Programming Project: **Testing**

Guillermo Román Díez

groman@fi.upm.es

E.T.S. Ingenieros en Informática
Universidad Politécnica de Madrid

2019-2020

Testing

Question

What is understood by *software testing*?

Testing

Question

What is understood by *software testing*?

- Let us start with two definitions taken from *IEEE Standard Glossary of Software Engineering Terminology*

Testing

Question

What is understood by *software testing*?

- Let us start with two definitions taken from *IEEE Standard Glossary of Software Engineering Terminology*

Testing

"*The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component*"

Testing

Question

What is understood by *software testing*?

- Let us start with two definitions taken from *IEEE Standard Glossary of Software Engineering Terminology*

Testing

"*The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component*"

Testing

"*The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items*"

Testing

- Can show only the **presence of faults, not their absence**
  - There exists a theoretical limitation: the problem of finding all faults in a program is *undecidable*

Testing

- Can show only the **presence of faults, not their absence**
  - There exists a theoretical limitation: the problem of finding all faults in a program is *undecidable*
- Must be carried out in a **systematic way** with the main goal of **finding defects** in a system

Testing

- Can show only the **presence of faults, not their absence**
    - There exists a theoretical limitation: the problem of finding all faults in a program is *undecidable*
- Must be carried out in a **systematic way** with the main goal of **finding defects** in a system
- Testing activities requires **planning**

Testing

- Can show only the **presence of faults, not their absence**
  - There exists a theoretical limitation: the problem of finding all faults in a program is *undecidable*
- Must be carried out in a **systematic way** with the main goal of **finding defects** in a system
- Testing activities requires **planning**
- Should be introduced in **all phases** of software development life cycle

Testing

- Can show only the **presence of faults, not their absence**
    - There exists a theoretical limitation: the problem of finding all faults in a program is *undecidable*
- Must be carried out in a **systematic way** with the main goal of **finding defects** in a system
- Testing activities requires **planning**
- Should be introduced in **all phases** of software development life cycle
- Helps in **reducing the overall cost** of the software development project

Testing

- Can show only the **presence of faults, not their absence**
  - There exists a theoretical limitation: the problem of finding all faults in a program is *undecidable*
- Must be carried out in a **systematic way** with the main goal of **finding defects** in a system
- Testing activities requires **planning**
- Should be introduced in **all phases** of software development life cycle
- Helps in **reducing the overall cost** of the software development project
- Helps in measuring and improving **software quality**

Testing

- Can show only the **presence of faults, not their absence**
  - There exists a theoretical limitation: the problem of finding all faults in a program is *undecidable*
- Must be carried out in a **systematic way** with the main goal of **finding defects** in a system
- Testing activities requires **planning**
- Should be introduced in **all phases** of software development life cycle
- Helps in **reducing the overall cost** of the software development project
- Helps in measuring and improving **software quality**
- **Reduces the risks** of using software

Testing historical phases

- **Phase I**: The *debugging-oriented* period – Testing was not separated from debugging

Testing historical phases

- **Phase I**: The *debugging-oriented* period – Testing was not separated from debugging
- **Phase II**: The *demonstration-oriented* period – Testing to make sure that the software satisfies its specification

Testing historical phases

- **Phase I**: The *debugging-oriented* period – Testing was not separated from debugging
- **Phase II**: The *demonstration-oriented* period – Testing to make sure that the software satisfies its specification
- **Phase III**: The *destruction-oriented* Period – Testing to detect implementation faults

Testing historical phases

- **Phase I**: The *debugging-oriented* period – Testing was not separated from debugging
- **Phase II**: The *demonstration-oriented* period – Testing to make sure that the software satisfies its specification
- **Phase III**: The *destruction-oriented* Period – Testing to detect implementation faults
- **Phase IV**: The *evaluation-oriented* Period – Testing to detect faults in requirements and design as well as in implementation

