Software Engineering

QA + Testing III

Software & Systems Engineering | Prof. Dr. Andreas Vogelsang | 11.12.2023

@andivogelsang
vogelsang@cs.uni-koeln.de

Learning Goals for Today

- Know more about testing pragmatics
- Know more intelligent methods to generate test inputs
- Know methods for testing quality attributes (performance, security, reliability)

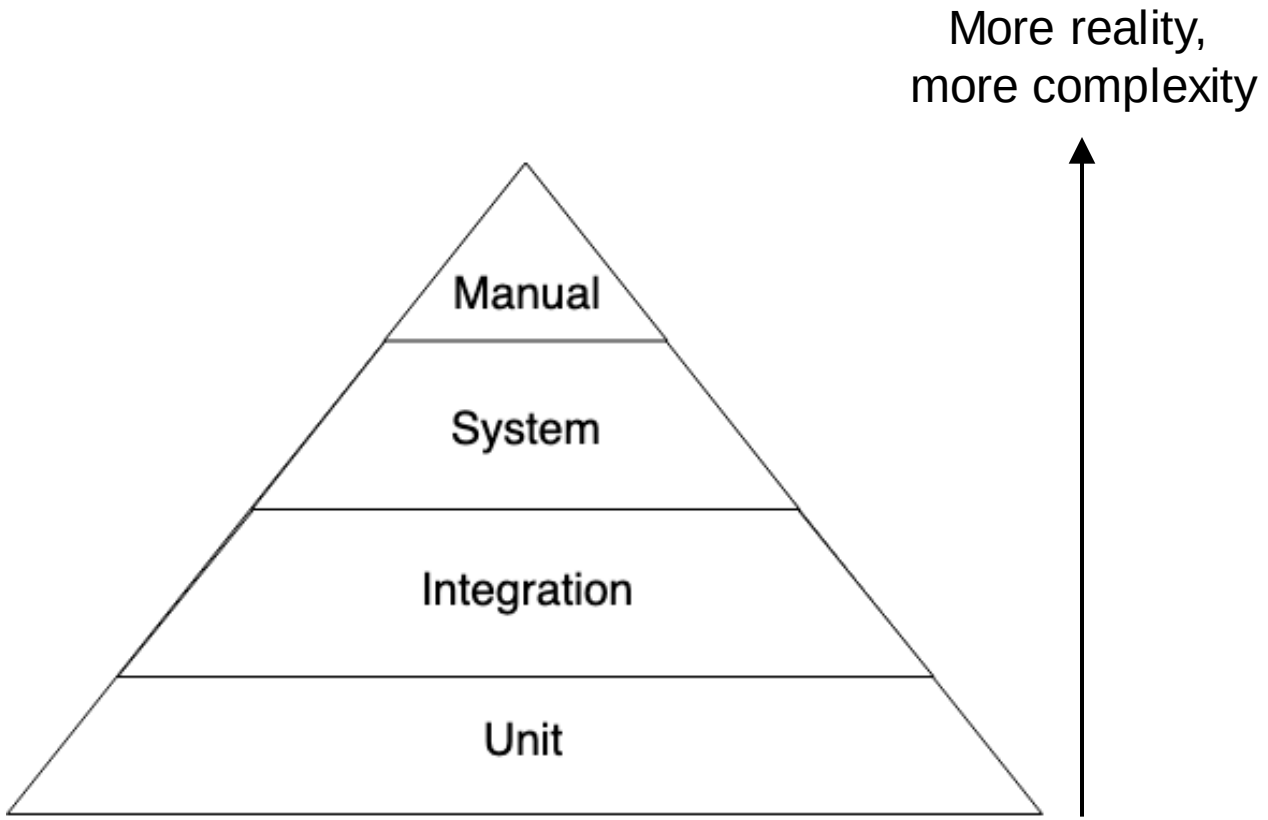
Universität zu Köln

Testing Pragmatics

The Testing Pyramid

The Testing Pyramid

- Main motivation: **optimize costs** of testing
- The size of the pyramid slice represents the number of tests one would want to carry out at each test level
- Unit testing is at the bottom of the pyramid: they are fast, require less effort to be written, and give developers more control.
- The integration test area is a bit smaller, indicating that in practice, we should do integration tests less than unit tests.
- Testers should then favor system tests less than integration tests and have even fewer manual tests.



