

Software Testing and Quality Engineering

Maurício Aniche and Arie van Deursen

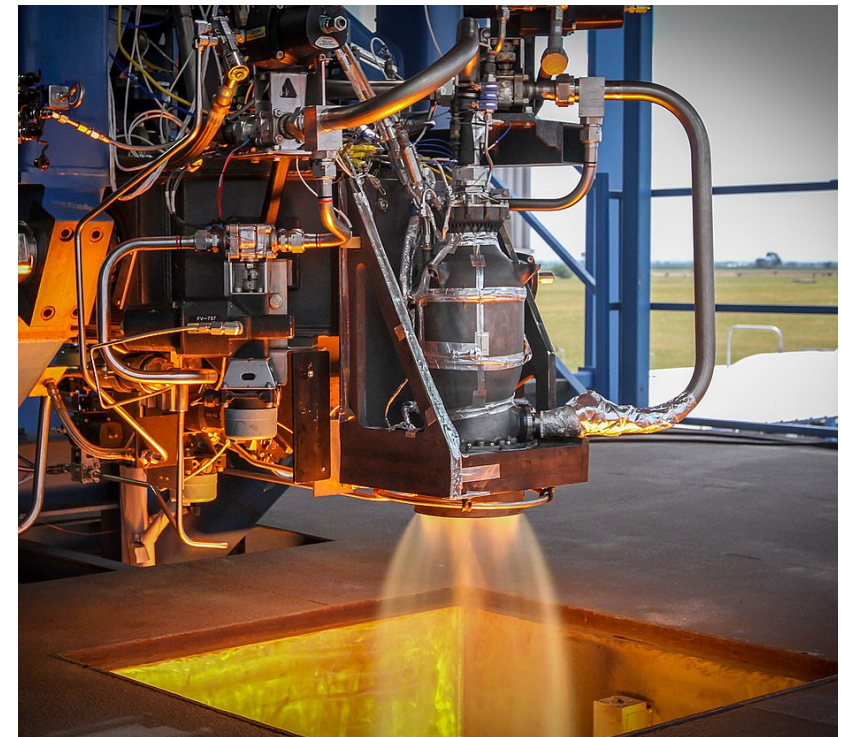


Why do we test?

1. To make informed decisions about expected quality when releasing software
2. To guide requirements elicitation, by identifying (simple to understand) execution scenarios.
3. To guide the design of the software that we create.

How we test our software: *Test Execution*

- In modern software development, we *release often*
- Releasing often implies testing often:
 - Automate test execution as much as possible
- Build a testing system aimed at
 - exercising the system under test
 - and verifying the observed behavior



How do we test our software: *Test Design*

- Decide which (of the infinitely many possible) test cases to create
 - Maximize *information gain*
 - Minimize *cost*
- A test strategy:
 - A systematic approach to arrive at test cases
 - Targeting specific types of faults
 - Until a given adequacy criterion is achieved
- *Test design begins at the start of your project*



What do we test?

Test Levels

- Different levels of granularity
- Unit testing
- Integration testing
- System testing
- ...

Test Types

- Different objectives
- Functionality (old / new)
- Security
- Performance
- ...