Testing historical phases

- **Phase I**: The *debugging-oriented* period – Testing was not separated from debugging
- **Phase II**: The *demonstration-oriented* period – Testing to make sure that the software satisfies its specification
- **Phase III**: The *destruction-oriented* Period – Testing to detect implementation faults
- **Phase IV**: The *evaluation-oriented* Period – Testing to detect faults in requirements and design as well as in implementation
- **Phase V**: The *prevention-oriented* Period – Testing to prevent faults in requirements, design, and implementation

Testing Process Maturity[1]

- **Level 0**: There's no difference between testing and debugging
  - Debug the programs with a few inputs chosen arbitrarily
- **Level 1**: The purpose of testing is to show that the software works
  - Software correctness is virtually impossible to achieve
- **Level 2**: The purpose of testing is to show that the software does not work
  - Testers may enjoy finding the problems, but the developers never want to find problems

⋮

[1] Boris Beizer, *Software Testing Techniques*

Testing Process Maturity

⋮

- **Level 3**: The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
  - It implies the acceptance of the fact that whenever we use software, we incur in some risks
  - And implies the assumption of the presence of bugs in any software
- **Level 4**: Testing is a mental discipline that helps all IT professionals to develop higher quality software
  - The purpose of testing is to improve the ability of the developers to produce high quality software

Testing terminology

Verification

"The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase"

- Is a technical tasks that uses knowledge about the software artifacts, technical requirements and specifications

Validation

"The process of evaluating software at the end of software development to ensure compliance with intended usage"

- Is not technical and depends on domain knowledge
- Is usually done by non-developers

Testing terminology

Software Fault (a.k.a. bug)

“*Is a static defect within the system*”

Testing terminology

Software Fault (a.k.a. bug)

"*Is a static defect within the system*"

Software Error

"*An incorrect internal state that is the manifestation of some fault.*"

- A *fault* might lead to an *error* but ot all faults come to errors
- By means of an error, a fault becomes apparent

Testing terminology

Software Fault (a.k.a. bug)

"*Is a static defect within the system*"

Software Error

"*An incorrect internal state that is the manifestation of some fault.*"

- A *fault* might lead to an *error* but ot all faults come to errors
- By means of an error, a fault becomes apparent

Software Failure

"*External, incorrect behavior with respect to the requirements or other description of the expected behavior*"

- Debugging is the process of finding a fault given a failure
- Faults and errors not always lead to software failures
- Ideally, software should be able to recover from an error and do not produce a failure

Testing terminology

Prefix values

"Any inputs (e.g. object creations, initializations, ...) necessary to put the software into the appropriate state to receive the test case values"

Testing terminology

Prefix values

"Any inputs (e.g. object creations, initializations, ...) necessary to put the software into the appropriate state to receive the test case values"

Test case values

"The input values (e.g. method parameters) necessary to complete some execution of the software under test"

Testing terminology

Prefix values

"Any inputs (e.g. object creations, initializations, ...) necessary to put the software into the appropriate state to receive the test case values"

Test case values

"The input values (e.g. method parameters) necessary to complete some execution of the software under test"

Expected results

"The result that will be produced when executing the test if and only if the program satisfies its intended behavior"

Testing terminology

Prefix values

"Any inputs (e.g. object creations, initializations, ...) necessary to put the software into the appropriate state to receive the test case values"

Test case values

"The input values (e.g. method parameters) necessary to complete some execution of the software under test"

Expected results

"The result that will be produced when executing the test if and only if the program satisfies its intended behavior"

Test case

"A test case is composed of the test case values, expected results, prefix values necessary for a complete execution and evaluation of the software under test"

Testing levels

- **Unit Testing**: assess software with respect to implementation and to the detailed design
    - Assess the units produced during the implementation phase
    - Evaluates how the elements of the software units interact with each other and their associated data structures
    - Is done by the programmer or by a testing engineer
- **Integration Testing**: assess software with respect to subsystem design
    - Evaluate whether the interfaces between software units in a given subsystem have consistent assumptions and communicate correctly
    - The programmer is usually the responsible of this phase

