Reporting of Performance Tests in a Continuous Integration Environment

Master's Thesis

Alexander Barbie

February 8, 2018

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Wulf

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 8. Februar 2018

Abstract

Software engineering becomes a more and more continuous development process. The complexity of software products and its code increases and even minor changes of a single team member can affect the whole product. Although a program's performance is an important quality criteria there is no proper framework to test performance. The existing frameworks perform benchmarks, yet no performance tests. A performance test compares a benchmark's score against a predefined assertions for the score. The microbenchmarking framework Java Microbenchmarking Harness (JMH) allows developers to write microbenchmarks for the programming language Java. In order to report performance tests in a continuous integration environment, we enhance the performance testing framework RadarGun. The framework RadarGun runs performance tests via JMH and compares them against user defined assertions. We enhanced the pipe-and-filter architecture of RadarGun. Therefore, we split the execution filter of RadarGun to report each performance test one by one, instead of reporting the results after all benchmarks were run. Hence, a user gets a progress monitor in real time, when executing his performance tests with RadarGun.

Additionally, we developed two plugins that integrate RadarGun and report the results. One plugin was developed for the continuous integration environment Jenkins. This plugin includes RadarGun as post build step, to report the results by RadarGun. When including our post build step in a build-pipeline configuration, a build fails, if a performance tests score does not meet the required lower or upper bound of a corresponding assertion. The results are visualized for each single build and in a build history. The build history plots for each performance tests the results in a chart. Furthermore, performance tests are comparable, if they were measured for the same run mode and timeunit. This plugin is named *RadarGun-Reporting* and is hosted in Jenkins' repository on GitHub.

The second plugin was developed for the integrated development platform Eclipse. Developers can run performance tests manually. Therefor, we provide a launch configuration and a view, similar to the unit testing framework JUnit. Finally, we demonstrate by an example how to write performance tests and execute them with RadarGun.

Contents

1 Introduction **1**
1.1 Motivation . . . 3
1.2 Document Structure . . . 4

2 Goals **7**
2.1 G1: Improve the Performance Testing Framework RadarGun . . . 7
2.2 G2: Develop a Jenkins Plugin to Execute and Visualize RadarGun . . . 8
2.3 G3: Develop an Eclipse Plugin to Execute and Visualize RadarGun . . . 8
2.4 G4: Write Performance Tests for Kieker . . . 8
2.5 G5: Feasibility Evaluation . . . 8

3 Foundations and Technologies **11**
3.1 Foundations . . . 11
3.1.1 Performance Influences in Java Programs . . . 11
3.1.2 Handling Performance Influences by Microbenchmarking . . . 12
3.1.3 Defining a Performance Test . . . 14
3.1.4 Analyzing Performance Tests Statistically . . . 15
3.2 Utilized Technologies . . . 19
3.2.1 The Integrated Development Environment Eclipse . . . 19
3.2.2 The Java Benchmarking Harness (JMH) . . . 19
3.2.3 The Pipe-and-Filter Framework TeeTime . . . 21
3.2.4 The Monitoring Framework Kieker . . . 21
3.2.5 The Performance Testing Framework RadarGun . . . 21
3.2.6 The Continuous Integration Environment Jenkins . . . 23
3.2.7 The Javascript Plotting Framework CanvasPlot . . . 24

4 Enhancing the Performance Testing Framework RadarGun **27**
4.1 Improving the Pipe-And-Filter Architecture . . . 27
4.2 Separating Performance Test Configurations from Benchmark Configurations 30
4.3 Creating an Import/Export Model for Performance Test Results . . . 31
4.4 Supporting Progress Monitoring . . . 38

5 Reporting Performance Tests in Jenkins **41**
5.1 Understanding the Stapling of Pages in Jenkins . . . 41
5.2 Understanding the Rendering of Objects in Jenkins . . . 44
5.3 Providing a Build Pipeline Step . . . 46

5.4 Configuring a Build Pipeline 49
5.5 Reporting a Single Build 51
5.6 Reporting a Build History 52

6 Reporting Performance Tests in Eclipse **55**
6.1 Understanding the Eclipse Rich-Client-Platform 55
6.2 Providing a RadarGun Launch Configuration in Eclipse 56
6.3 Reporting Performance Test Results in Eclipse 58
6.4 Visualizing Performance Test Results in Eclipse 59