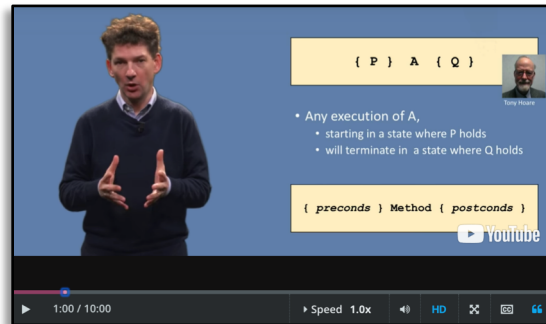
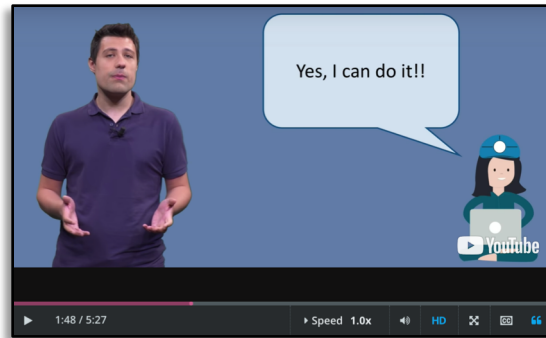
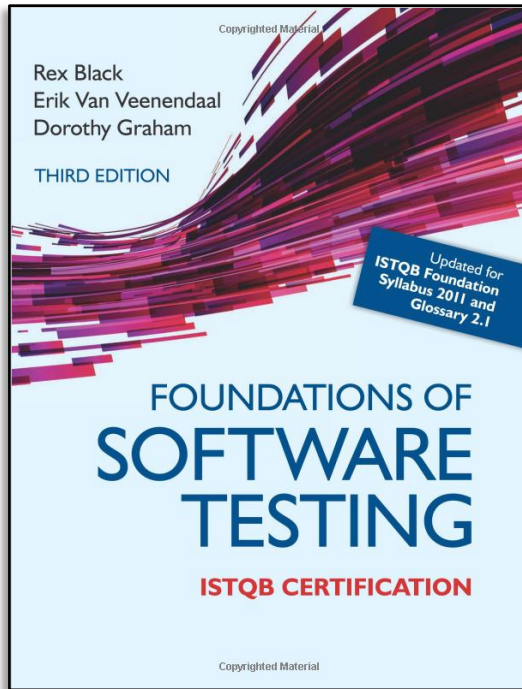
Learning Objectives

- **Knowledge level:**
 - Essential test methods, tools, techniques, ...
- **Application level:**
 - Actually use selected test techniques
- **Evaluation level:**
 - Decide what's useful in your project
 - Criticize, analyze, investigate, reflect, research, innovate, ...

Reliable Knowledge in Software Testing?

- Software testing is all about making trade-offs
 - Becomes easier with experience!
- Strategies, patterns, and processes are codified experience
 - You will need to know them!
- Our body of knowledge grows as reflective engineers / researchers:
 - Codify their knowledge and pass it on
 - Analyze successes and failures and report on those
 - Propose, implement, and evaluate novel testing strategies

Course Material



21 DEC 2012 **Line Coverage: Lessons from JUnit**
 posted in **Development** by **Arie van Deursen**

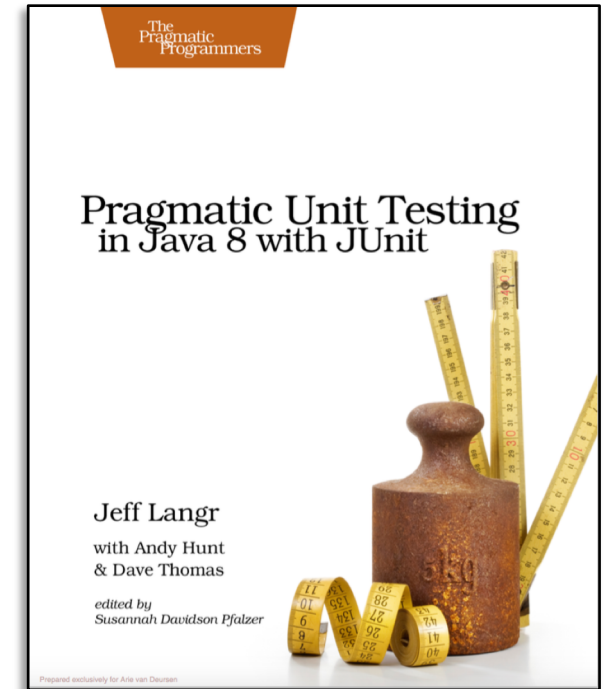
In unit testing, achieving 100% statement coverage is not realistic. But what percentage would good testers get? Which cases are typically not included? Is it important to actually measure coverage?

To answer questions like these, I took a look at the test suite of **JUnit** itself. This is an interesting case, since it is created by some of the best developers around the world, who care a lot about testing. If they decide not to test a given case, what can we learn from that?

