AARHUS UNIVERSITET

Microservices and DevOps

DevOps and Container Technology

Agile Development – an Opiniated Guide

Henrik Bærbak Christensen

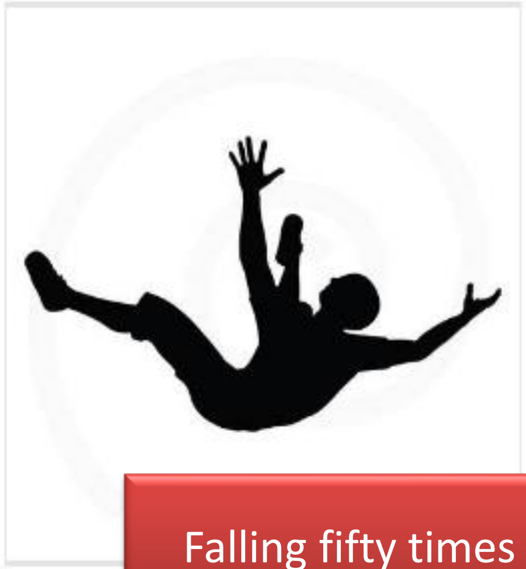
All seasoned developers...

- ... And I do not spend all my time developing, so...

- *Who am I to teach you how to develop software?*
- However, often I see **this:**
  - *I will just code these two SQL statements, add a table, aah – OK I also introduce the table init seq; wait, I have this code somewhere; Yeah, I will just generalize, and update all call sites, Hey – this must be a bug here...*

That is – *take large steps*

- When asked, people says
  - *It is much faster!*



Falling fifty times is *not* faster!

AARHUS UNIVERSITET

Take small steps...

- One of the four *test-driven development* principles:
  - *Take small steps*
- *Use the ladder, not the vaulting pole. – Henrik Bærbak*

AARHUS UNIVERSITET

Not Unknown Phenomenon

- *Build support code to help you build **strong** code!*
  - *Like taking the ladder/stairs – slow but steady, and you do not fall down and hurt yourself (too badly)!*

Scaffolding Code

AARHUS UNIVERSITET

The Code Perspective

Enough Metaphoric Speak

AARHUS UNIVERSITET

A Developer’s Scaffolding

- DevOps relies on *testing!* ***Get used to it!***
- ... And at all levels...
- Testing is *scientific process:*
  - *Hypothesis*
    - *Write test*
  - *Experiment*
    - *Execute*
  - *Conclusion*
    - *Pass/Fail*

Definition: **Unit test**

Unit testing is the process of executing a software unit in isolation in order to find defects in the unit itself.

Definition: **Integration test**

Integration testing is the process of executing a software unit in collaboration with other units in order to find defects in their interactions.

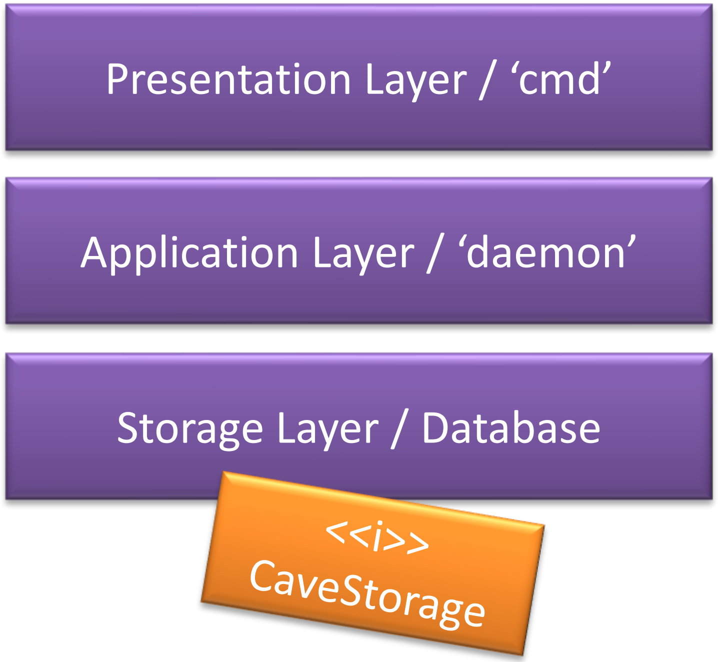
Definition: **System test**

System testing is the process of executing the whole software system in order to find deviations from the specified requirements.