7 Application Example of RadarGun **61**
7.1 Understanding the Benchmark Configuration by JMH 61
7.2 Defining Performance Tests in RadarGun 63
7.3 Writing Performance Tests for the Kieker Framework 63

8 Feasibility Evaluation **67**
8.1 Evaluating the Machine Identification 68
8.1.1 Methodology and Test Scenarios 68
8.1.2 Results and Discussion 69
8.1.3 Threats to Validity 69
8.2 Evaluating the Progress Monitoring 70
8.2.1 Methodology and Test Scenarios 70
8.2.2 Results and Discussion 71
8.2.3 Threats to Validity 80
8.3 Evaluating the Visualizing of a Build History 81
8.3.1 Methodology and Test Scenarios 81
8.3.2 Results and Discussion 82
8.3.3 Threats to Validity 85

9 Related Work **89**

10 Conclusions and Future Work **91**

Bibliography **97**

Appendix **101**

Chapter 1

Introduction

Software engineering becomes a more and more continuous development process. The complexity of software products and its code increase and even minor changes of a single team member can affect the whole product. To avoid integration failures, different approaches were proposed over the past two decades. Although Booch [9] has already named and proposed *Continuous Integration (CI)* in 1995, it took over one decade to get reasonable acceptance for common paradigms for CI. In 2006 Martin Fowler introduced this paradigms for CI in his paper *'Continuous Integration'* [17]. Continuous integration is a development practice helping software engineers to develop software and updates bit by bit. Usually, each team member integrates his bit of work at least daily. Thus, there are multiple integrations daily. To detect errors as fast as possible, each integration is verified by an automated build, including different functionality tests. If a developer changes a method and the new code does not pass the functionality tests, the developer receive a quick feedback with the corresponding errors. Hence, in CI it is crucial to provide many different tests to verify as many functions as possible. If the system merely performs trivial tests, there is only a nominal validity in the whole build process. Hence, an extensively automatic testing is an essential feature in CI. In essence, CI helps to optimize the software engineering process and reduces integration problems [17].

While functionality tests are already covered by continuous integration environments, performance tests are not. In 1991 a first standard ($ISO9126$) for software quality was established by the International Organization for Standardization (ISO). This standard was refined to $ISO/IEC25010$ [26] and an overview is given in Figure 1.1. Eight categories and 31 sub-categories are defined to evaluate the quality of software. These standards are provided in a generic way and are not very detailed. Nevertheless, they help developers to consider miscellaneous factors to enhance their products. As shown in Figure 1.1, a program's performance is a quality factor. Especially, time behavior and resource utilization are still not covered properly by CI environments. Time behavior can be, e.g., execution times, throughput, or response time. In this master's thesis we focus on execution times in the context of the programming language *Java*. Java is processed in Java bytecode and executed by the Java Virtual Machine (JVM). There are different factors that influence a Java program's performance, e.g., garbage collection and the Just-In-Time (JIT) compiler, which recompiles and optimizes the Java bytecode at runtime. The optimization process is based on gained knowledge about the execution behavior. Hence, the performance of program sections can differ significantly from run to run [23]. To measure the performance

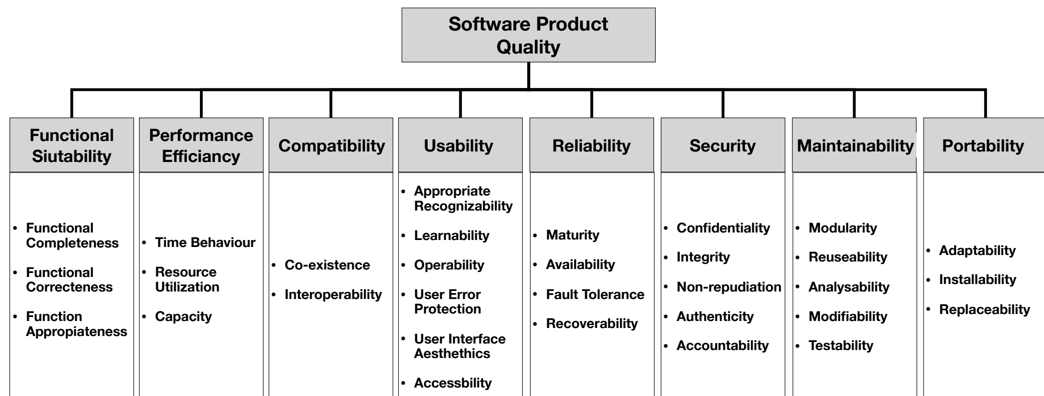


Figure 1.1. ISO/IEC 25010 standards for software quality [25]