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	221	84%	81%	1,727
JUnit.extensions	6	82%	87%	1,25
JUnit.framework	17	79%	90%	1,605
JUnit.runner	3	49%	41%	2,225
JUnit.textui	2	76%	76%	1,686
org.junit	14	80%	73%	1,655
org.junit.experimental	2	93%	83%	1,5
org.junit.experimental.categories	5	100%	100%	3,357
org.junit.experimental.main	8	80%	80%	1,569
org.junit.experimental.results	6	90%	87%	1,222
org.junit.experimental.runners	1	100%	N/A	1
org.junit.experimental.theories	14	96%	88%	1,674
org.junit.experimental.theories.internal	5	89%	92%	2,209
org.junit.experimental.theories.support	2	100%	100%	2
org.junit.internal	11	94%	94%	1,947
org.junit.internal.builders	8	98%	92%	2
org.junit.internal.matchers	4	75%	0%	1,391
org.junit.internal.requests	3	96%	100%	1,429
org.junit.internal.runners	18	73%	63%	2,155
org.junit.internal.runners.model	3	100%	100%	1,5
org.junit.internal.runners.rules	1	100%	100%	2,111
org.junit.internal.runners.statements	7	97%	100%	2
org.junit.matchers	1	9%	N/A	1
org.junit.rules	20	89%	96%	1,444
org.junit.runner	12	85%	88%	1,378
org.junit.runner.manipulation	9	89%	77%	1,632
org.junit.runner.notification	12	100%	100%	1,162
org.junit.runners	16	96%	90%	1,737
org.junit.runners.model	11	82%	73%	1,918

Report generated by [Cobertura](#) 1.9.4.1 on 12/22/12 2:25 PM.

Coverage of JUnit as measured by Cobertura



Lectures

- At mixed times, in the large aula
- Please do not use your electronic devices during the lecture
- Questions / interaction:
 - Hard in the aula – (also) use the break
 - Discussion forum “Lecture Q&A”
 - One topic per lecture.



← → ↻ 🔒 <https://brightspace.tudelft.nl/d2l/le/144558/discussions/List> ☆ ⓘ | 🗲 ⋮

Lecture Q&A ▾

Forum to post questions about the content of the lectures in case you could not ask them during the lecture. As teachers and TAs we will try to address them either here in the forum or in one of the next lectures.

If, as a student, you know the answer, also please offer it!

Topic	Threads	Posts	Last Post
Lecture 1: Introduction ▾	0	0	

Questions for the introductory lecture on April 23.

15 Lectures!

23/4: Introduction

26/4: Foundations

30/4: Functional testing

01/5: Model-based testing

03/5: Structural testing

07/5: Exploratory testing
(Jan Jaap Cannegieter)

10/5: Testability, mock objects

13/5: Software security 1
(Sicco Verwer)

15/5: Static/dynamic analysis
(Azqa Nadeem)

17/5: Test code quality

28/5: Web testing
(Frank Mulder)

