P2 - Exercise hour

Julius Oeftiger

2019-03-15
Outline

Ex. 3 Recap

Unit Testing

(Ex. 4 Precap)
Ex. 3 Recap

Parser
- Read input, create command

Command classes
- CommandLeft, -Up, ... implement ICommand
- Knowledge about board not needed!

Turtle
- Stores current position
- Executes commands
class Boardmaker {
    ... 
    public Field[][] makeBoardFrom(String program) ... {
        ... 
        List<ICommand> commands = parser.parse(program);
        for (ICommand c : commands) {
            turtle.executeCommand(c);
        }
    }
    ... 
}
class Turtle {
    public void moveRight(int steps) {...}
    public void moveTo(int x, int y) {...}
    ...

    public void executeCommand(ICommand cmd) {
        cmd.execute(this);
    }
}

```java
public class Turtle {
    public void moveRight(int steps) {...}
    public void moveTo(int x, int y) {...}
    ...

    public void executeCommand(ICommand cmd) {
        cmd.execute(this);
    }
}
```

Note: The turtle executes any command according to the command supertype...
```java
class CommandRight implements ICommand {
    private final int steps;

    CommandRight(int steps) {
        this.steps = steps;
    }

    @Override
    public void execute(Turtle turtle) {
        turtle.moveRight(steps);
    }
}
```

... and commands select the correct "move" method.
class CommandRight implements ICommand {
    private final int steps;

    CommandRight(int steps) {
        this.steps = steps;
    }

    @Override
    public void execute(Turtle turtle) {
        turtle.moveRight(steps);
    }
}

... and commands select the correct "move" method.

- Elegant way to avoid casting.
- Actual drawing takes place in turtle code.
- See "Design Patterns" book on course website (visitor pattern)
@Test
public void initGame() {
    Player jack = new Player("Jack");
    Player jill = new Player("Jill");
    Game game = new Game(10, new Player[]{ jack, jill });

    assertTrue(game.notOver());
    assertTrue(game.firstSquare().isOccupied());
    assertEquals(1, jack.position());
    // ...
}
JUnit 4: Assertions

- import static org.junit.Assert.*;
  - Provides methods like `assertTrue`, `assertEquals`,...
- Import static allows you to use all the (static) assert methods without having to use a qualified name like:
  - `Assert.assertTrue(...) vs assertTrue(...)"
JUnit 4: Assertions

assertTrue(condition);
assertEquals(expected, actual);

assertNull(object);
assertNotNull(object);

assertSame(expected, actual);
assertNotSame(expected, actual);
...


JUnit 4: Assertions

```java
assertTrue(condition);
assertEquals(expected, actual);

assertNull(object);
assertNotNull(object);

assertSame(expected, actual);
assertNotSame(expected, actual);
...
```

**assert condition;**

Do not use the Java assertions!
JUnit 4: Assertions

```java
assertTrue(jack.position() == 1);
```
JUnit 4: Assertions

```java
assertTrue(jack.position() == 1);
```

Tests failed: 1 of 1 test – 7 ms

- java.lang.AssertionError <3 internal calls>
- at exercise_04.JUnitExamples.testPosition1(JUnitExamples.java:12) <22 internal calls>
JUnit 4: Assertions

```java
assertTrue(jack.position() == 1);
```

What went wrong? Need to check the code...
JUnit 4: Assertions

```java
assertEquals(jack.position(), 1);
```
JUnit 4: Assertions

```java
assertEquals(jack.position(), 1);
```

Tests failed: 1 of 1 test - 10 ms

java.lang.AssertionError:
Expected :0
Actual :1
<Click to see difference>

<1 internal call>
- at org.junit.Assert.failNotEquals<Assert.java:834> <2 internal calls>
- at exercise 04.JUnitExamples.testPosition2<JUnitExamples.java:18> <22 internal calls>
JUnit 4: Assertions

```java
assertEquals(jack.position(), 1);
```

Wrong order: we expect 1, not 0!
JUnit 4: Assertions

```java
assertEquals(1, jack.position());
```
JUnit 4: Assertions

```java
assertEquals(1, jack.position());
```

Tests failed: 1 of 1 test - 9 ms

java.lang.AssertionError:
Expected :1
Actual   :0
<Click to see difference>

<1 internal call>
+ at org.junit.Assert.failNotEquals(Assert.java:834) <2 internal calls>
+ at exercise 04.JUnitTestExamples.testPosition3(JUnitExamples.java:24) <22 internal calls>
JUnit 4: Assertions

```java
assertEquals(1, jack.position());
```

Tests failed: 1 of 1 test – 9 ms

java.lang.AssertionError:
Expected : 1
Actual : 0
<br>Click to see difference>

+-<1 internal call>
  at org.junit.Assert.failNotEquals(Assert.java:834) <2 internal calls>
+- at exercise 04.JUnitExamples.testPosition3(JUnitExamples.java:24) <22 internal calls>

Correct order, but still unclear...
JUnit 4: Assertions

```
assertEquals("Jack is on the first square.", 1, jack.position());
```
JUnit 4: Assertions

```java
assertEquals("Jack is on the first square.", 1, jack.position());
```

