AARHUS UNIVERSITET

Software Engineering and Architecture

Comments and Hints on Mandatory

AARHUS UNIVERSITET

Fragile Tests

Avoid as best possible tying your test code to implementation details

‘test through the API’

AARHUS UNIVERSITET

The Problem

- Exercise: What is being tested here

```
@Test
public void blueShouldHaveLegionUnitAtPositionThreeTwo(){
    assertThat(game.getUnitMap().get(new Position( r: 3,  c: 2)).getOwner(), is(Player.BLUE));
    assertThat(game.getUnitMap().get(new Position( r: 3,  c: 2)).getTypeString(), is(GameConstants.LEGION));
}
```

- Exercise: What happens if I need to change the data structure into a Unit[ ][ ] matrix instead???

- Exercise: Could I have expressed this test using the Game's interface directly?

AARHUS UNIVERSITET

A Bit Trickier Problem

- *Test case: I cannot stack two friendly units*
- Problem: How to make the 'fixture'
  - Fixture = the setup of objects in preparation for the actual test
  - Here:
    - Have two RED archers on adjacent tiles
- Solutions?
  - 1) Age the world until two red units exist, move one next to the other
  - 2) gameImpl.setInternalMatrix(4, 4, new UnitImpl(RED, ARCHER))
- Exercise: Which is the solution with most **stability**?

Definition: **Stability (ISO 9126)**

The capability of the software product to avoid unexpected effects from modifications of the system.

AARHUS UNIVERSITET

The Tricky Problem

- Solution 1 is more cumbersome (requires a lot of steps, taken in the proper order (i.e. *less evident test!))* but is a more **stable** test case
  - Game's internal data structures are **encapsulated!**

- Solution 2 is fragile (**less stability**) because if I change stuff inside Game (which is the purpose of encapsulation) then *a lot of test cases will break and require refactoring work!*
  - *I.e. development slows down and becomes non-agile*

AARHUS UNIVERSITET

So What ???

- The better solution is actual a third one
  - Test stubs

- Inject delegates that configure the world to a particular setup
  - A ‘world setup strategy’ that for this particular test case configures the world to have two red archers just next to each other

- *Do not despair, everything will be revealed in time…*

Morale

- **Only** test implementation details if
  - The cannot just as easily be test-driven by using the ‘facade’ / outside interface(s)
  - They are highly complex by themselves and warrant a TDD approach because of this complexity
  - Are evaluated to be ‘stable’
    - I.e. the probability of changing this data structure in future is low

AARHUS UNIVERSITET

Cover the Requirements

A tradeoff, yes I know

AARHUS UNIVERSITET

TDD versus Systematic Test

- TDD is an *implementation* technique and not a Quality Assurance technique
  - We will discuss systematic testing later which *is* a QA technique
  - ”High probability that there are no defects in the code”

- But TDD means of course *covering the requirements!*
  - How else would you *drive* the implementation defining the code that covers the requirements into existence???

- Thus these tests must be in the test code!

Example

- This single test case
  - assertThat(game.moveUnit ((4,3), (5,5)), is(true));
- … was used to cover the full moveUnit into existence

- But moveUnit is pretty complex
  - Non passable terrain
  - No stacking
  - Attacking enemies
  - City conquest

Morale

- TDD is test-driven
  - Write the test to ensure
    - A unit can move from one plain (5,5) to another empty plain (6,6)
      - Write production code
    - Ahh, the unit is really no longer on (5,5) but is on (6,6)
    - A unit cannot move into ocean
      - Write production code
      - Ah – test that it is still on ‘from’ and not on ‘to’
    - A unit cannot stack onto a friendly unit
      - Write production code
- Result: The test code tests all requirements
  - ***And keep testing them throughout the system’s lifetime!***

AARHUS UNIVERSITET

Avoid inheritance for non-behavioral variability

AARHUS UNIVERSITET

Misunderstood subclassing

- Class MountainTile implements Tile ?

```
public interface Tile {

  /** return position of this tile on the board.
   * @return position of tile.
   */
  public Position getPosition();

  /** return the tile type as a string. The set of
   * valid strings are defined by the graphics
   * engine, as they correspond to named image files.
   * @return the type type as string
   */
  public String getTypeString();
}
```

Sorry, these are older slides. The getPosition() method was removed from the Tile interface

- Why is the ratio liabilities/benefits so askewed that it is **wrong** to introduce MountainTile**?**

AARHUS UNIVERSITET

When to use subclassing!

- **Only use subclasses if you get a distinct *behavioral* advantage**
  - Otherwise it is a lot of typing just to define some constant
  - Similar for units: cost, defense, attack,....

- **Parameterization** is run-time changeable, cheap, easy to overview...

```
public class StandardTile implements Tile {

  private Position position;
  private String type;

  public StandardTile(Position position, String type) {
    this.position = position;
    this.type = type;
  }

  @Override
  public Position getPosition() {
    return position;
  }

  @Override
  public String getTypeString() {
    return type;
  }

}
```

One issue: Too many academic textbooks overuse subclassing. They are plainly wrong!

AARHUS UNIVERSITET

Regarding Units

- Unit types are behavioral distinct!
- So: SettlerUnit extends AbstractUnit impl. Unit ?
- *Polymorphic design and it will work*
  - With some *not-so-nice* hacks ala
    - How does a settler unit kill itself as part of its action?
- **But the learning goal in SWEA is compositional design ☺**
  - Make a *StandardUnit/UnitImpl*, and delegate actions to a Strategy!
    - In the next mandatory exercises, for now just '**simplicity'**
  - See if you can get through...
    - Bit tricky with Archers' fortification action☺

My own experience

- I highly prefer
  - A ‘UnitImpl’ / ‘StandardUnit’ that embody general behavior, even if they are irrelevant for a specific subtype
    - ‘isMoveable’ boolean, shared by all, not just by archers
      - It is just always true for Settlers and for Legions…
      - … and opens for future flexibility at no cost
- Because
  - The ‘type switch’ usually pops up anyway
    - ‘if (theUnit instanceof ArcherUnit) {
      - ArcherUnit su = (ArcherUnit) theUnit;
      - su.setMoveable(false);
    - }