When to write which test?

The testing pyramid

Unit tests: When the component is about an algorithm or a single piece of business logic of the software system.

Integration tests: Whenever the component under test interacts with an external component (e.g., a database or a web service) integration tests are appropriate.

System tests: Use a risk-based approach. What are the absolutely critical parts of the software system under test?

Manual tests: Should only be used where requirements and specifications are incomplete or, if there is a lack of time.

Beware of exceptions

- Development of system software (operating systems, database systems,…) may demand more system tests because a lot happens at “low level” (disk I/O, socket communication)
- Development of Cyber-Physical systems may demand more system tests because the system strongly depends on its physical context (e.g., a water management station)

Universität zu Köln

Test-Driven Development

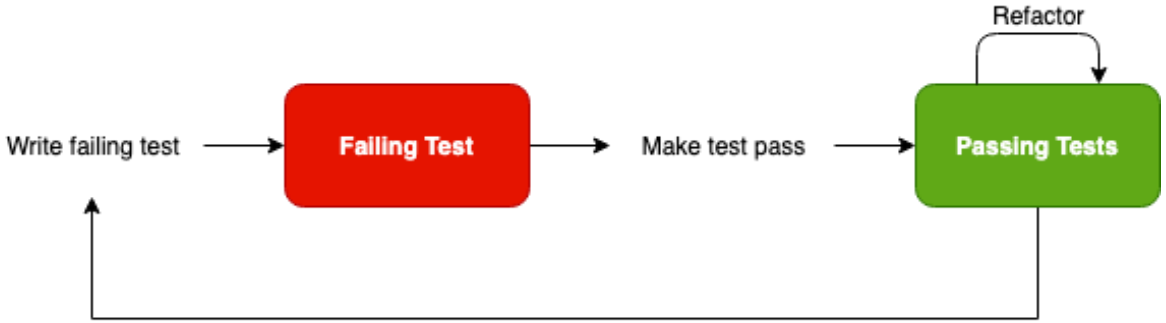
Test-Driven Development (TDD)

Motivation: Writing tests after coding often leads to late feedback or no tests at all

TDD approach: Write tests first (before coding)!

Advantages and claims:

- Thinking about tests also means thinking about requirements
- The tests control the pace of the implementation (simple vs. complex tests)
- Quick feedback
- Feedback on design (e.g., a class with many tests may have too many functionalities, too many mocks indicates tight coupling)



Pragmatic TDD

- Use TDD when you do not know how to design and/or architect a part of the system
- Use TDD when you are dealing with a complex problem, a problem in which you lack experience
- Do not use TDD when you are familiar with the problem, or the design decisions are clear in your mind (still, do not forget to write tests)

Regression Testing

Regression Testing (Regressionstesten)

Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

Goal

- every change has desired effects
- regression testing aims to detect undesired side-effects

Assumptions

- minor change to the program behavior
- program was largely correct before
- results only differ in old and new version where desired

Characteristics

- requires to document test results before changes
- actual results are not only compared to specification, but also to previous results for the same inputs

Universität zu Köln

Intelligent Testing Methods

Test Suite Quality

```java
public class Division {
  public static int[] getValues(int a, int b) {
    if (b == 0) {
      return null;
    }
    int quotient = a / b;
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

```java
@Test
public void testGetValues() {
  int[] values = Division.getValues(1, 1);
}