Testing levels

- **System Testing**: assess software with respect to architectural design
    - Assess whether the interfaces between modules (a group of software units) in a given subsystem have consistent assumptions and communicate correctly
    - It is responsibility of members of the development team
- **Acceptance Testing**: assess software with respect to the requirements
    - Evaluates whether the completed software in fact meets the requirements
- **Regression Testing**: Is done when some changes are made to software
    - Assess whether the updated software still possesses the functionality it had before the changes

Some famous software disasters

- 1994: Pentium floating-point division bug
  - Caused by the omission of five entries in a table of 1,066 values (part of the chip's circuitry) used by a division algorithm
  - They were left empty because a loop termination condition was incorrect
  - This bug is easy to find during unit testing but very hard at other testing levels
- 1996: Ariane 5 rocket exploded 7 seconds after liftoff
  - The disastrous launch cost approximately $370m
  - The developers of the Ariane 5 reused a successful inertial guidance system from the Ariane 4
  - Problems began to occur when the software attempted to work this 64-bit variable into a 16-bit integer
  - System tests that would have found the problem were technically difficult to execute, and so were not performed

Some famous software disasters

- 1999: Mars lander crashed
  - It was caused by a misunderstanding in the units of measure used by two modules created by separate software groups
  - One module was implemented using English units and forwarded the data to a module that expected data in metric unit
  - This is a typical integration problem
- 2012: Losing $460m in 45 minutes
  - In the stock market you win money if buy when the price is low, and sell when the price is high, but...
  - A flag which had previously been used was repurposed for use in a new functionality and the program believed it was in a test environment where the price is automatically drive up to perform the test
  - The result is that the system buy expensive and sell cheaper (as much fast as possible...)

Testing techniques

- Depending on the *knowledge about the system used* to write the tests, two main testing techniques can be used:
- **Functional Testing**: The selection of the test cases is based on the requirements
  - The software under test is viewed as a *black-box*
- **Control-flow Testing**: The software under test is known (*white-box*)
  - Test cases are determined by the implementation of the software

Testing Techniques: Functional Testing

- Emphasizes in the **external behaviour** of the software
- An exhaustive testing of values in the input domains is impossible
- The goal of the **domain testing** techniques is to select a set of input values with the highest probability of finding errors
- Two main options:
  - *Equivalence partitioning*
    - Divides the input domain into equivalence classes
    - A representative value of each class is equivalent to a test of any other value
  - *Boundary value analysis*:
    - Many errors occur at the boundaries of the domains
    - Test cases must evaluate the limit cases of the input/output domain

Testing Techniques: Control Flow Testing

- Requires to **know the source code** of the program under test
- The control structure of a program can be represented by a graph: the **control flow graph (CFG)** of the program
  - The CFG is a graph that represents a program and contains all paths that might be traversed through the program during its execution
- A common criterion is the **statement coverage**: the test suite of a program must *exercise* every statement of the program at least once
  - Other criteria can be included to it and strengthens the test suite: *decision coverage*, *condition coverage*, *path coverage*, ...

Testing Techniques: Other techniques

- **Random Testing**
  - The program is tested by generating random inputs
  - Is not systematic and not very reliable
- **Data Flow Testing Testing**
  - Emphasizes on the variables defined and used, and their values, at different points of the program
- **Mutation Testing**
  - The goal of the mutation testing is to assess the quality of the test cases which should be robust enough to fail with mutant code
  - The software under test is mutated (changed) and we check whether the test suite is able to detect the errors introduced

Software Testing Automation

- On average, testing requires up to 50% of software development costs
- Testing probably has more **repetitive** tasks than any other aspect of software development
- **Generating test cases** is typically a hard problem that requires **intervention** from the test engineer
  - There exist research tools for automatically generating test cases, but is still an open problem
- After any **change** in the program, all tests must be **executed again**
  - Do it manually is a boring task and could be a nightmare
  - Due to loss of attention of the tester, manually testing is a very error-prone task
- Thus. . .

