JavaEE quality

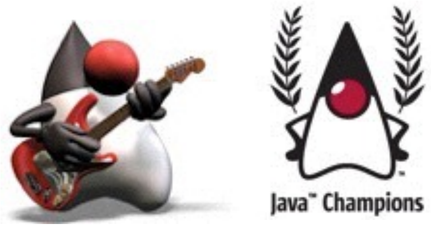
adam-bien.com

I'm working as Java EE/SE developer, consultant, sometimes author, speaker and trainer with Java since 1995...

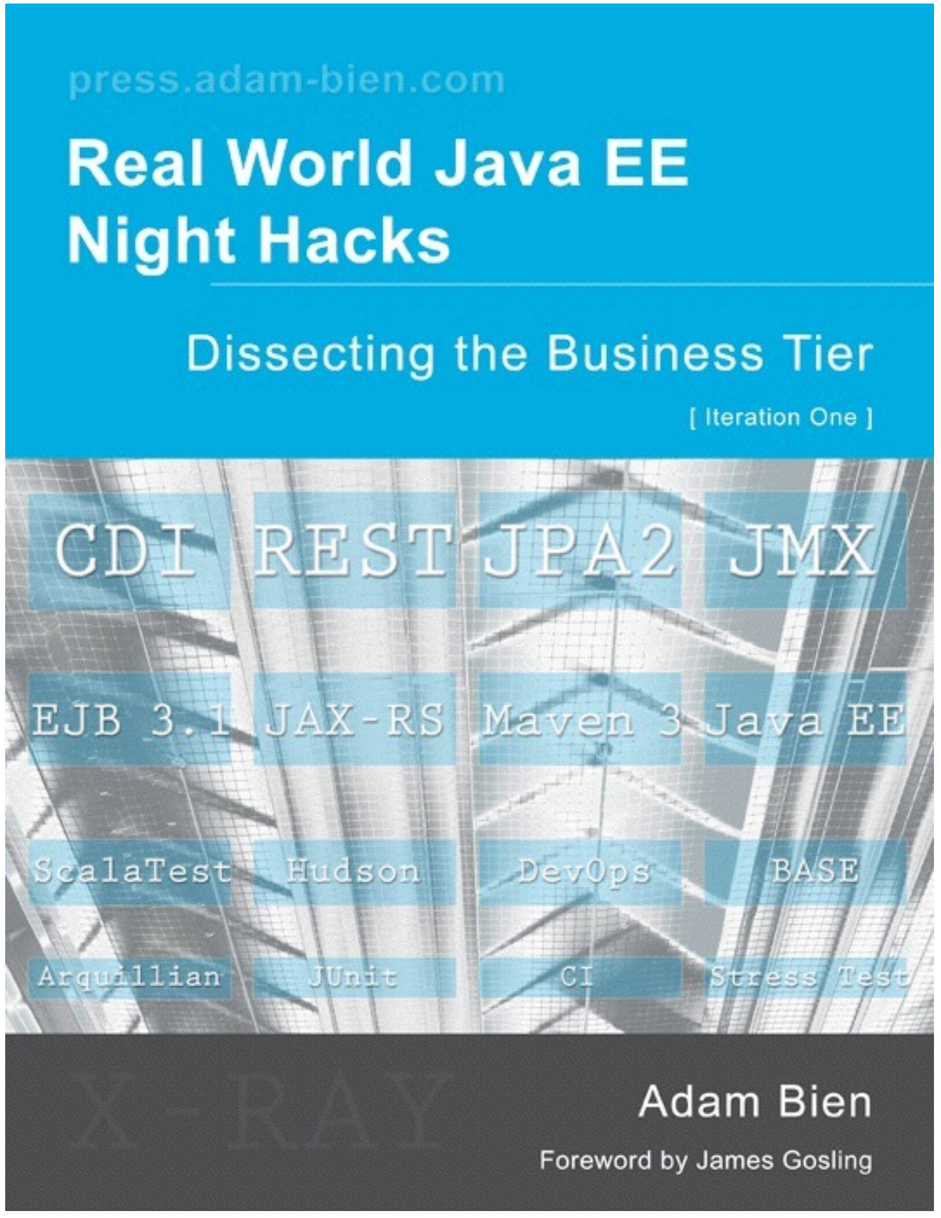
...and still really enjoying it!