07/6: Design by contract

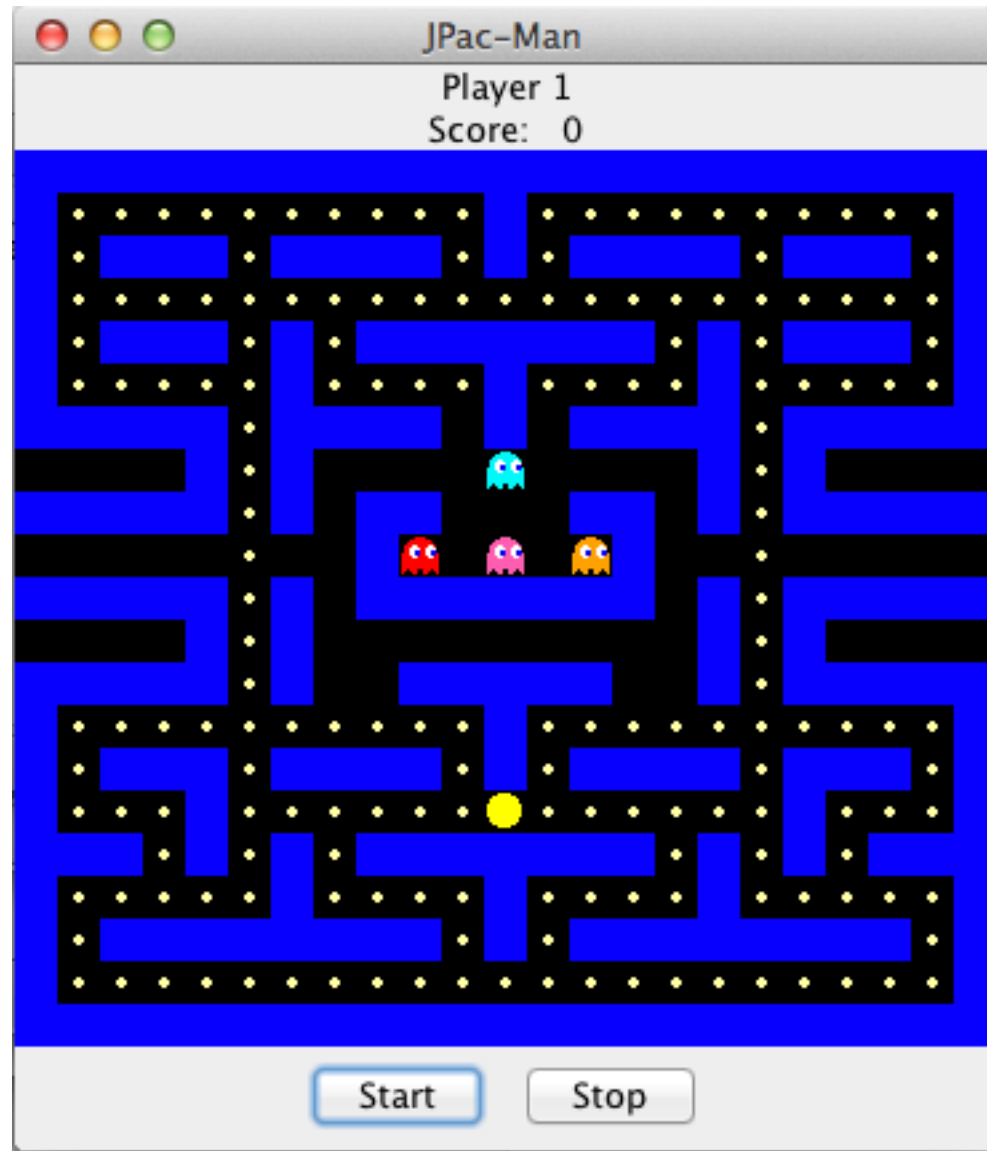
11/6: Search-based testing
(Annibale Panichella)

14/6: Breaking changes in OS
(Tim van der Lippe)

20/6: Testing at Spring
(Stéphane Nicoll)