Software Testing Automation

- On average, testing requires up to 50% of software development costs
- Testing probably has more **repetitive** tasks than any other aspect of software development
- **Generating test cases** is typically a hard problem that requires **intervention** from the test engineer
  - There exist research tools for automatically generating test cases, but is still an open problem
- After any **change** in the program, all tests must be **executed again**
  - Do it manually is a boring task and could be a nightmare
  - Due to loss of attention of the tester, manually testing is a very error-prone task
- Thus. . .

Automate your tests!!!

Test Automation Frameworks

- Test automation is *the use of software that executes tests and compare the output of the program with the expected output*
- **Reduces the time** taken in testing and helps to reduce the risks inherent to any software
- Helps to **increase the percentage of code covered** by the tests
- **Reduces** the need for **manual testing** and discovers defects that manual testing cannot expose
- **Regression** tests are **easily** and **quickly** executed
- For acceptance tests, it helps to know what portion of the **functionality** are already **implemented**

There are multiple frameworks for Java:

JUnit, *Arquillian*, *Mockito*, *JTest*, *TestNG*, *JWalk*, *Powermock*, . . .

Test suites

- It can be thought that test suites are expensive to develop
- But indeed, test suites reduce the debugging time more than the amount of time spent building the test suite
- Reduce the amount of bugs, and consequently the risks in using the code, in delivered code
- Helps in developing better programs, as programs need to be *testable* to develop the test suite

We are going to use JUnit 5 to write a test suite of your programs

JUnit 5

- **JUnit** is an open source framework composed by a set of Java classes and methods for writing and running tests for testing Java programs
  - Allows to compare the results and the expected results
  - Helps to build a test suite, that is, a set of tests that can be run at any time (at any environment)
  - Is integrated in most IDE's (e.g. Eclipse)
- We are going to use **JUnit 5**
- JUnit 5 uses **Java annotations** to indicate that a method is a test
- JUnit 5 defines multiple **assertion methods** to compare the obtained results and the expected results

Writing tests in JUnit 5

- How to write a test with JUnit 5?
  - Write a class with annotated methods
  - The testing class must be in a different directory of the class under test (remember the structure of a Maven project)

```
@Test                                  // JUnit test
void testSomething() {                 // test name
    C c = new C();                     // Prefix values
    int res = c.add(3,4)               // Input values
    assertEquals(7, res);              // Expected results
}
```

- Annotation `@Test` indicates to JUnit that this is a test method
- `assertEquals(a,b)` checks if `a`, the expected results, and `b`, the result of invoking the tested method, are equals
  - If not, the test fails

Assert methods in JUnit

- The package `org.junit.jupiter.api.Assertions` contains multiple *assert* methods to check the results obtained

`assertEquals(a,b)` `assertNotEquals(a,b)`	Invokes equals to check whether both parameters are equal
`assertTrue(a)` `assertFalse(a)`	Check whether a boolean parameter is evaluated to `true` or `false`
`assertNull(a)` `assertNotNull(a)`	Check whether an object parameter is evaluated to `null` or is not `null`
`assertSame(a,b)` `assertNotSame(a,b)`	Uses == to check whether both parameters have the same identity
`assertIterableEquals` `assertArrayEquals`	Check whether two iterators/arrays returns equal elements in the same order
`assertThrows`	Check whether an exception is thrown
`assertTimeout`	Check whether a method runs within a given timeout

All assertions can be found here:

`https://junit.org/junit5/docs/current/user-guide`

JUnit Annotations

- All methods annotated with `@Test` are identified as tests by JUnit
- Object initialization are needed for most tests and adding it in all tests would end up in a very repetitive code
- JUnit includes some method annotations to avoid repetitions:
  - `@BeforeAll`: The method is executed only once before running any test
  - `@BeforeEach`: The method is executed before running each test
  - `@AfterAll`: The method is executed only once after running all tests
  - `@AfterEach`: The method is executed after running each test

JUnit Annotations

- `@Nested`: Test classes can be nested to group tests
  - Methods of the outer classes annotated with `@BeforeEach` are executed before those annotated with `@BeforeEach` in the inner class (
- `@RepeatedTest`: The test is executed multiple times
- `@TestInstance`: Creates only one instance of the test class
- `@DisplayName`: Can be used to set a more readable name for the test
- `@ParameterizedTest` and `@ValueSource` produces multiple input values for the same test

