {
    Position p = new Position(0,0);
    // iteration 1
    assertEquals( GameConstants.PLAINS,
        game.getTileAt(p).getTypeString());
    assertEquals( new Position(0,0),
        game.getTileAt(p).getPosition());

    // iteration 2
    p = new Position(7, 12);
    assertEquals( GameConstants.PLAINS,
        game.getTileAt(p).getTypeString());
    assertEquals( new Position(7,12),
        game.getTileAt(p).getPosition());

    p = new Position(GameConstants.WORLDSIZE-1, GameConstants.WORLDSIZE-1);
    assertEquals( GameConstants.PLAINS,
        game.getTileAt(p).getTypeString());
}
```

Productioncode driver

Reliability





# And then I...

AARHUS UNIVERSITET

- Add test cases for each of the 'exceptions'
  - Test drive that one tile that is a mountain
    - Write the test case 'shouldHaveMountainAt2\_2()'
    - See it fail
    - Enhance the production code to
      - `world[2][2] = new StandardTile(GameConstants.MOUNTAIN);`
      - (or `world.put(new Position(2,2), new StandardTile.....)`)
      - (or ...)
    - See it pass
  - Test drive that one that is Ocean
    - You get it...



# Stable test cases

AARHUS UNIVERSITET

- The *more* your testcases *only* use the given Game, City, Unit interfaces...
- The more stable your test cases will be against refactoring inner data structures!
- So
  - `game.getCityAt(p)` 😊
- Not
  - `((GameImpl) game).getInternalCityHashMap.get(p)` ☹️

# Design Issues

- *Which datastructure should I use?*
  - Any you like
    - Array of arrays
    - HashMaps
    - Perhaps even play with just 'redCity' 'blueCity'
      - Knowing that it *will* become 'fake-it' some time in the future
  - *You have a strong set of test cases and as long as you use the stable Game, City, Unit, interface methods, you can quite quickly refactor your internals*
  - *It is called **encapsulation!***
    - *One of the strongest tools in a software engineer's toolbox!*

Have a look at the  
Sweep slides (week 1)



# Those 'read-only' interfaces

- I stated that *try to keep City, Unit as read only interfaces*
- *How?*
- Actually it is the 'facade' pattern which we will return to later, so...
- Keep it simple, do what you find best
  - We can refactor later, right!





# The 'default-and-patch' approach

- How to fill a datastructure?

```
public GameImpl() {
    // Setup the tile, city and unit locations, creating the world
    this.tiles = new TileImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];
    this.cities = new CityImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];
    this.units = new UnitImpl[GameConstants.WORLDSIZE][GameConstants.WORLDSIZE];

    // Runs through the different tiles to give them the right terrain types
    for(int i = 0; i < GameConstants.WORLDSIZE; i++) {
        for(int j = 0; j < GameConstants.WORLDSIZE; j++) {
            if(i == 1 && j == 0) {
                tiles[i][j] = new TileImpl(GameConstants.OCEANS);
            }
            else if(i == 0 && j == 1) {
                tiles[i][j] = new TileImpl(GameConstants.HILLS);
            }
            else if(i == 2 && j == 2) {
                tiles[i][j] = new TileImpl(GameConstants.MOUNTAINS);
            }
            else {
                tiles[i][j] = new TileImpl(GameConstants.PLAINS);
            }
        }
    }
}
```

Lot of i's and j's,  
lot of if's



# The 'default-and-patch' approach

- Default it, next patch it...

That is:

- a) Fill entire structure with default element
- b) Patch the exceptions

```
for(int i = 0; i < GameConstants.WORLDSIZE; i++){  
    for(int j = 0; j < GameConstants.WORLDSIZE; j++){  
        g.createTile(newPosition(i, j), GameConstants.Plains);  
    }  
}
```

```
g.createTile(new Position(0, 1), GameConstants.HILLS);  
g.createTile(new Position(1, 0), GameConstants.OCEANS);  
g.createTile(new Position(2, 2), GameConstants.MOUNTAINS);
```

Analyzability