Learning in the Labwork

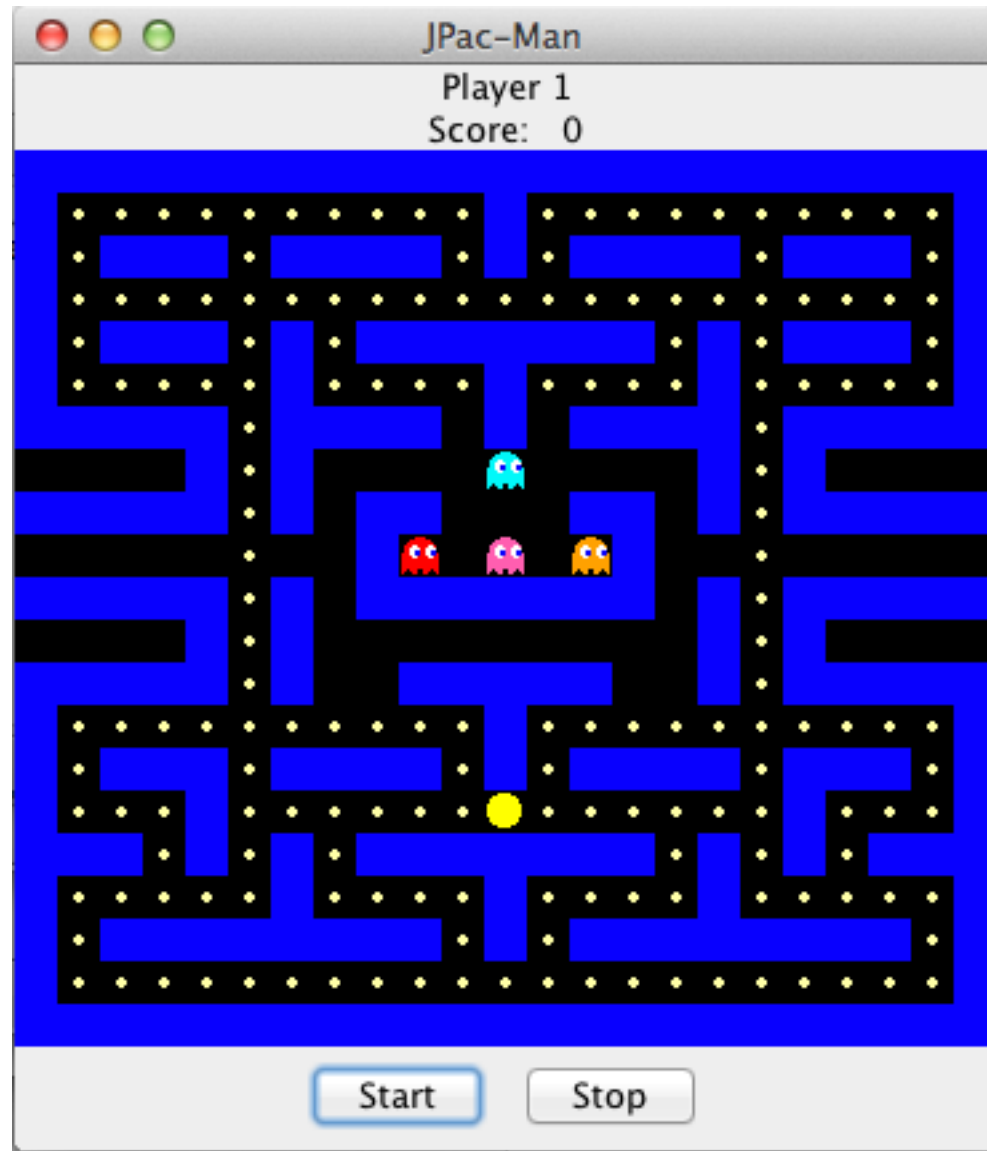
- End-to-end testing
- Structural testing
- Functional testing
- State-based testing
- Decision-table based testing
- Boundary-value testing
- CORRECT
- AAA



github.com/SERG-Delft/jpacman

Labwork Tools Used

- JUnit 5
- AssertJ
- Mockito
- Java 9
- IntelliJ
- Git
- Gradle
- GitLab CI




github.com/SERG-Delft/jpacman

Exams

- Midterm, Exam, Resit
- ~40 Multiple Choice Questions
- Midterm: May 24 (1st 10 lectures)
- Exam: July 2 (all material)
- Resit: August 16 (all material)

Multiple Choice:

People *hate* git because:

- A. git's commands are inconsistent and confusing.
- B. rebasing and push forcing are overly complicated operations
- C. handling merge conflicts can be a nightmare.
- D. All of the above
-  E. None of the above.

Multiple Choice: People *love* git because git

- A. supports understanding a change in its historical context
- B. supports isolating changes and moving them around
- C. supports identifying and discussing changes
- D. scales to 1000s of distributed developers



- E. All of the above

Your Questions Count!


- You can propose MC questions
- Bonus points if included in the exam
- Bonus points if discussed in class
- Submit:
 - 24h before next lecture
 - One week before (midterm) exam
 - Brightspace -> Assignment -> “Student supplied exam questions”

Secure <https://avandeursen.com/2016/07/24/asking-students-to-create-exam-questions/>

24
JUL
2016

Asking Students to Create Exam Questions

posted in Teaching by Arie van Deursen



Do you also find it hard to come up with good multiple choice questions? Then maybe you will like the idea of letting students propose (rather than just answer) questions. A colleague suggested this idea, arguing that it would benefit the students (creating a question requires mastering the material) and would save me work as well.

I liked this idea, and during the last three years I have applied it in my undergrad software testing course. This is a course for around 200 students which are evaluated based on an individual multiple choice exam (besides programming work conducted in pairs).