@Test
public void testZero() {
  int[] values = Division.getValues(1, 0);
}
```

Statement Coverage? Branch Coverage?

Will the tests detect errors?
What is the tests' *fault detection capability*?

Universität zu Köln

Mutation Testing

Mutation Testing

Motivation: How do we know if we have tested enough?

Idea: Determine the fault detection capability by measuring the number of bugs that are detected by a test suite

Approach: Change small parts of the code (randomly), and check if the tests can find the introduced fault.

Terms

Mutant: Given a program P, a mutant called P' is obtained by introducing a syntactic change to P. A mutant is ***killed*** if a test fails when executed with the mutant.

Syntactic Change: A small change in the code that mimics typical human mistakes. Such a small change should make the code still valid, i.e., the code can still compile and run.

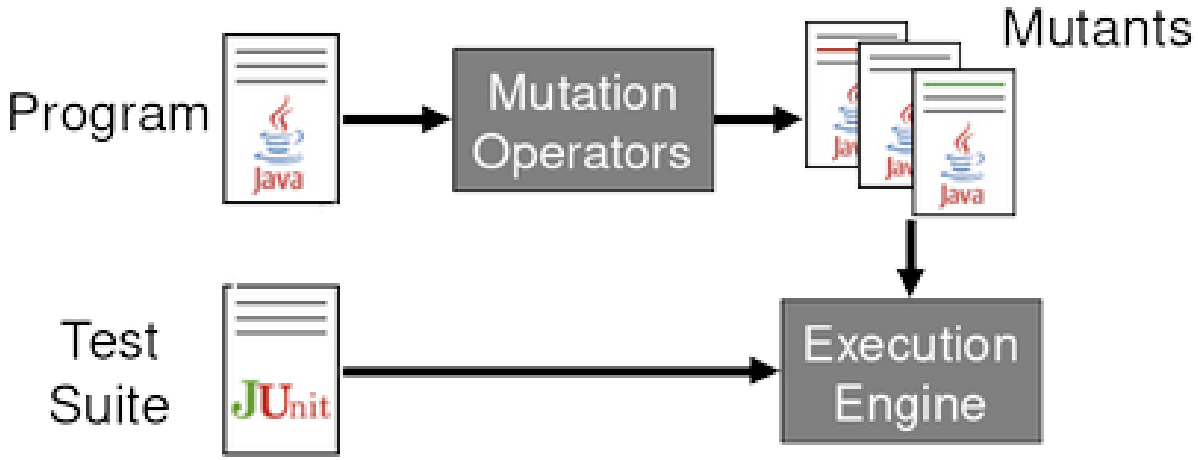
Assumptions

- The **Competent Programmer Hypothesis**: We assume that the program is written by a competent programmer, who creates a program that is either correct, or it differs from a correct program by a combination of simple errors.
- The **Coupling Effect**: The coupling effect hypothesis states that simple faults are coupled with more complex faults (i.e., test cases that detect simple faults, will also detect complex faults).

Consequences on Mutant Size

- Mutants size should be small

Mutation Testing: Overview



$$mutation\ score = \frac{killed\ mutants}{mutants}$$

Goal

Add tests to the test suite to increase the mutations score

Mutation Testing: Example

Mutant 1

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b != 0) { // bug (mutation)
      return null;
    }
    int quotient = a / b;
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

```
@Test
public void testGetValues() {
  int[] values = Division.getValues(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}

@Test
public void testZero() {
  int[] values = Division.getValues(1, 0);
  assertNull(values);
}
```

Mutant 2

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b == 0) {
      return null;
    }
    int quotient = a * b; // bug (mutation)
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

$$mutation\ score = \frac{killed\ mutants}{mutants}$$

$$mutation\ score = \frac{1}{2}$$

Mutation Testing: Example

Mutant 1

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b != 0) { // bug (mutation)
      return null;
    }
    int quotient = a / b;
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

Mutant 2

```
public class Division {
  public static int[] getValues(int a, int b) {
    if (b == 0) {
      return null;
    }
    int quotient = a * b; // bug (mutation)
    int remainder = a % b;

    return new int[] {quotient, remainder};
  }
}
```

```
@Test
public void testGetValues() {
  int[] values = Division.getValues(1, 1);
  assertEquals(1, values[0]);
  assertEquals(0, values[1]);
}

@Test
public void testZero() {
  int[] values = Division.getValues(1, 0);
  assertNull(values);
}