From SkyCave

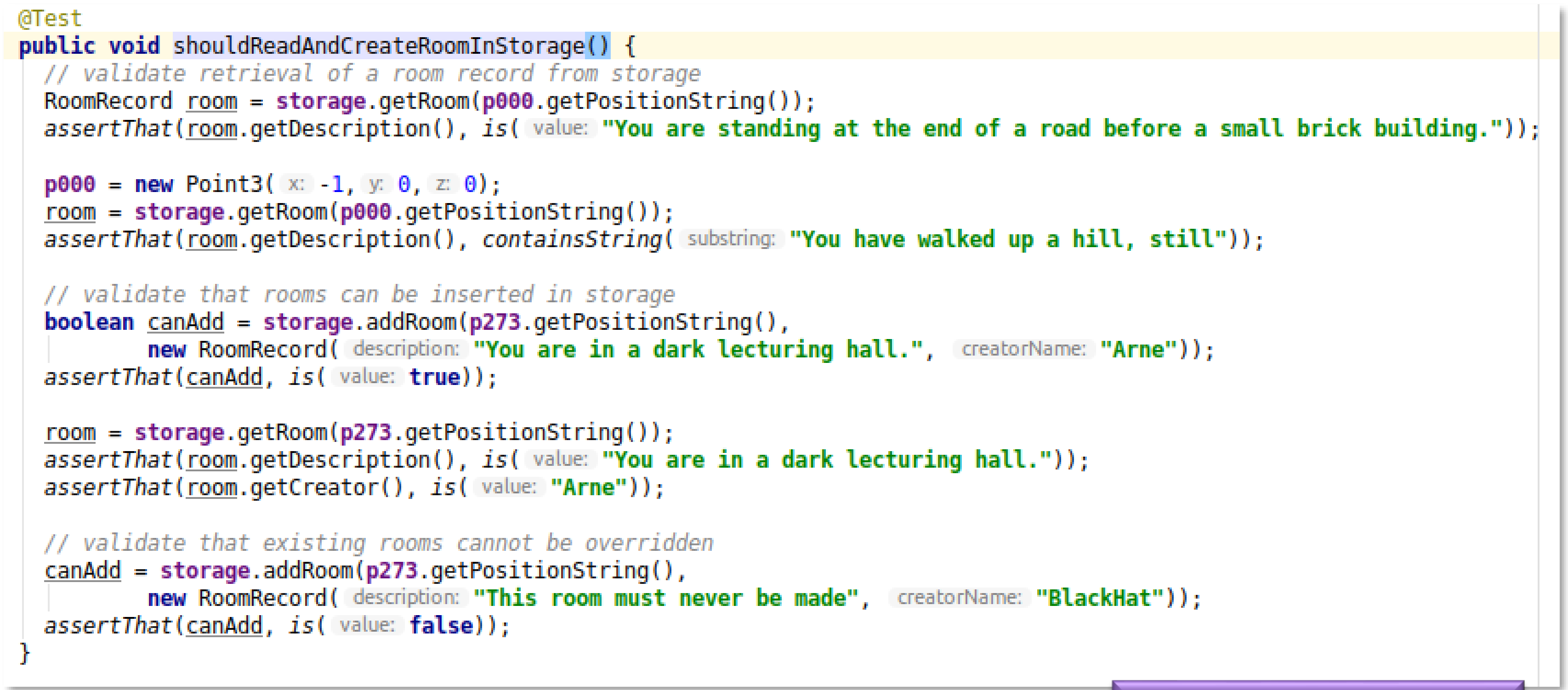
- Our present ‘monolith’ SkyCave architecture

- Starting from the Bottom: The Storage layer...



Make Tests!

- Hypothesis: *Storage can Create and Read rooms*

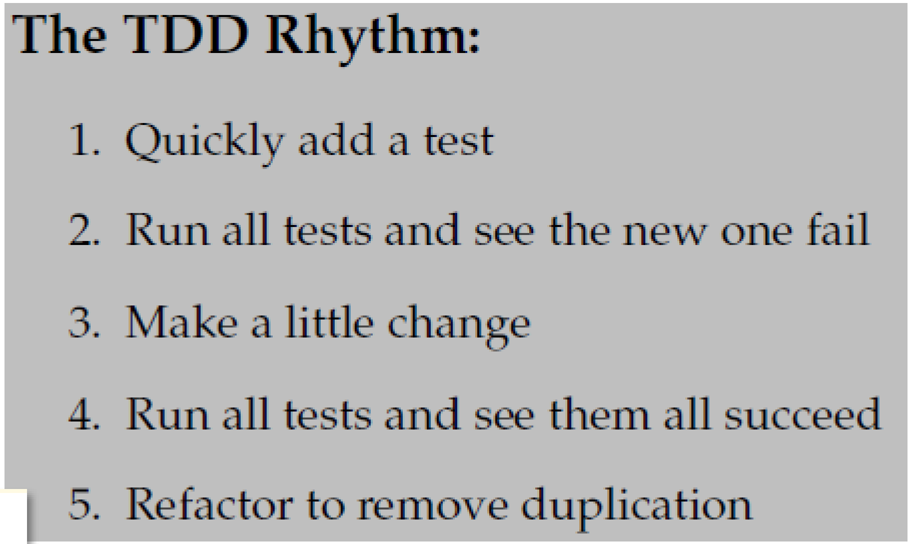


```
@Test
public void shouldReadAndCreateRoomInStorage() {
  // validate retrieval of a room record from storage
  RoomRecord room = storage.getRoom(p000.getPositionString());
  assertThat(room.getDescription(), is( value: "You are standing at the end of a road before a small brick building."));

  p000 = new Point3( x: -1, y: 0, z: 0);
  room = storage.getRoom(p000.getPositionString());
  assertThat(room.getDescription(), containsString( substring: "You have walked up a hill, still"));

  // validate that rooms can be inserted in storage
  boolean canAdd = storage.addRoom(p273.getPositionString(),
          new RoomRecord( description: "You are in a dark lecturing hall.",  creatorName: "Arne"));
  assertThat(canAdd, is( value: true));

  room = storage.getRoom(p273.getPositionString());
  assertThat(room.getDescription(), is( value: "You are in a dark lecturing hall."));
  assertThat(room.getCreator(), is( value: "Arne"));

  // validate that existing rooms cannot be overridden
  canAdd = storage.addRoom(p273.getPositionString(),
          new RoomRecord( description: "This room must never be made",  creatorName: "BlackHat"));
  assertThat(canAdd, is( value: false));
}
```

TestRoomPlayerStorage

What??? You have no DB!