<https://avandeursen.com/2016/07/24/asking-students-to-create-exam-questions/>

Your Overall Grade

- Labwork: peer-graded.
 - Must be ≥ 5.75
 - Counts as 20% of final grade.
- Mid Term: graded.
 - Can be used to improve final grade
 - Then counts as 40% of final grade
- Exam / resit:
 - Must be ≥ 5.75
 - Counts as 40 or 80% of final grade

$$\text{Grade} = \frac{L + 2 * \max(M, E) + 2 * E}{5}$$



Software Quality and Testing Lab

CSE1110

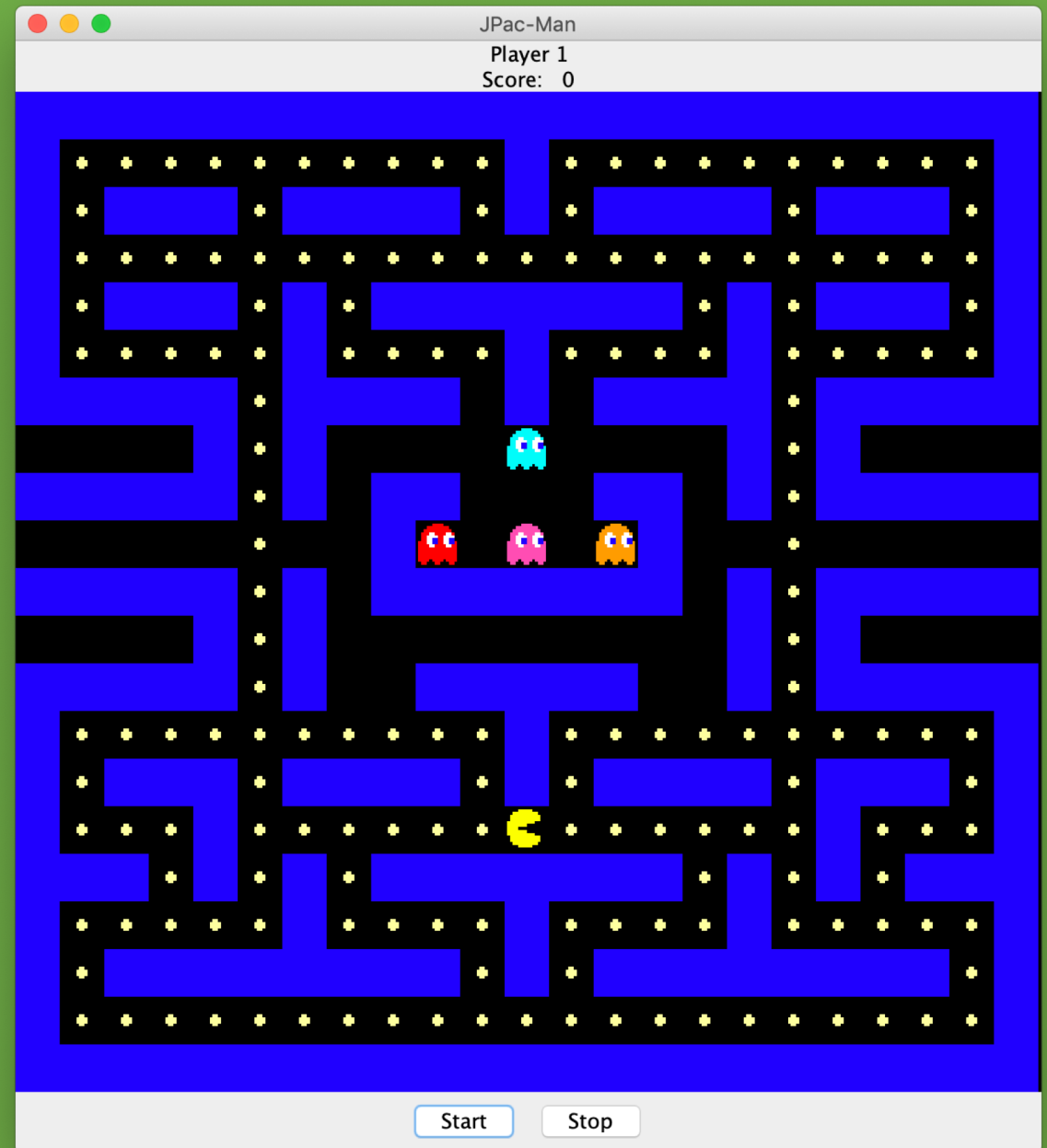
Casper Boone, Max Lopes Cunha
Delft University of Technology

Testing JPacman

Our own Java
implementation of Pac-Man

Already has some tests, but
it is your job to improve this!