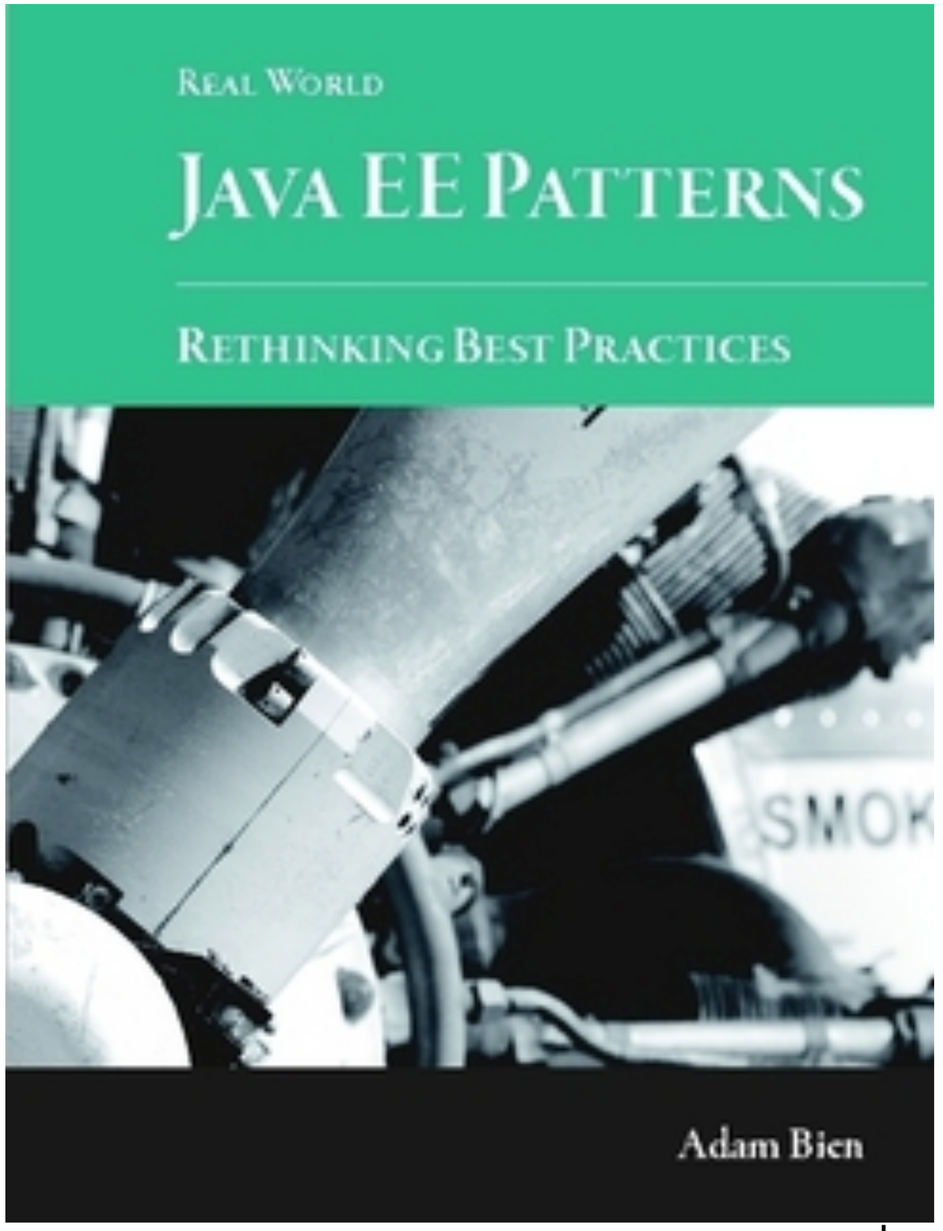
Java Programming Language rocks!