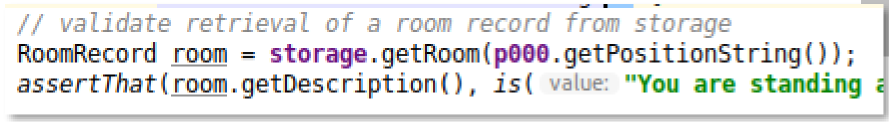
- I am a TDD guy
  - Top-down approach
  - *Code the API before the implementation!!!*



The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

- So I *quickly add test*



```
// validate retrieval of a room record from storage
RoomRecord room = storage.getRoom(p000.getPositionString());
assertThat(room.getDescription(), is( value: "You are standing a
```

- See it fail, and then...
  - *I do not start a Microsoft SQL Server 2017 and start hacking SQL statements... Why? Because my tests would not be automatic!*

AARHUS UNIVERSITET

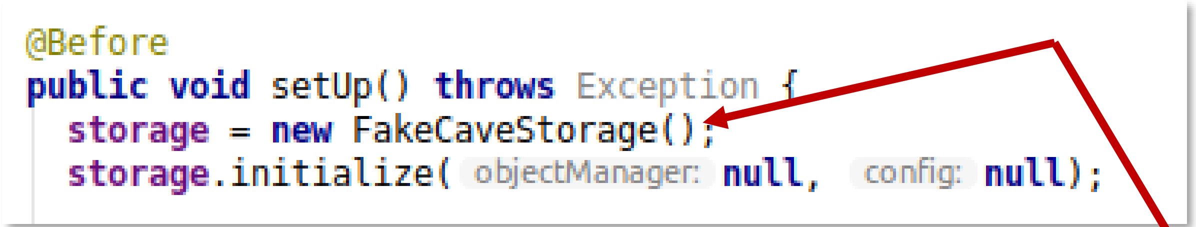
Unit Tests are Fast!

- Testing against a real DB is an *Integration test*. Requires a lot of setup to start a DB, wipe its contents, shut it down again.

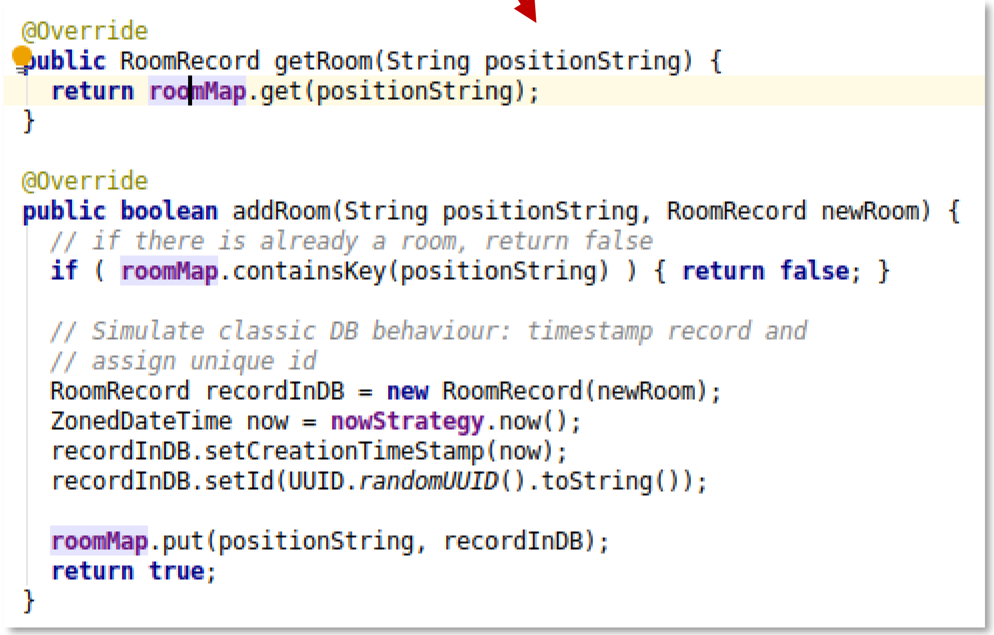
- ***It is to slow when you run your unit tests twice every minute!!! And you do, right?***

- So I use *Test doubles* – replacements for the *real depended-on-unit* that acts like it but are fast, lightweight, and so easy to code that they contain no errors!

Fake Object



```
@Before
public void setUp() throws Exception {
  storage = new FakeCaveStorage();
  storage.initialize( objectManager: null,  config: null);
```



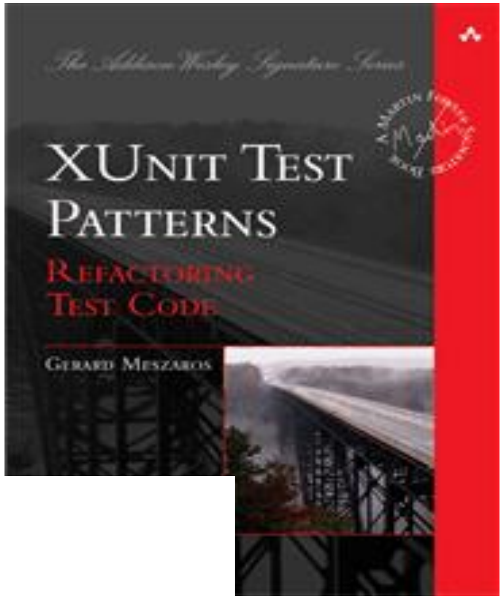
```
@Override
public RoomRecord getRoom(String positionString) {
  return roomMap.get(positionString);
}

@Override
public boolean addRoom(String positionString, RoomRecord newRoom) {
  // if there is already a room, return false
  if ( roomMap.containsKey(positionString) ) { return false; }

  // Simulate classic DB behaviour: timestamp record and
  // assign unique id
  RoomRecord recordInDB = new RoomRecord(newRoom);
  ZonedDateTime now = nowStrategy.now();
  recordInDB.setCreationTimeStamp(now);
  recordInDB.setId(UUID.randomUUID().toString());

  roomMap.put(positionString, recordInDB);
  return true;
}
```