Tests failed: 1 of 1 test - 15 ms

```
java.lang.AssertionError: Jack is on the first square.
Expected :1
Actual   :0
<Click to see difference>
```

```
<1 internal call>
+ at org.junit.Assert.failNotEquals(Assert.java:834) <1 internal call>
+ at exercise_04JUnitExamples.testPosition4(UnitExamples.java:30) <22 internal calls>
```
JUnit 4: Assertions

```java
assertEquals("Jack is on the first square.", 1, jack.position());
```

Tests failed: 1 of 1 test - 15 ms

java.lang.AssertionError: Jack is on the first square.
  Expected :1
  Actual :0
  <Click to see difference>

<1 internal call>
+ at org.junit.Assert.failNotEquals(Assert.java:834) <1 internal call>
+ at exercise_04.JUnitExamples.testPosition4(JUnitExamples.java:30) <22 internal calls>

Provide a message (describing the expected outcome).
JUnit 4: Assertions

```java
assertTrue(game.notOver() &&
        game.firstSquare().isOccupied() &&
        (1 == jack.position()) &&
        (1 == jill.position()));
```
JUnit 4: Assertions

```java
assertTrue(game.notOver() &&
            game.firstSquare().isOccupied() &&
            (1 == jack.position()) &&
            (1 == jill.position()));
```

Tests failed: 1 of 1 test – 9 ms

java.lang.AssertionError <3 internal calls>
  at exercise_04JUnitExamples.testPosition5(JUnitExamples.java:38) <22 internal calls>
JUnit 4: Assertions

```java
assertTrue(game.notOver() &&
            game.firstSquare().isOccupied() &&
            (1 == jack.position()) &&
            (1 == jill.position()));
```

Tests failed: 1 of 1 test – 9 ms

java.lang.AssertionError <3 internal calls>
  at exercise_04.JUnitExamples.testPosition5(JUnitExamples.java:38) <22 internal calls>

Which condition made the assertion fail?
JUnit 4: Assertions

assertTrue("Game is not over.", game.notOver());
assertTrue("First square is occupied.",
    game.firstSquare().isOccupied());
assertEquals("Jack is on the first square.",
    1, jack.position());
assertEquals("Jill is on the first square.",
    1, jill.position());
JUnit 4: Assertions

assertTrue("Game is not over.", game.notOver());
assertTrue("First square is occupied.",
           game.firstSquare().isOccupied());
assertEquals("Jack is on the first square.",
             1, jack.position());
assertEquals("Jill is on the first square.",
             1, jill.position());

Tests failed: 1 of 1 test – 9 ms

java.lang.AssertionError: First square is occupied.
<2 internal calls>
at exercise_04.JUnitExamples.testPosition6(JUnitExamples.java:52) <22 internal calls>
JUnit 4: Assertions

```java
assertTrue("Game is not over.", game.notOver());
assertTrue("First square is occupied.",
          game.firstSquare().isOccupied());
assertEquals("Jack is on the first square.",
             1, jack.position());
assertEquals("Jill is on the first square.",
             1, jill.position());
```

Use one condition per assertion!
JUnit 4: Initialization

```java
class Game {
    private Player[] players;
    private int score;

    public Game(int score, Player[] players) {
        this.score = score;
        this.players = players;
    }
}
```
JUnit 4: Initialization

```java
private Game game;
private Player jack, jill;

@Before
public void initNewGame() {
    jack = new Player("Jack");
    jill = new Player("Jill");
    game = new Game(10, new Player[] { jack, jill });
}
```

Use `@Before` to initialize a new game before each test method.
JUnit 4: Setup & Teardown

- @Before
  - Executed before each test method
  - Use for initializing things common to all tests

- @After
  - Clean up after tests
  - Executed even if @Before or @Test fails
JUnit 4: Setup & Teardown

- **@Before**
  - Executed before each test method
  - Use for initializing things common to all tests

- **@After**
  - Clean up after tests
  - Executed even if @Before or @Test fails

- **@BeforeClass (must be static)**
  - Executed once before any @Test method is executed.
  - Use for time intensive tasks

- **@AfterClass (must be static)**
  - Clean up after all tests were run
JUnit 4: Testing Exceptions

- Make sure an exception is thrown
- Useful for making sure errors (e.g. bad input) are actually detected and handled correctly

```java
@Test(expected=IllegalMoveException.class)
public void negativeMoveIsIllegal() throws IllegalMoveException {
    turtle.moveRight(-1);
}
```
JUnit 4: Testing Performance

- Testing execution speed using the `timeout` parameter
- Time in milliseconds (!)

```java
@Test(timeout=10)
public void turtleIsFast() {
    turtle.moveRight(10);
}
```
JUnit 4

- **No** control over order of execution (!)
JUnit 4

- **No** control over order of execution (!)
- Tests should not depend on other tests
JUnit 4

- **No** control over order of execution (!)
- Tests should not depend on other tests
- Do not share data between tests (instance variables, ...)
Writing good tests