workshops.adam-bien.com



adam-bien.com

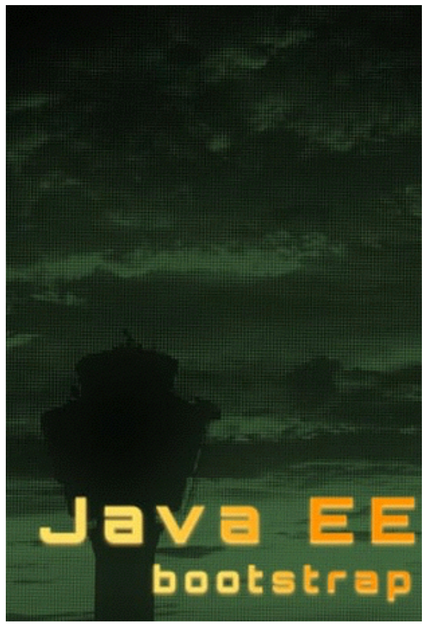


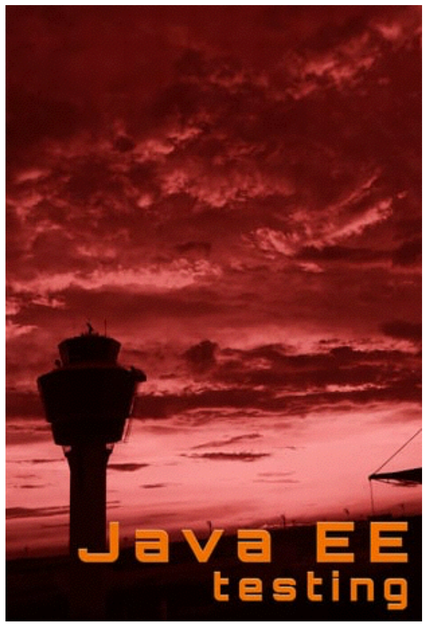


adam-bien.com

airhacks.TV

adam-bien.com





adam-bien.com

airhacks.io

Unit Testing

Unit Testing

"In computer programming, unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures are tested to determine if they are fit for use.

Intuitively, one can view **a unit as the smallest testable part of an application**. In procedural programming, a unit could be an entire module, but is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method."

Mutation Testing

Mutation Testing

Mutation testing (or Mutation analysis or Program mutation) is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves modifying a program in small ways. Each mutated version is called a mutant and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called killing the mutant. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants. Mutants are based on well-defined mutation operators that either mimic typical programming errors (such as using the wrong operator or variable name) or force the creation of valuable tests (such as dividing each expression by zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Integration Testing

Integration Testing

“The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. These "design items", i.e. assemblages (or groups of units), are exercised through their interfaces using black box testing, success and error cases being simulated via appropriate parameter and data inputs.

Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test whether all the components within assemblages interact correctly, for example across procedure calls or process activations, and this is done **after testing individual modules, i.e. unit testing**. The overall idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages.”

System Testing

System Testing

“System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.

As a rule, system testing takes, as its input, all of the "integrated" software components that have passed integration testing and also the software system itself integrated with any applicable hardware system(s).

The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limited type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.”

Torture Testing

Stress Testing

"Stress testing (sometimes called **torture testing**) is a form of deliberately intense or thorough testing used to determine the stability of a given system or entity. It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results. Reasons can include:

- to determine breaking points or safe usage limits
- to confirm intended specifications are being met
- to determine modes of failure (how exactly a system fails)
- to test stable operation of a part or system outside standard usage

Stress Testing

"Stress testing, in general, should put computer hardware **under exaggerated levels of stress in order** to ensure stability when used in a normal environment.

These can include extremes of workload, type of task, memory use, thermal load (heat), clock speed, or voltages. Memory and CPU are two components that are commonly stress tested in this way."

Load Testing

Load Testing

“Load testing is the process of putting demand on a system or device and measuring its response. Load testing is performed to determine a system’s behavior under both normal and anticipated peak load conditions. It helps to identify the maximum operating capacity of an application as well as any bottlenecks and determine which element is causing degradation. When the load placed on the system is raised beyond normal usage patterns, in order to test the system's response at unusually high or peak loads, it is known as stress testing. The load is usually so great that error conditions are the expected result, although no clear boundary exists when an activity ceases to be a load test and becomes a stress test.

Load Testing

There is little agreement on what the specific goals of load testing are.[citation needed] The term is often used synonymously with concurrency testing, software performance testing, reliability testing, and volume testing. Load testing is usually a type of non-functional testing although it can be used as a functional test to validate suitability for use."