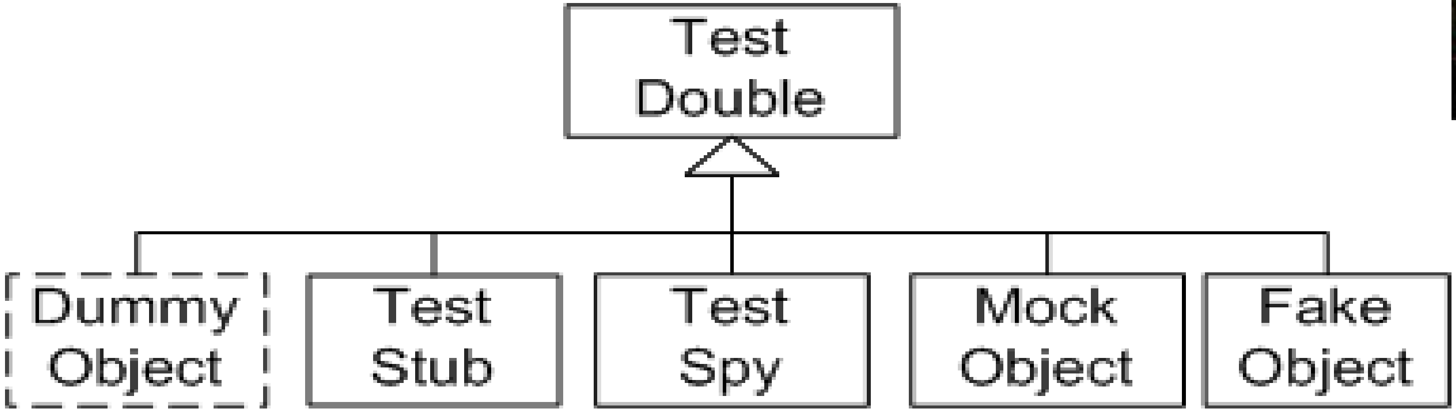
Use a HashMap or ? Or ? as DB table!

AARHUS UNIVERSITET

Meszaros

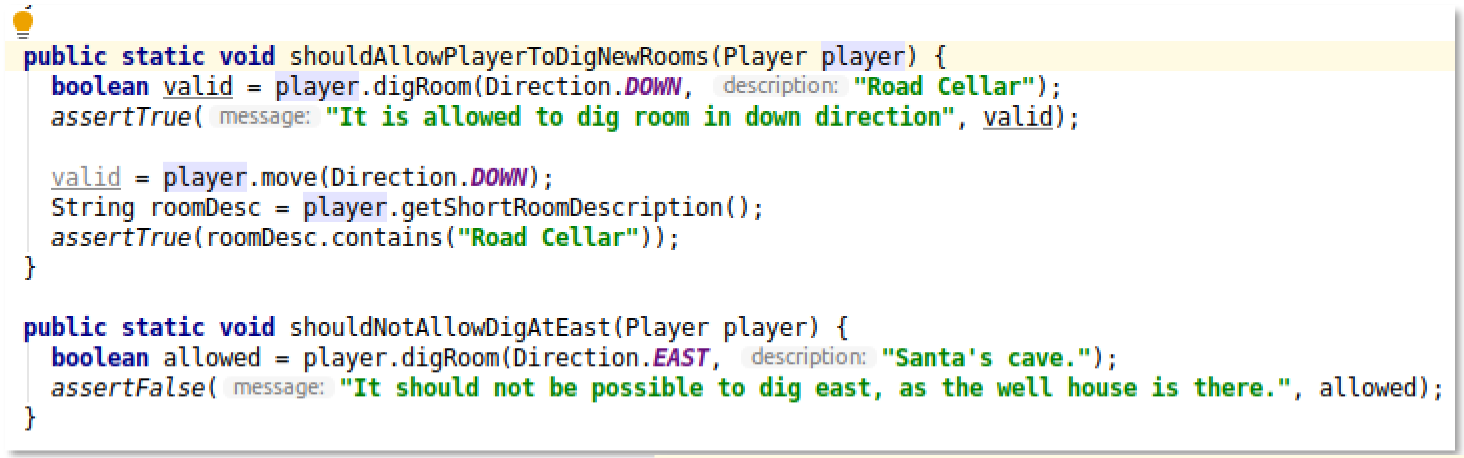
- Test doubles makes your software testable, because they
  - Break dependencies on ‘heavy objects’
    - Databases, remote services,
  - Breaks dependency chains to isolate the *unit-under-test*



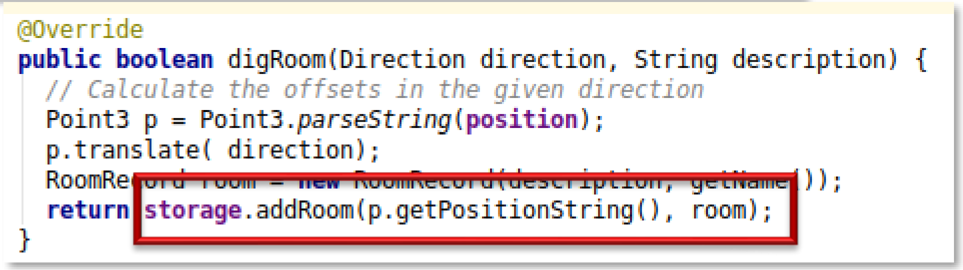


Testing at the Application Layer

- Hypothesis: *PlayerServant can dig new rooms*
  - Fast development, as no setup is required