@Test
public void testGetValuesDifferent() {
  int[] values = Division.getValues(3, 2);
  assertEquals(1, values[0]);
  assertEquals(1, values[1]);
}
```

$$mutation\ score = \frac{2}{2}$$

Universität zu Köln

Mutation Testing: Conclusions

Challenges

- **Automation is key**: Mutants should be created automatically (based on *mutation operators*). There are tools for that.
- **Some mutants may be impossible to kill** (i.e., a mutations score of 100% cannot be reached)
- **Mutation testing is computationally expensive**. The test suite must be run on every mutant.

Effectiveness

- Studies about mutation testing have shown that mutants can provide a good indication of a test suite's fault detection capability, if the mutation operators are carefully selected, and the equivalent mutants are removed.
- A study by Just et al. shows that mutant detection is positively correlated with real fault detection. In other words, the more mutants a test suite detects, the more real faults the test suite can detect as well.
- This correlation is independent of the coverage

Fuzz testing

Fuzz Testing

Negative software testing method that feeds malformed and unexpected input data to a program, device, or system with the purpose of finding security-related defects, or any critical flaws leading to denial of service, degradation of service, or other undesired behavior

Terms

Fuzzer: A program or framework used to generate fuzz tests.

Goal

Create test cases that make the System-under-Test crash (only valid test oracle)

How to generate fuzzed inputs?

- **Mutation-based Fuzzing:** Creates permutations from example inputs (e.g., replacing or appending characters). Since mutation-based fuzzing does not consider the specifications of the input format, the resulting mutants may not always be valid inputs.
- **Generation-based Fuzzing:** Takes the file format and protocol specification of the SUT into account when generating test cases. Generative fuzzers take a data model as input that specifies the input format.

Testing Reliability

Reliability Testing

Reliability of the program logic

- How does a system handle unexpected inputs?
- General approach: Use random input values
- Realism or probability of these input values is irrelevant

Reliability of the underlying infrastructure

- How does a system handle errors in the underlying infrastructure (e.g., server crash)?
- Important for distributed systems (e.g., cloud services)

Random testing