of program functions there are different solutions. A trivial approach is to place timestamps before and after the execution of a program section and compute the difference of both end and start to obtain the runtime. In Section 3.1.2 we describe the disadvantages and pitfalls of this and further approaches to benchmark runtime. To measure performance properly, we utilize the microbenchmarking framework Java-Microbenchmarking-Harness (JMH) to define benchmarks. However, benchmarking alone is still not sufficient for performance testing. A benchmark becomes a performance test, if and only if it is compared against predefined assertions. Due to the different runtime of program sections from run to run, this assertions are *intervals*, e.g., time intervals with lower and upper bounds for the runtime. Thus, we utilize the performance testing framework RadarGun [21], which is based on JMH for running microbenchmarks and supports an automatically result comparison against predefined and hardware dependent assertions for the runtime.

Since we are in the context of Java, we utilize Jenkins as CI system (see Section 3.2.6). The advantage of Jenkins over other CI systems, e.g., Bamboo, is that Jenkins is community driven under the Creative Commons Attribution Share-Alike license and everyone can easily install custom plugins. Our Jenkins system uses JUnit [31] as unit testing framework. JUnit verifies the correctness of a process and its result. To test for the correct results, JUnit uses predefined assertions. If the result does not meet the assertion, either the assertion is wrong or the tested method. However, there still is no tool in Jenkins for automatic performance testing.

With this master's thesis we make the following contributions: First, we enhance the performance testing framework RadarGun, e.g., to allow a statistically rigorous analysis of the measurements as suggest by Georges, Buytaert, and Eeckhout [18]. Second, we introduce the enhanced RadarGun into the continuous integration system Jenkins to automatically execute and visualize performance tests. Third, we introduce RadarGun to

the IDE Eclipse and thus, enable software engineers to do performance testing during the development process on a local machine. Finally, we demonstrate how to write performance tests for two different frameworks.

1.1 Motivation

Among the correct execution of a program, its performance is an important factor. With the growth of programs and services, loading and process times can increase. For example, if a developer changes a method and builds, unknowingly, a bottle-neck that slows down the entire program by 10 %, this should be highlighted during the build process. To address this issue, different frameworks were developed (discussed in Chapter 9). However, none of these frameworks is a proper performance testing framework. Although they all measure the runtime, they often still ignore program influences, e.g., the JIT compiler. Hence, there is a threat to the validity of these measurements. Additionally, there are no possibilities to define assertions to compare the result against each other. Moreover, all these frameworks have in common that they do not involve the hardware on which performance tests are executed. However, the hardware is a crucial factor for the performance of an application. Operations can be handled differently on different hardware and faster CPUs manage instructions in less time than slower CPUs.

In 2017, a performance testing framework named RadarGun [21] was developed at the University of Kiel. This performance testing framework is open-source and freely available. Henning, Wulf, and Hasselbring [21] define requirements for a performance testing tool:

(1) Automatically and repeatedly execute the JVM and the program section of interest.
(2) Automatically aggregate measurements to a single, representative measurement score, e.g., the minimum, the median, the maximum, or the average.
(3) Automatically differentiate between different machines on which the measurements are collected.
(4) Use assertions to check whether the measurement score is within a time interval.

The first prototype performs hardware-dependent benchmarks and compares them against predefined assertions. Thus, we use the potential of RadarGun to integrate it into an continuous integration environment and help software engineers to create products with a better quality management. Nevertheless, this prototype has to be enhanced. Although RadarGun is built upon a Pipe-And-Filter architecture by utilizing the Pipe-And-Filter framework TeeTime (see Section 3.2.3), the execution of benchmarks blocks the whole process until all benchmarks have finished. Only after all benchmarks finished, they are compared against the predefined assertions. Hence, there is no proper progress monitoring. Thus, we enhance the architecture of RadarGun to break up the blockade and run and report each performance test one by one. Thereby, the CI system is able to abort the build process and report the feedback to the developer immediately after a performance test has failed. To increase the validity of the performance tests, we also implement a statistically rigorous analysis, which is suggested by Georges, Buytaert, and Eeckhout [18], into the

result comparisons made by RadarGun. Therefor, we compute a confidence interval to a predefined confidence level for each result and compare the predefined assertions against this confidence interval. If a confidence interval undercuts or exceeds the lower or upper bounds of an assertion, the performance test has failed. Thereby, we handle outliers, and consequently limit the change of good and bad coincidence.