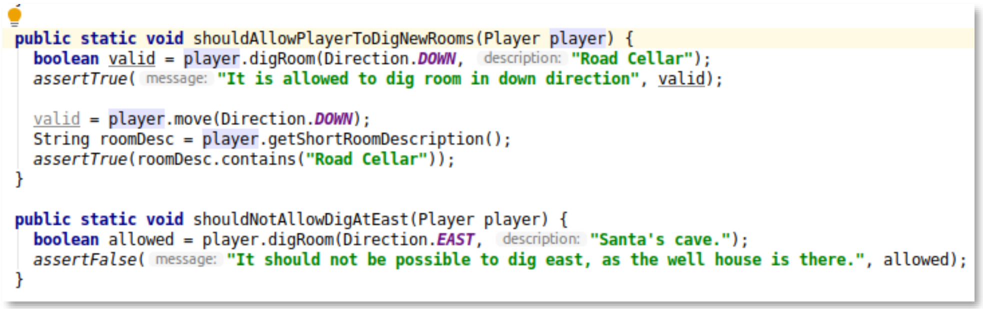
```
public static void shouldAllowPlayerToDigNewRooms(Player player) {
  boolean valid = player.digRoom(Direction.DOWN,  description: "Road Cellar");
  assertTrue( message: "It is allowed to dig room in down direction", valid);

  valid = player.move(Direction.DOWN);
  String roomDesc = player.getShortRoomDescription();
  assertTrue(roomDesc.contains("Road Cellar"));
}

public static void shouldNotAllowDigAtEast(Player player) {
  boolean allowed = player.digRoom(Direction.EAST,  description: "Santa's cave.");
  assertFalse( message: "It should not be possible to dig east, as the well house is there.", allowed);
}
```



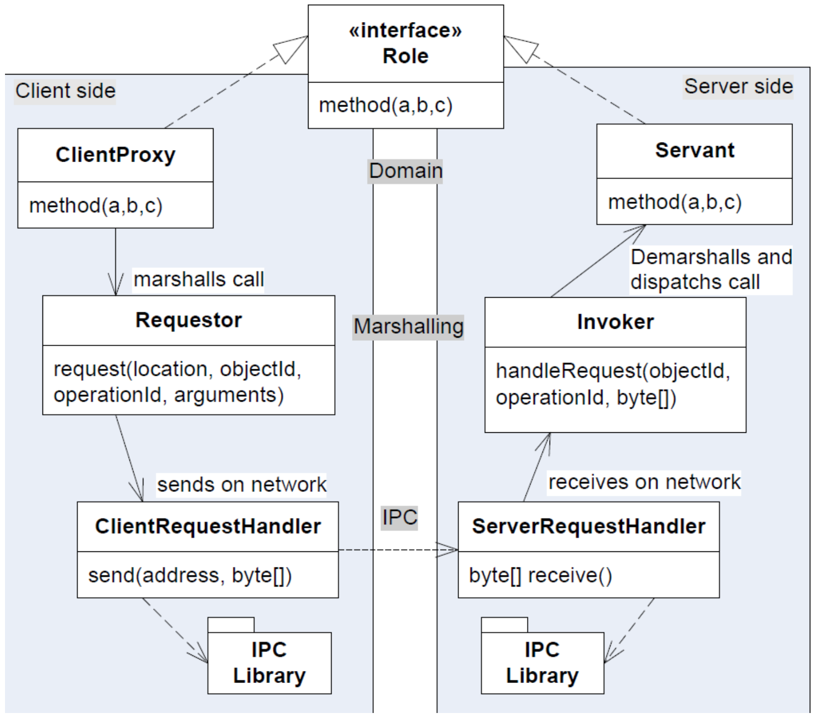
```
@Override
public boolean digRoom(Direction direction, String description) {
  // Calculate the offsets in the given direction
  Point3 p = Point3.parseString(position);
  p.translate( direction);
  RoomRecord room = new RoomRecord(description, getName());
  return storage.addRoom(p.getPositionString(), room);
}
```

Other Example

- SkyCave is a distributed client-server system...
- Hypothesis: *PlayerProxy can dig new rooms*



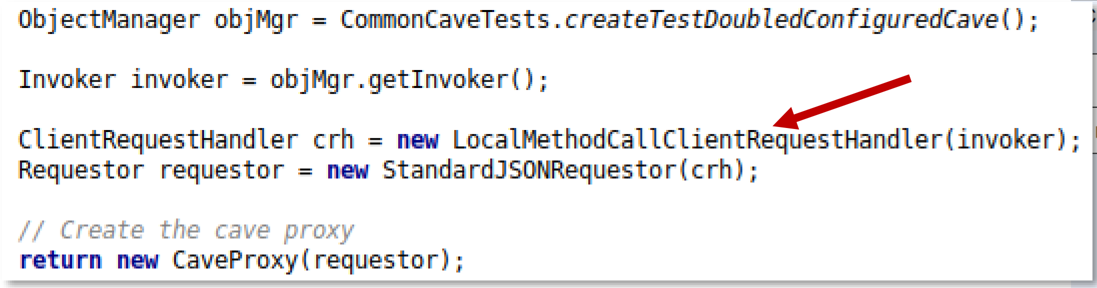
```
public static void shouldAllowPlayerToDigNewRooms(Player player) {
  boolean valid = player.digRoom(Direction.DOWN, description: "Road Cellar");
  assertTrue( message: "It is allowed to dig room in down direction", valid);

  valid = player.move(Direction.DOWN);
  String roomDesc = player.getShortRoomDescription();
  assertTrue(roomDesc.contains("Road Cellar"));
}

public static void shouldNotAllowDigAtEast(Player player) {
  boolean allowed = player.digRoom(Direction.EAST, description: "Santa's cave.");
  assertFalse( message: "It should not be possible to dig east, as the well house is there.", allowed);
}
```