All annotations can be found here:
`https://junit.org/junit5/docs/current/user-guide`

JUnit testing good practices

- Test only **one code unit at a time**
  - If this unit has multiple test cases, test them separately
- Aim that each unit test method performs **only one or two assertions**
- Make each test independent, do not make a *chain* of test cases in one test method
- Name your test on what they actually test and use **significant names** (use `@DisplayName`)
  - All methods must have their specific test or tests
- **Test the exceptions** that should be thrown by the methods
- For readability use the most **appropriate assertion** method
  - Use the correct order of the parameters to receive the failure feedback properly
- **Avoid printing** information in unit tests (as it will not be automatically checked)

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is:

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is: **$5**

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is: **$5**
- The cost to fix the same defect **after running a full build**:

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is: **$5**
- The cost to fix the same defect **after running a full build**: **$50**

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is: **$5**
- The cost to fix the same defect **after running a full build**: **$50**
- If the bug is found during the **integration tests**:

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is: **$5**
- The cost to fix the same defect **after running a full build**: **$50**
- If the bug is found during the **integration tests**: **$500**

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is: **$5**
- The cost to fix the same defect **after running a full build**: **$50**
- If the bug is found during the **integration tests**: **$500**
- If the bug is found during the **system tests**:

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is: **$5**
- The cost to fix the same defect **after running a full build**: **$50**
- If the bug is found during the **integration tests**: **$500**
- If the bug is found during the **system tests**: **$5000**

The cost of fixing bugs

Question

What do you think fixing a bug would cost on average?

- Google's has estimated the cost of delay in fixing defects
- The cost to fix a bug **immediately** after a programmer had introduced is: **$5**
- The cost to fix the same defect **after running a full build**: **$50**
- If the bug is found during the **integration tests**: **$500**
- If the bug is found during the **system tests**: **$5000**

The goal is to detect the bugs as early as possible

Some typical problems

- Software development organizations have continued to face software quality problems
  - Defects in code create maintenance (unwanted) costs
  - To get rid of defects a lengthy testing phase is done when the code is "frozen"
  - Poorly written code is difficult to understand and, thus, difficult to change
- Failing to meet actual needs
  - Ambiguities in the specification can easily lead to a project built on assumptions
  - At the end of many projects, the user detects that the software did not solve its real needs
  - Huge efforts are done to create more precise specifications, which ends up in longer projects and difficult to change

Waterfall Problems

- The classical **waterfall** model is divided in five phases performed sequentially
  - Requirements, Analysis, Design, Code and Testing
- This model presents some problems
  - Do not respond efficiently changes in the requirements
  - Phases has to be complete to pass to the next phase
  - A delay in the project could led to a reduction of the testing phase
  - Frequently do not involve the end-user in the phases
    - The user could be involved in the requirements definition phase
    - And might be involved in the testing phase, with little margin for making modifications

Test-Driven Development

- Similarly to the waterfall model, when we program, we usually follow these steps:
    1. First we design
    2. Implement the design
    3. Write the test somehow (usually not very exhaustively neither systematically)

Test-Driven Development

- Similarly to the waterfall model, when we program, we usually follow these steps:
  1. First we design
  2. Implement the design
  3. Write the test somehow (usually not very exhaustively neither systematically)

- **Test-Driven Development** turns this steps around and is based in a simple rule

Test-Driven Development

- Similarly to the waterfall model, when we program, we usually follow these steps:
  1. First we design
  2. Implement the design
  3. Write the test somehow (usually not very exhaustively neither systematically)

- **Test-Driven Development** turns this steps around and is based in a simple rule

Only ever write code to fix a failing test

Test-Driven Development

Test-Driven Development

Is a programming technique for building software that guides software development by writing tests

- Test-Driven Development programming sequence:

