Debugging and Sokoban Intro

Marcel Zauder

February 26, 2020

Exercise 04

Tasks

1. Test **Game#play(IDie)** with two different IDies: one mocked by hand, one mocked using Mockito
2. Compare these two approaches
3. Test all **Squares** in the game, use Mockito to mock unrelated objects
4. Add a new square: ScrambleUpSquare, test it
5. Cover the code

In **Game.java**:

```java
public void play(Die die) {
    ...
}
```

Change to:

```java
public void play(IDie die) {
    ...
}
```

Then test with:

```
@BeforeEach
public void initializeTest() {
    ...
    testGame = new Game(GAMESIZE,players,DIESIDES);
    IDie mockDie = mock(IDie.class);
    when(mockDie.roll()).thenReturn(1, 2, 3, 4, ...);
    testGame.play(mockDie);
}
```

Another mocking example:

```
@Test
public void testPlayerSwapOnly(){
    Game mkGame = mock(Game.class);
    FirstSquare mkFirstSquare = mock(FirstSquare.class);
    LastSquare mkLastSquare = mock(LastSquare.class);
    when(mkGame.firstSquare()).thenReturn(mkFirstSquare);
    when(mkGame.getSquare(2)).thenReturn(mkLastSquare);
    when(mkLastSquare.position()).thenReturn(2);
    Player Jack = new Player("Jack");
    Jack.joinGame(mkGame);
    Jack.swap(mkLastSquare);
    assertEquals(2,Jack.position());
}
```

The *swap* behaviour is implemented in the **Player**, so we mock the **Game** and the **Squares**.

Exercise 04

Mocking Tips

1. Don't mock the object that you're trying to test - that defeats the purpose of the test
2. Try and keep your tests simple (but still thorough!), so you have to mock as little behaviour as possible
3. The **When/Then Cookbook** might help you: https://www.baeldung.com/mockito-behavior

Code Coverage

1. No need to get 100% coverage
2. For every line/method, you should either cover it, or explain **why** you didn't cover it (e.g. "not covering trivial getters/setters")

Debugging

1. **Breakpoint.** Tell the debugger to halt here, as soon as it gets to this line. Add and remove breakpoints by left-clicking next to a line number.
2. **Current Position.** Program is currently halted on this line, the line hasn't yet been executed.
3. **Local Variables.** An overview of the current variable values.
4. **Call Stack.** The current method call stack.
5. **Navigation Tools.** Control where to go next (step over this line, step into it, etc.)
6. **Stop.** Stop the program, stop debugging.

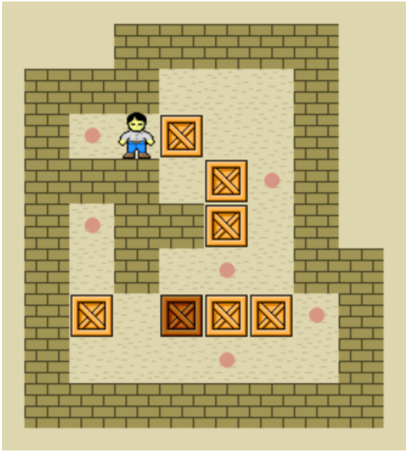
Debugging

7. **Continue.** Continue running this program, either until it exists, or until it hits the next breakpoint.
8. **Debug Button.** Click this to run the program in debug mode. This will halt the program as soon as it hits a breakpoint. You can also debug a program by right-clicking on a main class, a test class or a test method, and clicking on "Debug As". We have already done this here, to get to this view.
9. **Java View vs. Debugger View**. Debug view (right button) is this view, Java view (left button) is the view you normally use when coding.

Debugging

Live DEMO:
Debugging the Turtle Game

Sokoban



Sokoban

Definition

- starting with the same number of boxes and goal tiles/storage locations
- player can go up, down, left and right
- the boxes need to be pushed on the goal tiles
  - the boxes can be placed on any storage location
  - boxes on storage locations can still be moved
- boxes may not be pushed into other boxes or walls
- boxes cannot be pulled
- the puzzle is solved when all boxes are on a storage location

Sokoban (Notation)

#	:	Wall
_	:	empty tile
P	:	Player
G	:	Goal tile/storage location
B	:	Box

	#	#	#	#	
#	#		G	#	#
#	P		B	G	#
#		B			#
#	#				#
#	#	#	B	#	#
#					#
#				B	#
#	G	#		G	#
#	#	#	#	#	#

Your Task

Tasks

1. Set up the game representation (implement classes like **Game**, **Player**, **Tile** etc.)
2. Write a parser that reads the board specification. (There are already predefined levels in the 'levels' folder)
3. Write an ASCII renderer which prints any state of the gameboard (Use 'System.out.print' method)
4. Write tests so that the predefined levels are parsed correctly. There is also another game file called 'fail.sok' that contains 2 boxes and 1 goal tile which therefore should not be accepted by the parser (You can add more levels if you like)
5. Tag the solution with 'sokoban1' and make sure the tag is pushed to the remote repository.