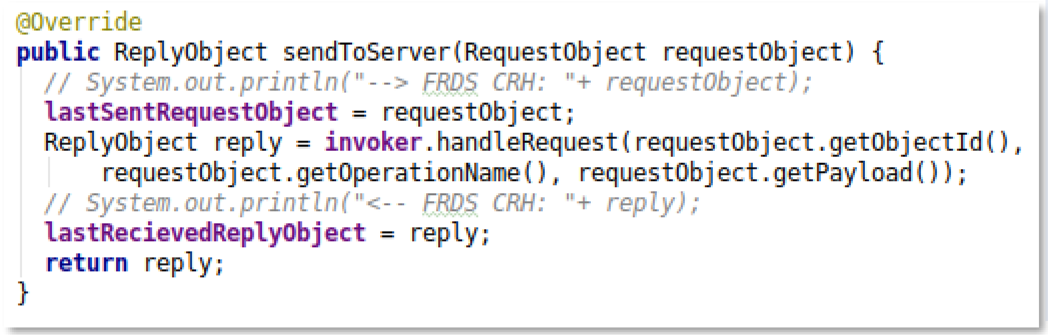
- *Actually, it is the same test-case!*
  - Refactored into a ‘CommonPlayerTest’

Fake Object IPC

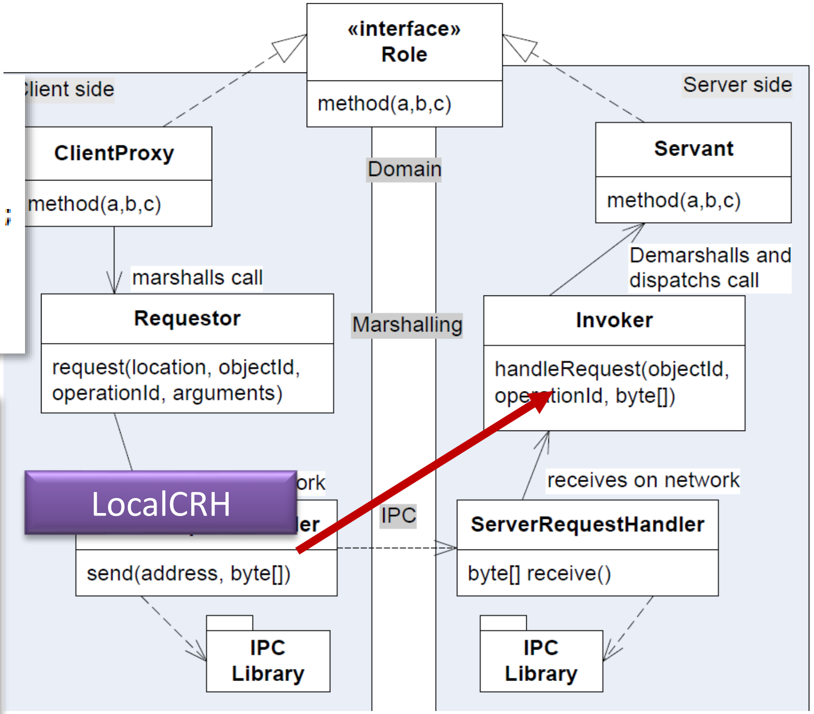
- IPC purpose: Get data from client to server
  - We can use a test double instead!



```
ObjectManager objMgr = CommonCaveTests.createTestDoubledConfiguredCave();

Invoker invoker = objMgr.getInvoker();

ClientRequestHandler crh = new LocalMethodCallClientRequestHandler(invoker);
Requestor requestor = new StandardJSONRequestor(crh);

// Create the cave proxy
return new CaveProxy(requestor);
```