The advantage of using performance tests in automated builds in CI systems is the storage of builds and the regularly restarting automatic build process. We collect a history of all performance tests for all builds and are able to detect performance issues in a specific build. Thereby, a performance trend of our product can be easily visualized. RadarGun is executable by Jenkins and also by developers via a console. However, it solely exports performance tests results measuring the average runtime per execution. A visualization is missing entirely. Hence, we need to enhance this tool to be able to applicate it in a CI environment. Consequently, we face the additional requirements

(5) Visualize the test results in a CI environment.
(6) Group performance tests in the visualization.
(7) Visualize performance tests with different run modes and measured timeunites.
(8) A possibility to manually execute and visualize performance tests.

Thus, we require tools that can utilize RadarGun, execute performance tests, and report the results in a CI environment. An overview of our envisioned approach can be seen in Figure 1.2. The enhanced RadarGun framework can be used in two different stages of the development process. On the one hand, software engineers are able to run performance tests manually. On the other hand, CI servers can run performance tests automatically during the build process. Software engineers receive an output containing the performed tests and results. In CI environments we obtain a history of builds and visualized results. Additionally, the comparison of performance tests and builds can be done in CI environments.

1.2 Document Structure

The goals of this master's thesis are presented in Chapter 2. We aim to improve the performance testing framework RadarGun to a JUnit like framework for hardware-dependent performance testing. In Chapter 3, we present foundations and technologies related to our work. The enhancement of the performance testing framework RadarGun is accompanied in Chapter 4. To allow the usage of RadarGun in a CI environment we create tools that are usable in Jenkins and Eclipse. Therefor, we develop a Jenkins plugin that utilized RadarGun to report and visualize performance tests as described in Chapter 5. Additionally, the framework RadarGun is integrated into an Eclipse plugin to report and visualize the results of RadarGun during runtime in Eclipse as described in Chapter 6. In Chapter 7 we demonstrate by an example how to write performance tests properly. We focus on a detailed evaluation of our framework. In Chapter 8 we present and discuss our feasibility evaluation. Finally, we discuss the related work to our approach in Chapter 9 and give a

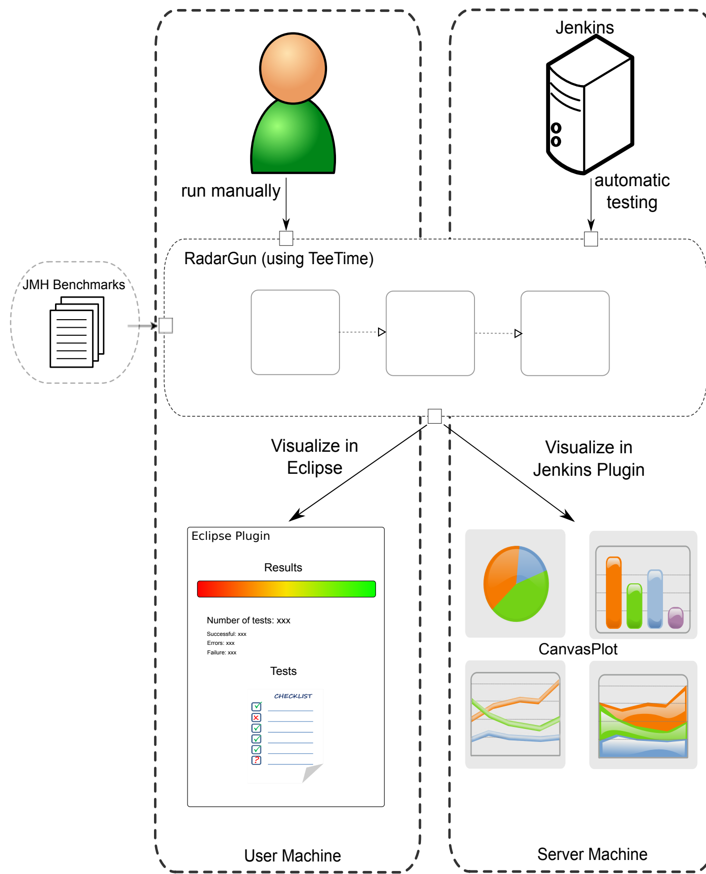


Figure 1.2. An overview of our approach to execute and visualize performance tests in a CI environment

conclusion and features for future work in Chapter 10.