- Testing with random and independent inputs from the input domain of a program
- Random testing can generate probabilistic guarantees about the failure likelihood of a system (e.g., testing 23,000 random inputs without error guarantees that a system does not crash more than once in 10,000 runs with a probability of 90%)
- Challenge: How to define the expected output?
- Fuzz testing is a special form of random testing

Chaos Engineering

Chaos Engineering

- Experimenting on a software system in production in order to build confidence in the system's capability to withstand turbulent and unexpected conditions
- Perturbation model: Programs that mimic rare or catastrophic events that can happen in production
- First perturbation model (*the Chaos Monkey*) developed by Netflix: disabling computers in Netflix's production network
- General perturbation models: server shutdowns, latency injection, resource exhaustion

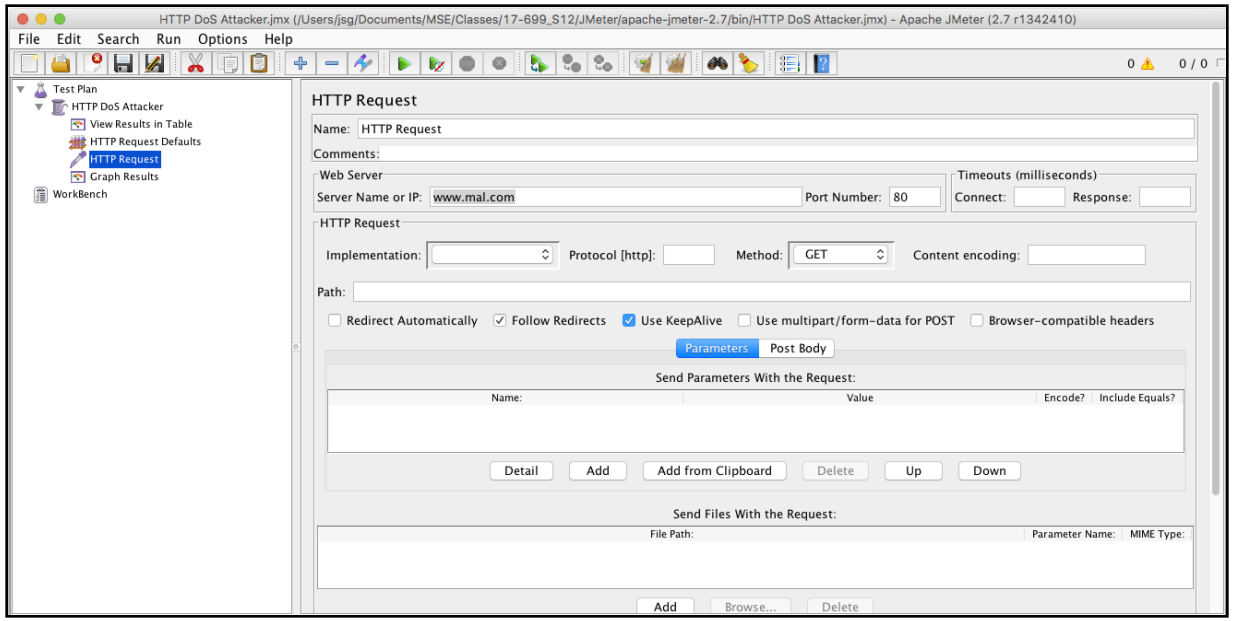
Performance Testing

Unit and Regression Testing for Performance

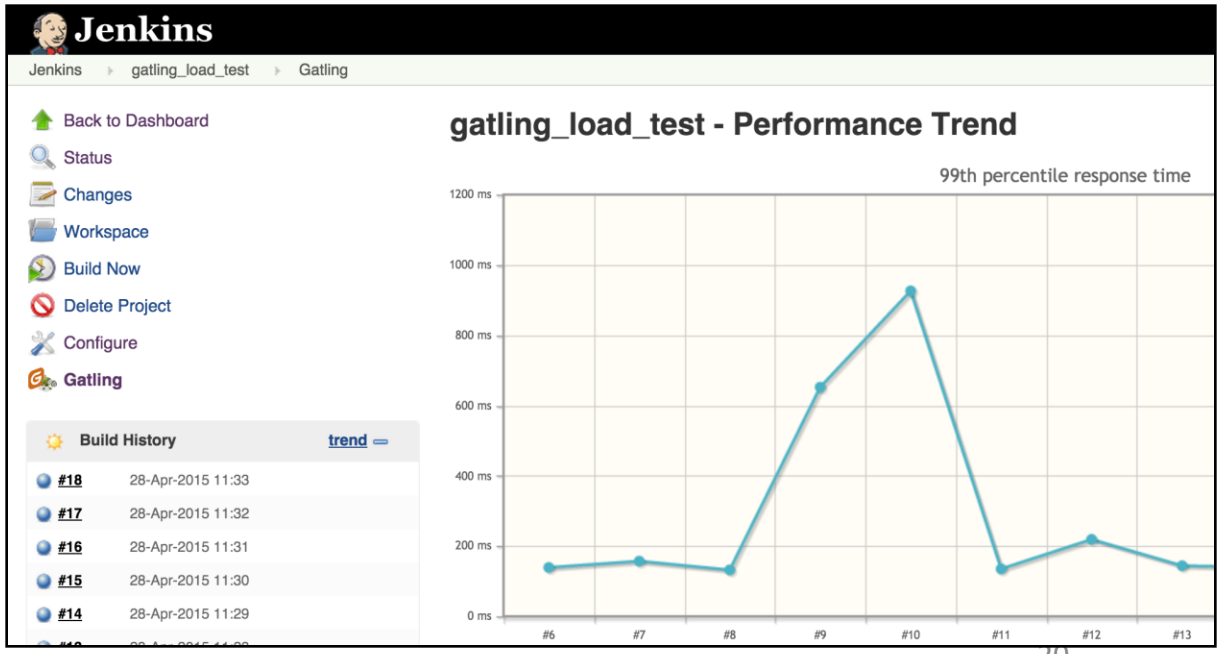
Unit and Regression Testing for Performance

- Measuring the execution time of critical components
- Recording of execution time and compare over time

JMeter



Gatlin

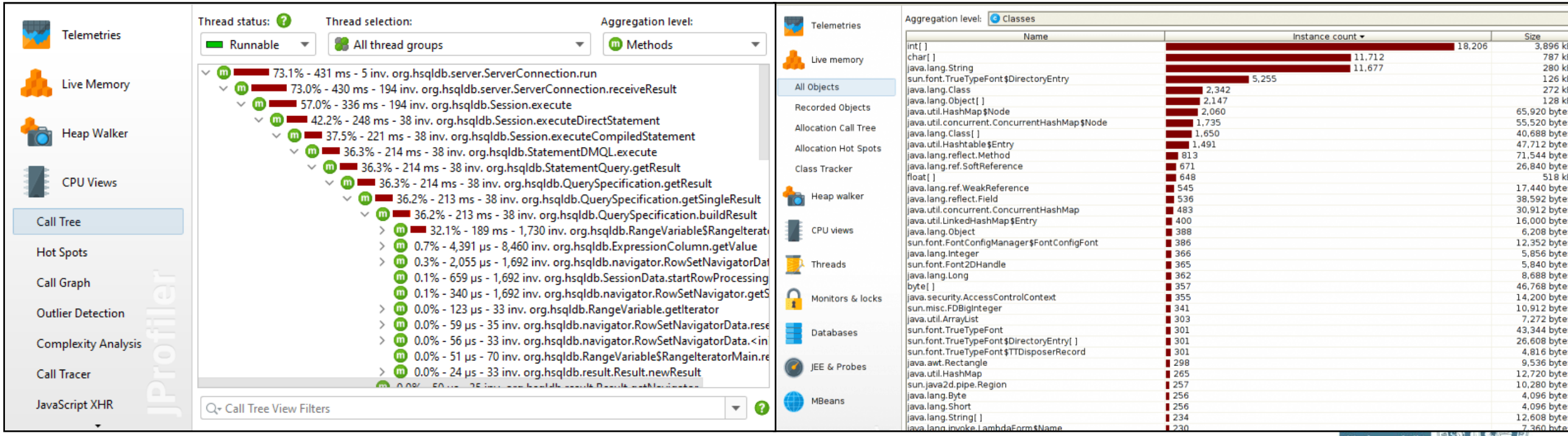


Profiling

Profiling

Finding bottlenecks in execution time and resource utilization

JProfiler



Types of Performance Tests

Technique	Description
Baseline testing	• Performing a single transaction as a single user for a specified period or for a specified number of transaction repetitions • Running without other activities under otherwise normal conditions • Establishing a point of comparison for further test runs
Load testing	• Testing the application with maximum target load, but not exceeding it • Test performance targets (e.g., response time, throughput, etc.) • Approximation of the expected peak load of the application
Scalability testing	• Test application with increasing load • Scaling should not require a new system or software redesign
Soak (stability) testing	• Provide continuous load to application over a period • Identify problems that occur over a longer period, e.g., memory leakage
Spike testing	• Test system with high load for a short period of time • Verify system stability during a burst of simultaneous user and/or system activity with varying load levels over varying periods of time.
Stress testing	• System resource overload • Ensure that the system fails safely and recovers properly