Has all the basics, but could
use a few new features



Lab

4 assignments

In pairs

Different skills

Writing tests, answering theory questions and a little bit of new implementation

**Learn to write tests using different techniques
and at different levels**

Teams

In pairs

So, that's exactly 2 people

Same lab session

Pair programming

Show us that you worked on the assignment together:
commit often (both) and discuss changes in merge requests

Structure and Deadlines

Part 0: Get acquainted with the environment and tools.

Deadline: 03-05-2019, 5.00 pm - Review Deadline: 10-05-2019, 5.00 pm

UNGRADED

Part 1: Unit tests and boundary tests.

Deadline: 17-05-2019, 5.00 pm - Review Deadline: 28-05-2019, 5.00 pm

Part 2: Structural testing and mock objects.

Deadline: 03-06-2019, 5.00 pm - Review Deadline: 10-06-2019, 5.00 pm

Part 3: System tests, state-based testing, and mocking.

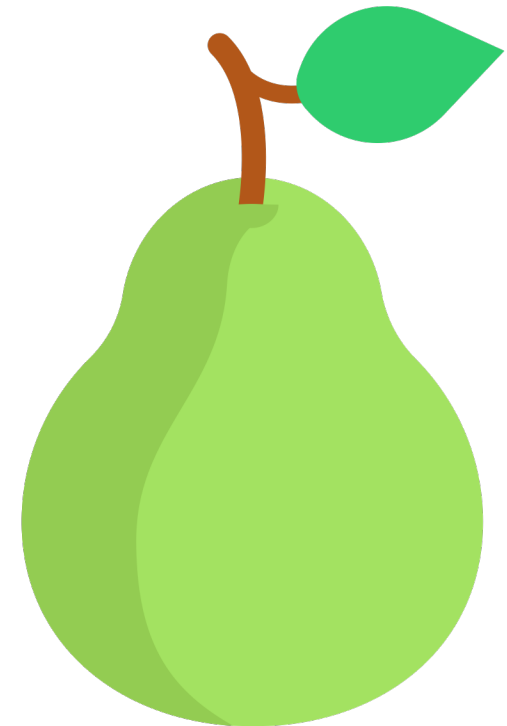
Deadline: 21-06-2019, 5.00 pm - Review Deadline: 27-06-2019, 5.00 pm

<https://se.ewi.tudelft.nl/cse1110-2019/>

Peer reviewing

**After submitting on *Peer*,
review your own solution + someone else's**
Learn about things you can improve and see
different approaches to solving the problem

**Grades are based on self grading,
reviews and TA checks**



DEVELOP IN



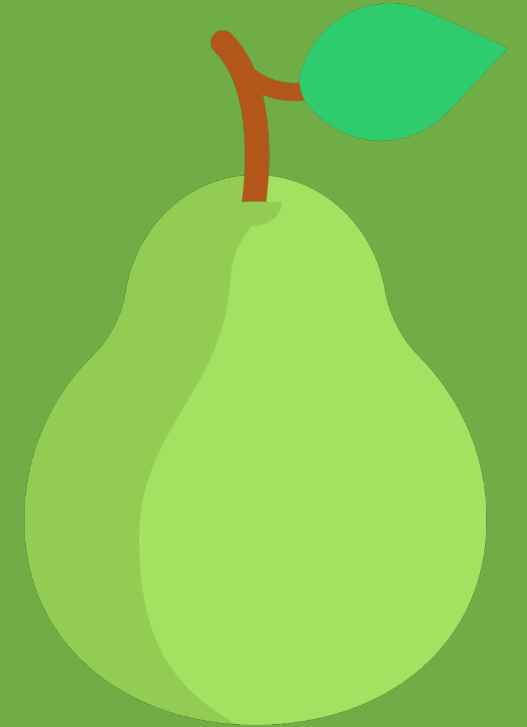
MANAGE CHANGES ON



GitLab

gitlab.ewi.tudelft.nl

REVIEW ON



peer.ewi.tudelft.nl

Tools

Build using gradle



Test using

- JUnit 5
- AssertJ assertions
- Mockito










Static Analysis

The logo for Checkstyle, featuring the word "checkstyle" in a lowercase, sans-serif font. The letter "i" is yellow, and the letter "e" is red. A red wavy line is positioned below the text.The logo for PMD, featuring the letters "PMD" in a bold, sans-serif font. The "P" is black, and the "M" and "D" are red. Below the letters is the tagline "DON'T SHOOT THE MESSENGER" in a smaller, black, uppercase font.The logo for SpotBugs, featuring the word "SpotBugs" in a bold, sans-serif font. The letter "o" in "Spot" is replaced by a magnifying glass with a red bug inside the lens. The magnifying glass handle is black.