Chapter 2

Goals

Although, there are different benchmarking tools to analyze Java programs and visualize these results in a CI system, e.g., Jenkins (see Section 3.2.6), none of these tools tests the performance of Java programs hardware-dependent, as described in Chapter 9. Since the hardware a program is executed on affects the program's performance, this is a crucial factor in performance testing. Henning, Wulf, and Hasselbring [21] define requirements for a performance testing tool, here we extend these requirements to report performance tests in a continuous integration environment, see Section 1.1. Additionally, we enhance the parts, which are not well-implemented by Henning, Wulf, and Hasselbring [21].

2.1 G1: Improve the Performance Testing Framework RadarGun

We enhance the performance testing framework RadarGun to satisfy the requirements couched by Henning, Wulf, and Hasselbring [21]. Therefor, we improve its internal architecture to extend the output by RadarGun. This improvement contains (1) an export model for serialization and deserialization of results, (2) a statistically rigorous analysis of performance test results, (3) a progress monitoring in real time, (4) and the separation of performance test configurations from benchmark configurations. This improvement of RadarGun's architecture is needed, since it executes all benchmarks before reporting the single results. Thus, a progress monitoring in real time is not possible. Furthermore, the performance test configuration and the exported data can not be divided from the benchmark configuration. Thus, we split up the benchmark execution stage to report a benchmark's result before executing the next one. To increase the validity of performance tests, we add a statistically analysis to the comparator stage. This prevents the test from failing, if 1 out of 100 execution does not meet the asserted time, due to a process that corrupts the runtime. Furthermore, a statistically rigorous analysis reduces the possibility that an automatic testing system or a user draws wrong conclusions from the results of a performance test.

2.2 G2: Develop a Jenkins Plugin to Execute and Visualize RadarGun

Due to the evolution of programs managed in CI environments, the utilized data structures and changed program code can influence the runtime of these programs. Developers want to see whether a new data structure's efficiency performs better, worse or same as before. Thus, we develop a plugin to utilize RadarGun for automatic performance testing in the CI environment Jenkins. We utilizes Jenkins' User Interfaces (UI) to report and visualize performance tests. The reports can be shown for single builds or a set of builds. Furthermore, a history of all performance tests for all builds will be bundled into one overview page for each project. Additionally, this feature allows to compare equivalent tests on different versions of a program.

2.3 G3: Develop an Eclipse Plugin to Execute and Visualize RadarGun

RadarGun lacks of a comfortable way to be used in the development process. So far, all tests have to be run in a console or from within the IDE by providing a dedicated Main class. To allow the usage of RadarGun in the development process of software engineers, we develop a plugin for Eclipse that visualizes the reported test results similar to its unit test plugin JUnit. Therefore, we include an extra view for RadarGun in Eclipse and support the configuration of custom launches. The results will be reported and visualized in real time.

2.4 G4: Write Performance Tests for Kieker

In *The Performance Testing Framework RadarGun* the performance test framework RadarGun was proposed and evaluated with performance tests for the Pipe-And-Filter Framework TeeTime. To evaluate a second framework, that is, to increase the external validity, we write performance tests for the monitoring framework Kieker and execute them with RadarGun. Since Kieker already utilizes the monitoring overhead benchmarking framework "Performance Benchmarking of Application Monitoring Frameworks" [54], we will compare our results with the results reported by "Performance Benchmarking of Application Monitoring Frameworks".

2.5 G5: Feasibility Evaluation

After reaching the first three goals, we introduced the performance testing framework RadarGun to a CI environment for automatic testing and the IDE Eclipse for manually

testing. Thus, we have to ensure that our integration of the different tools works correctly. Due to a lack of possibilities to automatically test our approach, we conduct a feasibility study to show our environment works correctly. This evaluation is performed on at least two different systems.