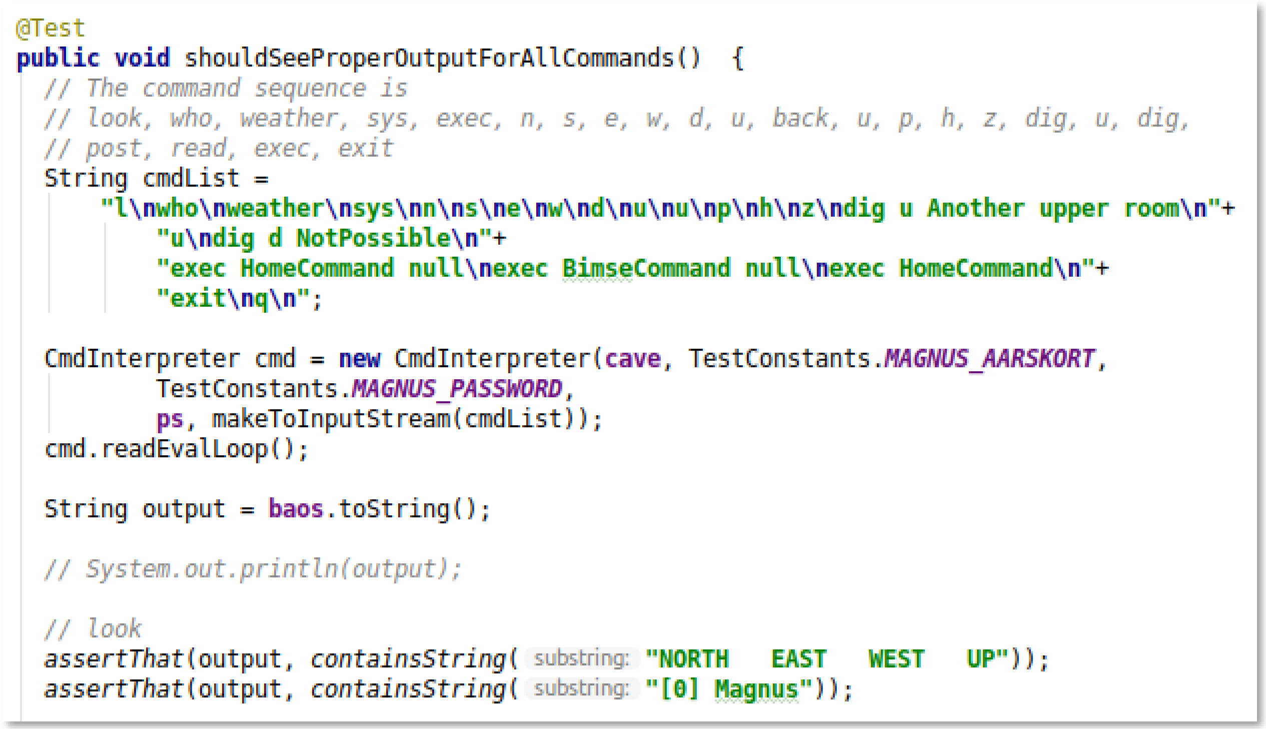


```
@Override
public ReplyObject sendToServer(RequestObject requestObject) {
  // System.out.println("--> FRDS CRH: "+ requestObject);
  lastSentRequestObject = requestObject;
  ReplyObject reply = invoker.handleRequest(requestObject.getObjectId(),
      requestObject.getOperationName(), requestObject.getPayload());
  // System.out.println("<-- FRDS CRH: "+ reply);
  lastRecievedReplyObject = reply;
  return reply;
}
```



Testing at the Presentation Layer

- We can even move the testing to the UI layer now
  - ”Automated System Testing”



```java
@Test
public void shouldSeeProperOutputForAllCommands()  {
  // The command sequence is
  // look, who, weather, sys, exec, n, s, e, w, d, u, back, u, p, h, z, dig, u, dig,
  // post, read, exec, exit
  String cmdList =
      "l\nwho\nweather\nsys\nn\ns\ne\nw\nd\nu\nu\np\nh\nz\ndig u Another upper room\n"+
          "u\ndig d NotPossible\n"+
          "exec HomeCommand null\nexec BimseCommand null\nexec HomeCommand\n"+
          "exit\nq\n";

  CmdInterpreter cmd = new CmdInterpreter(cave, TestConstants.MAGNUS_AARSKORT,
          TestConstants.MAGNUS_PASSWORD,
          ps, makeToInputStream(cmdList));
  cmd.readEvalLoop();

  String output = baos.toString();

  // System.out.println(output);

  // look
  assertThat(output, containsString( substring: "NORTH   EAST   WEST   UP"));
  assertThat(output, containsString( substring: "[0] Magnus"));
```

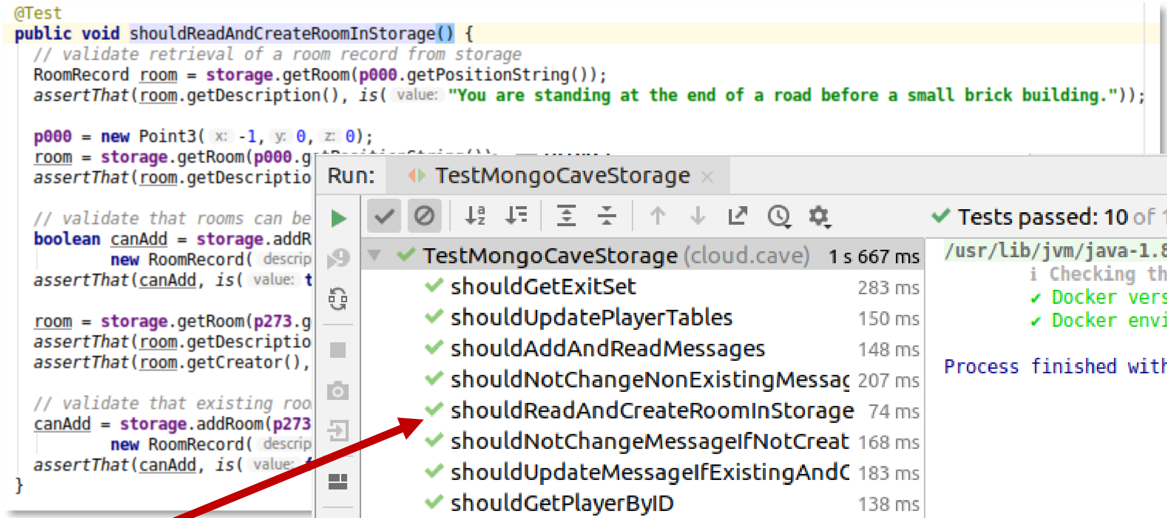
Compare...