  Test → Code → Refactor

- **Test-Driven-Development**
  1. Write the test for next functionality you want to add
  2. Write the code you need to pass the test
  3. Transform the current code to a *better* design (*refactor*)

- TDD mantra:

  Red → Green → Refactor

TDD Step 1: Write the tests

- Writing test is more than writing a test: you are designing the interface of the program
  - It requires to think about how the code is going to be used
  - Many library interfaces are design without considering the user (other programmers)
- Thinking in the interface helps us to design the code according to the real needs of the user
  - Helps to avoid over-designs
  - You define exactly what your software needs to be able to do now (not tomorrow)
- And last but not least, tests are not ambiguous (requirements written in natural languaje are ambiguous very frequently)

TDD Step 2: Write just enough code

- The test initially fails and TDD recommends to write the minimal code to pass the test
- The main idea behind TDD is that the test guide what to implement so as to make progress on the development
- According to TDD, the goal is to make the test pass as quickly as possible
- We will think in the improvements later
    - After each test is passed, we have a refactoring phase

TDD Step 3: Refactor

Refactoring [2]

is the process of making changes to existing, working code without changing its external behavior. In other words, changing how it does it, but not what it does. The goal is to improve the internal structure

- In this step you have to take a look to the code and think in a way to make it better
- This phase is as important as the previous one
- Doing this way we are developing software in small increments
  - We are improving the design by altering it in a controlled manner
  - We have the test to check whether this changes still works: the tests are our safety net

[2] David Astels, *Test-Driven Development: A Practical Guide*

TDD Workflow

1. **From requirement to tests**
   - We produce a **list of test** (not code) from the requirements
   - Do not confuse tests with tasks
     - A test has a clear connection with the software capabilities
     - Tests are better than tasks for guiding our work
     - Tests must be concrete and executable
     - Tests must have a concrete input values and an expected output
   - There is no definition of what is a *good* test: we have heuristics and guidelines
     - Should be: small, atomic, isolated, focused, . . .
   - The test list is never finished, we start working with an *initial list* and it will grow up

TDD Workflow

2. **Starting with the first test**
    - We continue by writing the first test (ideally, the easiest one)
        - To do so, we have to imagine the ideal form of the production code for this particular test
        - We focus on what we could have instead on what we have (*programming by intention*)
    - We have to create:
        - Test class and the test method with its corresponding code
        - The class and the method under test
    - The test fails and we have to solve it by writing code in the class under test
        - The code should be as simple as possible
        - We move from *red* to *green*

TDD Workflow

3. **Making progress test by test**
   - Which test should be the next one? We can progress in two different ways:
     - *Breadth-first*: Writing test for the public interface and faking the internals and then proceed with the internals
     - *Depth-first*: Writing full tests for a slice of functionality and the proceed with the next slice
   - After adding a new test and write its code we have to **refactor**:
     - Remove redundant tests
     - Remove code duplications (in tests and in the code under test)
     - Generalize the solution
     - Go to smaller methods
     - Remove hard-coded code
     - Re-organize the logic in different classes
     - . . .

Test selection strategies

- There is no recipe, we have to rely on our expertise and intuition
- We could have four strategies for evaluating which test should be the next one:

1. **Details vs. Big picture**: should we first implement the overall framework or the concrete algorithms?
2. **Uncertain vs. Familiar**: should we start reduce uncertainty or proceed on the safe and known path?
3. **High value vs. Low-hanging Fruit**: should we start with the tests that leads to more progress?
4. **Happy path vs. Error situations**: should we start with the normal or with the exceptional scenarios?

Further Reading

- P. Ammann and J. Offutt, **Introduction to Software Testing**, 1 ed., Cambridge University Press, New York, NY, USA, 2008.
- K. Beck, **Test Driven Development: By example**, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- L. Koskela, **Test Driven: Practical TDD and Acceptance TDD for Java Developers**, Manning Publications Co., Greenwich, CT, USA, 2007.
- L. Koskela, **Effective Unit Testing: A Guide for Java Developers**, Running Series, Manning, 2013.