Functional Testing

Functional Testing

Functional testing is a quality assurance (QA) process and a type of black box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (not like in white-box testing).

Functional Testing usually describes what the system does.

Acceptance Testing

Acceptance Testing

In engineering and its various subdisciplines, acceptance testing is a test conducted to determine if the requirements of a specification or contract are met. It may involve chemical tests, physical tests, or performance tests.

In systems engineering it may involve black-box testing performed on a system (for example: a piece of software, lots of manufactured mechanical parts, or batches of chemical products) prior to its delivery.

Acceptance Testing

Software developers often distinguish acceptance testing by the system provider from acceptance testing by the customer (the user or client) prior to accepting transfer of ownership. In the case of software, acceptance testing performed by the customer is known as user acceptance testing (UAT), end-user testing, site (acceptance) testing, or field (acceptance) testing.
A smoke test is used as an acceptance test prior to introducing a build to the main testing process.

Software Quality

Cyclomatic Complexity

Cyclomatic Complexity

Cyclomatic complexity (or conditional complexity) is a software metric (measurement). It was developed by Thomas J. McCabe, Sr. in 1976 and is used to indicate the complexity of a program. It is a quantitative measure of logical strength of the program. It directly measures the number of **linearly independent paths through a program's source code**. […]

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

Cyclomatic Complexity

The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. For instance, if the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code. If the code had a single IF statement containing a single condition, there would be two paths through the code: one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE.

Build

- Pragmatic, Simple and Fast Maven Builds
- Beyond Maven: deployment and provisioning
- Maven vs CD

Unit Testing

- Unit Testing with JUnit
- Parameterized tests
- Mocking with mockito
- Testing Java EE 7 with JUnit

Integration Testing

- Killer Use Cases for arquillian
- Plain JUnit For Integration Tests?
- JPA, Persistence, Bean Validation

System Testing

- UI and Acceptance Testing
- UI testing with Graphene and a bit Selenium
- PhantomJS and CasperJS
- System Tests with JUnit, Java 8 ...and JavaScript?
- System Tests or Back To Reality
- Functional Integrated Testing with or without Fitnesse
- Parameterized Functional Tests

Stress and Load Tests

- Implementation and Monitoring strategies
- Nightly Stress Tests with JMeter
- "Business" Traceability

Continuous Integration / Continuous Deployment

- Automation with Jenkins
- Build Pipelines with Jenkins
- Automatic Server Setup and Deployment
- Promotion and Deployment
- Deployment Process Orchestration
- Useful Plugins and Hacks
- Performance Tuning and Repository Manager Integration

Pragmatic Quality

- Minimalism
- Consistency
- Straightforwardness

Measurable Quality

- Cyclomatic Complexity
- RFC (# Response For Class) -> # of methods invoked by calling a public method.
- LCOM4 (# of responsibilities of a class) -> Lack of Cohesion, number of related methods, should be 1.

Measurable Quality

- Number of Children
- Depth of Inheritance Tree (DIT)
- Couplings

“No Discussion” Quality

- Sonar
- Continuous Quality Assurance
- Rule Adjustments and Customization for JavaEE
- Jenkins + Sonar = The Dream Team

The Future Is Now

- Beyond Java--Useful tools
- Vagrant
- Virtual Box / VMWare
- docker

Adam Bien's Workshops

Munich (MUC) Airport Workshops, 11th, Spring Edition:

Java EE 7 Bootstrap, Monday, April 4th, 2016
also available as: [Streaming / Download Edition]

Effective Java EE 7, Tuesday, April 5th, 2016
also available as: [Streaming / Download Edition]

Java EE 7 Architectures, Wednesday, April 6th, 2016

Dedicated Virtual Workshops [on demand]

Collateral materials from previous editions.

Special Event: Continuous Java EE 7 Testing, Deployment and Code Quality, July 11th, 2016
also available as: [Streaming / Download Edition]

Special Event: Microservices With Java EE 7 and Java 8, July 12th, 2016

Non-Java Event: JavaScript for Java Developers, Monday, May 2nd, 2016

Non-Java Event: HTML 5 for Java Developers, Tuesday, May 3rd, 2016

Thank You!

blog.adam-bien.com
twitter.com/AdamBien