Make sure everything works, run `gradle check`

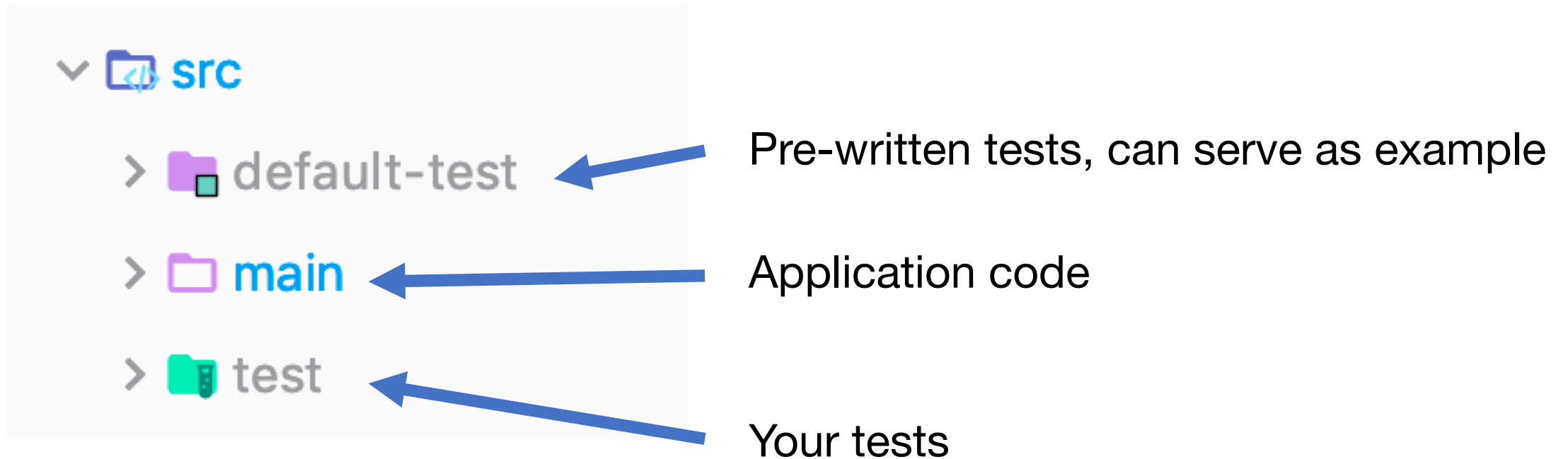
Continuous Integration

Pipeline **Jobs 2**

Status	Job ID	Name		Coverage	
Test					
 passed	#417536	test	 00:00:44  35 seconds ago	79.4077%	
Static Analysis					
 running	#417537	warnings	 00:00:13		

Project Structure

Simple Java project, mostly default setup



Quick tips

- 💡 Write clear commit messages
- 💡 Use merge requests and discuss changes
- 💡 Run `gradle check` before committing
- 💡 Don't let consecutive builds fail: fix issues first
- 💡 Ask questions: if you think your test is too complicated, it probably is
- 💡 Start early, the difficulty varies

Team Formation

TODAY!

Find a partner

Real life or Brightspace forum

Register your group on Brightspace

Collaboration > Groups

Create a GitLab account

Use student email and NetID as username
gitlab.ewi.tudelft.nl

Assignment

Will be released after the lecture

Software Testing and Quality Engineering
STQE Labwork, CSE1110

Edition 2018/2019

Arie van Deursen, Maurício Aniche
Casper Boone, Max Lopes Cunha, Azqa Nadeem

Delft University of Technology

April 21, 2019

1 Introduction

In this document, you will find everything about your JPACMAN labwork.

The objective of this lab work is to help you learn how you can apply the various tools and test strategies discussed during the lectures in practice.

You will apply these techniques to a simple game called JPACMAN, inspired by Pacman and written in Java. The amount of coding that needs to be done is relatively