Security Testing

Software vs. Security Testing

Software vs. Security Testing

The goal of software testing is to check the correctness of the implemented functionality, while the goal of security testing is to find bugs (i.e., vulnerabilities) that can potentially allow an intruder to make the software misbehave.

Common Vulnerabilities in Java

- Code injection vulnerability
- Type confusion vulnerability
- Buffer overflow vulnerability
- Arbitrary Code Execution (ACE)
- Remote Code Execution (RCE)

Security Testing Approaches	White-box	Black-box
Static approaches	Review, Bug patterns, ASTs, CFGs, DFDs	Reverse engineering
Dynamic approaches	Tainting, Dynamic validation, Symbolic execution	Penetration testing, Reverse engineering, Behavioral analysis, Fuzzing

Taint Analysis

Taint Analysis (for confidentiality)

Ensure that sensitive data does not “flow” from a source to a sink

Terms

Source: Sensitive information (e.g. passwords, locations, contacts,...) passed to variables, return values, I/O streams,…

Sink: Untrusted communication channels (e.g. displaying or sending data).

Universität zu Köln

Taint Analysis – Example

Credit card number should not be `visible`

Security labels: *secret, public*

Source: *Variable* `creditCardNb`

Sink: *Variable* `visible`

```
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
     visible = true;
}
```

label(`creditCardNb`*) = secret*

label(`x`*) = secret*

label(`visible`*) = public*

label(`condition`*)*
= label(x) + label (1000)
= secret + public = secret

label(`visible`*)*
= label(condition) + label (true)
= secret + public
= secret

Penetration Testing

Pen(etration) Testing

- Authorized simulated cyberattack on a computer system, performed to evaluate the security of the system
- Penetration testing checks the SUT in an end-to-end fashion, which means that it is done once the application is fully implemented.
- Pen testing tools contain a **Vulnerability Scanner** module that either runs existing exploits or allow the tester to create an exploit. They also contain **Password Crackers** that either brute-force passwords or perform a dictionary attack (i.e., choose inputs from pre-existing password lists).

Brenan Keller
@brenankeller

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

Tweet übersetzen

10:21 nachm. · 30. Nov. 2018 · Twitter for iPhone

25.413 Retweets **1.605** Zitierte Tweets **63.976** „Gefällt mir"-Angaben

Universität zu Köln