Chapter 3

Foundations and Technologies

In the following, we introduce the theoretical and technical background required to understand our approach for reporting performance tests in a continuous integration environment. We utilize a composition of different frameworks, which solve subtasks, in the CI environment. The theoretical background is introduced in Section 3.1. The frameworks we utilize are introduced in Section 3.2.

3.1 Foundations

As described in Section 1.1, our entire development environment is situated in the context of the programming language Java. Although, various categories in Figure 1.1 are already covered by automatic testing in a CI environments, performance testing is not. Especially, time behavior and resource utilization are still not covered properly. Time behavior can be, e.g., execution times, throughput, or response time. In this master's thesis we focus on execution times. To report performance tests, the performance tests have to be executed first. Due to different influences, the performance of Java programs can differ from run to run. In the following, we introduce some core aspect we have to consider to measure performance properly. Subsequently, we discuss how to avoid some pitfalls in performance testing and how to analyze the results of performance tests.

3.1.1 Performance Influences in Java Programs

One factor that influences a program's performance is the hardware the program is executed on. The heart piece of each computer system is the Central Processing Unit (CPU). A CPU can execute solely a few specific instructions, which are called assembly or binary code. Thus, all applications must be translated into these instructions before a CPU is able execute them [44]. Since writing binary or assembly code can be quite laborious, different programming languages were developed, allowing us writing code similar to giving orders in our natural language. The programming language Java is taking advantage of the platform independence of scripting languages and the better performance of compiled languages. Applications are compiled into an assembly language, the Java bytecode. This Java bytecode is then interpreted by the *Java Virtual Machine (JVM)* giving it the advantage of an interpreted language [44].

Just-in-Time Compilation (JIT) [44] is part of the JVM and compiles Java bytecode into machine code at run time after the application has started. Hence, methods are not compiled until the first time they are called. The JVM maintains a call count for each method that is called and increments it every time the method is called. Until its call count does not exceed a specific threshold for JIT compilation, the method is interpreted but not compiled by the JVM. Frequently used methods are compiled soon after the application and JVM started. Afterwards, the call count will be reseted and the procedure starts from the beginning. Reaching the threshold again leads to a recompilation with further optimizations of the method. Less-used methods are compiled later or are sometimes never compiled [24]. Thus, Java benefits on the one hand from the platform independence of interpreted languages, on the other hand of the performance of compiled languages. All in all, the JIT compilation helps to balance startup times and long term performance. Due to Oaks [44], the compiler influences the performance of a JVM most.

Warm-up Time defines the time the JVM needs, e.g. for class loading and bytecode interpretation. Starting the JVM the first time leads to possibly thousands of method calls. Compiling all of these methods can significantly affect startup time. Lion et al. [41] found that this class loading and bytecode interpretation is a recurring overhead and can be a performance bottleneck. Notice that most of the JIT compilation is performed in the warm-up phase.

Garbage Collection is the automatic memory management done by Java. This is one of the advantages of Java over other programming languages like *C* or *C++*. *Garbage Collection (GC)* describes the process of reclaiming memory occupied by objects that are no longer in use by the application. The GC identifies which objects are in use and which are not and reclaims the memory of the unused method by deleting it. An object is in use, if some part of an application still maintains a pointer to that object. An object that is no longer referenced by any part of the application is unused [47].

Heap Size represents the size of a repository that collects referenced objects, unused objects, and free memory. Unused objects are ready for garbage collection. This heap size can be specified manually. On the one hand, for a large heap size full garbage collection is slower. On the other hand, it occurs less frequently. If the heap size is small, full garbage collection is faster, but occurs more frequently.

3.1.2 Handling Performance Influences by Microbenchmarking

Benchmarking is used to compare and analyze the results or processes of program execution based on specified measurement criteria. If we measure, e.g., the runtime of a program, we receive the time elapsed from start to end of the execution. This leads to a metric allowing us to compare the execution times of a program. Due to many influences possibly

corrupting the runtime of an application, we do not want to measure the execution time of an application as a whole, but the execution time of single methods. Hence, we intend to use microbenchmarks. Microbenchmarking is used to measure the runtime of small segments of code, although the size of these small segments is not defined clearly.