- Comparing to a non-test-doubled architecture
  - Start the database, wipe its contents, run the init script to fill in the default tables, update the server's config file with the proper (ip,port) of the database, start the server, start the daemon, try to dig a new room to the north, validate that it worked ok, ...

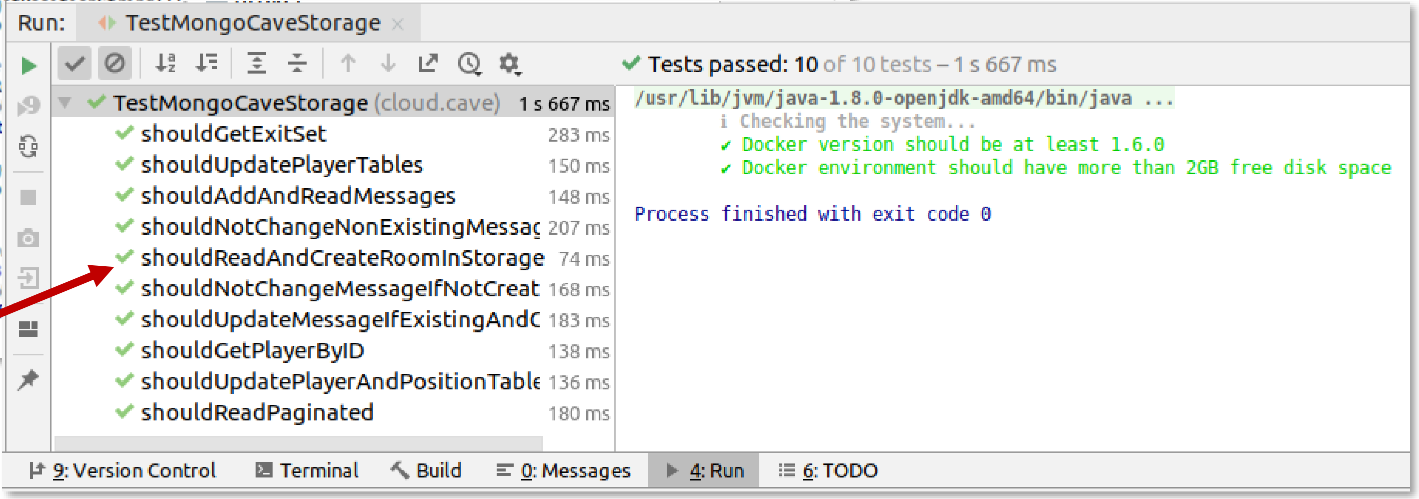
- Large Aarhus based company war-story
  - 1.000 hours of manual test before release...

And that is not all...

- Remember the low level tests test at the storage level?
  - Hypothesis: *Storage can Create and Read rooms*
  - Reuse them for *service tests* using the **real database!**



```
@Test
public void shouldReadAndCreateRoomInStorage() {
  // validate retrieval of a room record from storage
  RoomRecord room = storage.getRoom(p000.getPositionString());
  assertThat(room.getDescription(), is( value: "You are standing at the end of a road before a small brick building."));

  p000 = new Point3( x: -1, y: 0, z: 0);
  room = storage.getRoom(p000.g
  assertThat(room.getDescriptio

  // validate that rooms can be
  boolean canAdd = storage.addR
          new RoomRecord( descrip
  assertThat(canAdd, is( value: t

  room = storage.getRoom(p273.g
  assertThat(room.getDescriptio
  assertThat(room.getCreator(),

  // validate that existing roo
  canAdd = storage.addRoom(p273
          new RoomRecord( descrip
  assertThat(canAdd, is( value:
}
```



```
✓ Tests passed: 10 of 10 tests – 1 s 667 ms

✓ TestMongoCaveStorage (cloud.cave)          1 s 667 ms
    ✓ shouldGetExitSet                       283 ms
    ✓ shouldUpdatePlayerTables               150 ms
    ✓ shouldAddAndReadMessages               148 ms
    ✓ shouldNotChangeNonExistingMessag       207 ms
    ✓ shouldReadAndCreateRoomInStorage        74 ms
    ✓ shouldNotChangeMessageIfNotCreat       168 ms
    ✓ shouldUpdateMessageIfExistingAndC      183 ms
    ✓ shouldGetPlayerByID                    138 ms
    ✓ shouldUpdatePlayerAndPositionTable     136 ms
    ✓ shouldReadPaginated                    180 ms
```

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
        i Checking the system...
        ✔ Docker version should be at least 1.6.0
        ✔ Docker environment should have more than 2GB free disk space

Process finished with exit code 0
```

Cost Benefit Analysis

- Costs:
  - I have to *program to an interface*
    - To have two or more implementations of the CaveStorage
  - I do have to develop and maintain the ‘FakeCaveStorage’
    - 12-15 methods with a lot of hashMap manipulations (260 LOC)
  - Dependency Injection is required
    - To control which implementation of CaveStorage to use
- Benefits
  - I am **forced** to *program to an interface* ☺
  - I am **forced** to *use dependency injection* ☺

Cost Benefit Analysis

- Benefits ...
  - I have most of my code under fast test execution
  - I can reuse my test cases to develop real-service code in the service tests

The Ladder, not the Vaulting Pole

- Take small steps by
  - Building test cases to ensure you do not fall
  - External services/modules replaced by test doubles
    - That are covered by test cases
  - ... That you reuse for the real interaction code...