- Consider different inputs and parameters
  - Common inputs, values raising exceptions
- Boundary values
  - "off-by-one" errors
- Uncommon values
  - null (if allowed by the contracts)
  - Empty list, array, ...
- Test outputs
  - Returned values and exceptions
- Test side effects
  - State of the system
Writing good tests

- Test classes should thoroughly test a single class
- Write test **during** development
  - You can write them even before you implemented the functionality. You know you’re done, when all tests pass.
- Write tests for every feature
Writing good tests

- As with all code: **make it readable**
  - Proper, self-explaining naming
  - JavaDoc if needed
  - Use the appropriate assertions
  - Keep tests short (few assertions per method)
Mocking

- Some components may be hard to test
  - Non-deterministic results (e.g. a die)
  - Behaviour that is difficult to reproduce (e.g. network failures)
  - Slow or expensive components
  - Incomplete components
Mocking

- Some components may be hard to test
  - Non-deterministic results (e.g. a die)
  - Behaviour that is difficult to reproduce (e.g. network

Let’s just fake it!
Mocking

- Mock objects: Crash test dummies for programmers
- Fake the real thing by manually specifying the behaviour
- Use in place of real objects
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

Create a mock object
→ it can be used like any other object of that type

\(^0\)Code from https://site.mockito.org/#how
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");

Tell the mock object how to behave.
Here: when get(0) is called, return the String "first".

0 Code from https://site.mockito.org/#how
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");

// the following prints "first"
System.out.println(mockedList.get(0));

// the following prints "null",
// because get(999) was not stubbed
System.out.println(mockedList.get(999));

\(^0\)Code from https://site.mockito.org/#how
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");

// the following prints "first"
System.out.println(mockedList.get(0));

// the following prints "null",
// because get(999) was not stubbed
System.out.println(mockedList.get(999));

\(^0\) Code from https://site.mockito.org/#how

Read the documentation!
public ISquare moveAndLand(int moves) {
    assert moves >= 0;
    return game.findSquare(position, moves)
           .landHereOrGoHome();
}

@Test
class GameTest {
    @Test
    public void testMoveAndLand() {
        Game game = new Game(10, new Player("Jack"));
        ISquare start = game.getSquare(2);
        ISquare destination = startSquare.moveAndLand(2);
        assertEquals(game.getSquare(4), destination);
    }
}
public ISquare moveAndLand(int moves) {
    assert moves >= 0;
    return game.findSquare(position, moves)
        .landHereOrGoHome();
}

@Test
public void testMoveAndLand() {
    Game game = new Game(10, new Player("Jack"));
    ISquare start = game.getSquare(2);
    ISquare destination = start.moveAndLand(2);
    assertEquals(game.getSquare(4), destination);
}

Also needs Game.getSquare, Game.findSquare and ISquare.LandHereOrGoHome to work properly!
@Test
def testMoveAndLandOnly():
    game = mock(Game)

    when(game.isValidPosition(anyInt())).thenReturn(true)

    start = mock(Square)
    stop = mock(Square)

    when(game.findSquare(1, 2)).thenReturn(start)
    when(start.landHereOrGoHome()).thenReturn(stop)

    destination = testSquare.moveAndLand(2)
    assertEquals(stop, destination)
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class); // create fake Game
    // tell the game mock what to do if isValidPosition() is called
    when(game.isValidPosition(anyInt())).thenReturn(true);

    ISquare testSquare = new Square(game, 1);
    ISquare start = mock(Square.class);
    ISquare stop = mock(Square.class);

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

    ISquare destination = testSquare.moveAndLand(2);
    assertEquals(stop, destination);
}
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class); // create fake Game
    // tell the game mock what to do if isValidPosition() is called
    when(game.isValidPosition(anyInt())).thenReturn(true);

    // create target on which we want to test moveAndLand()
    ISquare testSquare = new Square(game, 1);
    ISquare start = mock(Square.class); // mocks for findSquare()
    ISquare stop = mock(Square.class); // and landHereOrGoHome()

    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

    ISquare destination = testSquare.moveAndLand(2);
    assertEquals(stop, destination);
}
@Test
public void testMoveAndLandOnly() {
    Game game = mock(Game.class); // create fake Game
    // tell the game mock what to do if isValidPosition() is called
    when(game.isValidPosition(anyInt())).thenReturn(true);

    // create target on which we want to test moveAndLand()
    ISquare testSquare = new Square(game, 1);
    ISquare start = mock(Square.class); // mocks for findSquare()
    ISquare stop = mock(Square.class); // and landHereOrGoHome()

    // mock behaviour of game and start
    when(game.findSquare(1, 2)).thenReturn(start);
    when(start.landHereOrGoHome()).thenReturn(stop);

    ISquare destination = testSquare.moveAndLand(2);
    assertEquals(stop, destination); // actual test for testSquare
}
Exercise 4

- Test Snakes & Ladders (our implementation)
- Fix exercise 3 if necessary
- Use JUnit and Mockito
- Write good tests with code coverage and qualitative criteria in mind
- See exercise_04.md for more details