Java allows to measure runtime in a trivial way. In the following, we discuss two different approaches to measure the runtime with Java and explain why they do not measure runtime exactly. For this purpose, we generate pseudo random numbers with the *Random* class provided by Java and measure the execution time of the function generating this random numbers.

```
1 final Random random = new Random();
2 final long start = System.nanoTime();
3   random.nextInt();
4 final long end = System.nanoTime();
5 System.out.println(end - start);
```

Listing 3.1. A first approach to measure execution time.

A first naive approach is shown in Listing 3.1. We just put timestamps before (Line 1) and after (line 4) the random number has been generated (Line 3). The difference of `end` and `start` (Line 5) represents the time past between both timestamps. Taking account of Section 3.1.1, we can not ensure that this measured execution time represents the real execution time. For example the execution could have taken more time, due to the warm-up of the JVM. This approach considers none of the performance influence we describe in Section 3.1.1. A machine's background processes can always decelerate a method. However, there are even worse negative factors, since there is no warm-up time for the JVM. Thus, the JIT compiler does not optimize the method and following the Java bytecode, yet not the machine code, is interpreted. Since the executing machine code is faster than executing Java bytecode, this impairs the performance of any method. Consider Listing 3.2 which shows an improved measurement approach.

The measurement is outsourced into a separate method (Line 3) which can be executed several times to warm-up the JVM. Furthermore, the loop generates random numbers 100.000 times (Line 6 – 8). Consequently, there are many more measurements than in the first approach in Listing 3.1. This reduces random factors that impair the method. However, the JIT compiler could still optimize the generation of a random number, e.g. by generating some random numbers inline without iterating the loop 100.000 times. This distorts the comparing metric, since all the iterations do not necessarily take 100.000 times longer than one iteration.

```
1  final Random random = new Random();
2
3  void testPerformanceRandom() {
4     final long start = System.nanoTime();
5
6     for(int i = 0; i < 100000; i++) {
7        random.nextInt();
8     }
9
10    final long end = System.nanoTime();
11    System.out.println(end - start);
12 }
```

Listing 3.2. A better approach to measure execution time.

The key disadvantage of the second approach is the lack of isolation between methods with the same purpose, yet different implementations. If we want to compare the random number generation with two different implementation of this method (e.g., *Random* and *SecureRandom*) and we run them one after another, than one of the methods could leave garbage and an active GC distorts the whole measurement. This problem is solvable with a sophisticated thread management. Therefore, we use the open-source framework Java Microbenchmarking Harness (JMH) (see Section 3.2.2) that considers all this problems by running benchmarks in separate threads. However, we still have to consider this and other pitfalls, if we aim to test performance of an application correctly. We demonstrate in Chapter 7 by an example how to write a performance test properly.

3.1.3 Defining a Performance Test

We already mentioned in Chapter 1 that benchmarks are no proper performance tests. Benchmarks are performed to measure the execution time of methods, neither more, nor less. To judge whether a method performs well or not, we require a reference point to compare the given results with. When performing such benchmarks, we intuitively compare the measured time with a time we expect for this method to run. This comparison we do in our minds. Hence, we generate the result whether a execution performed as expected or not, ourself and not computer that executes the benchmark. Moreover, other developers do not know which execution time we expect for benchmark, if we do not communicate this time. Therefor, we define that a performance test needs an assertion for the execution time a benchmark has to meet. Whether the benchmark meets this assertion or not, decides whether the benchmark was successful or has failed.

Since, a benchmark's performance can always differ from run to run [23], we need a metric that gives us evidence, if a benchmark performed well or not. Two good metrics

are *throughput* and *execution time per task*. When measuring the throughput, we want to know how often a task can be executed in a given time. Measuring the execution time per task is the exact opposite of measuring the throughput. Measuring the execution time per task computes how much time a task needs for one single execution. However, there are more metrics to measure performance. Nevertheless, all of them have in common that they can be evaluated by a cardinal scale. Unlike functional tests, the assertion can not be a single value the test's output can be tested against. We mentioned and discussed different performance influences in Section 3.1.1. By properly avoiding performance influences the deviation of single method executions from the mean of all executions does not vary too much. However, a method can return prematurely, due to an invalid argument exception, e.g., by code changes. Thus, it takes less time (or more executions) than usually. Vice versa, a new and slower mechanism can suddenly impair the whole method. Thus, the assertion for a runtime should specify an expected lower and upper bound. Therefore, we retain that a assertion for performance testing is a closed interval with lower and upper bounds on the metric we are measuring.

When testing the *execution time per task* of a method with a given assertion, the time unit, which we decide for, is important for the comparison and comprehensibility. If we measure a task's average runtime in milliseconds, yet compare it against the assertion interval given in seconds, this can lead to wrong conclusions. Due to rounding errors, it is possible that we corrupt our measurements this way. Vice versa, if we measure the time in seconds and define assertions in milliseconds, for example the interval $[123456.789, 987654.321]$, the computer can convert the units and compare them. However, we can not compare the results intuitively. Instead we must convert this values ourself, since a comparison of two different units is more complicated and fault-prone. Therefor, we retain that the time unit we measure is exactly the same unit the assertions are defined for.

A crucial factor for the runtime and number of computations is the hardware a computation is performed on. A faster CPU executes more operations in less time and thus, a benchmark performs better. Vice versa, a benchmark performs worse on a slower CPU. Thus, performance tests have to consider the hardware they are performed on and hence are always hardware-dependent.

In summary, a performance test is a benchmark whose result is compared against an hardware-dependent assertion. Assertions are intervals with a lower and upper bound and are defined for exactly the same unit which the benchmark is defined for.

3.1.4 Analyzing Performance Tests Statistically

To reduce the drawing of incorrect conclusions from performance test results, a statistically rigorous data analysis of the results is recommend by Georges, Buytaert, and Eeckhout [18]. The goal is to eliminate wrong conclusions from the measured data as good as possible. Therefore, we classify errors in the measured data in two main groups: *systematic errors* and *random errors*. Within the scope of performance testing *systematic errors* occur on mistakes in the design of benchmarks. This leads a *bias* into the measurements, which

can corrupt the accuracy of the results. Furthermore, these errors can not be eliminated by the statistically analysis, since it is up to the benchmark writer to control and prevent *systematic errors* [18]. The errors which benchmark writers are unable to control or prevent are *random errors*. These errors are unpredictable and non-deterministic [18]. Since they can corrupt performance tests in way to increase or decrease the measured runtime, they are unbiased. In Section 3.1.1 we describe some sources that influence a program's performance. Although, we describe in Section 3.1.2 how to handle performance influence by proper microbenchmarking, it is not always possible to prevent outliers, due to unexpected events in the system the tests are performed on. Hence, outliers need to be evaluated whether they are a result of an unexpected external event. In this case they may corrupt the data and thus need to be handled. Georges, Buytaert, and Eeckhout [18]"[...] advocate discarding outliers and applying statistically rigorous data analysis to the remaining measurements".

Since we can not discard outliers in our measurements (see Section 3.2.2), we require a proper statistical model to handle unpredictable *random errors*. For this purpose, we use confidence intervals to draw better conclusions from the measurements. A confidence interval is a statistical interval estimator that is computed from the measured data. These intervals are closely related to statistical significance testing. If a corresponding hypothesis test is performed, the significance level is the probability of rejecting the null hypothesis when it is true. For example, a null hypothesis implies there is no difference between a population mean and a sample mean. If the null hypothesis is rejected, there is a significant difference between both means. The confidence level is the complement of the level of significance. A confidence level says, if the same population is sampled on numerous occasions and interval estimates are made on each occasion, the null hypothesis is accepted with the given level of confidence. E.g., a 95% confidence level reflects a level of significance of 5%. Thus, if the null hypothesis is not reject, the resulting intervals would bracket the true population parameter in approximately 95% of the cases.

Note that the desired level of confidence is set by the tester and not by the measured data. We use confidence intervals to draw better conclusions from performance tests. Since we do not execute an infinity number of measurements, we do not know the true mean of a method's performance. Thus we have to estimate, whether our measured mean is near the true mean or not. This estimation is done by confidence intervals. In the following, we give a mathematical introduction to confidence intervals and how they are estimated.

Estimating the confidence interval

The definition of confidence intervals is based on [27]. We are given a random experiment $\mathcal{E}(\chi, (W_\theta)_{\theta\in\Theta})$ with sample space $\chi = (X_1, ..., X_n) \subseteq \mathbb{R}^d$, $X_1, \ldots, X_n$ are random variables, and a family of probability measures $(W_\theta)_{\theta\in\Theta} = (P_\theta^X)_{\theta\in\Theta}$. Let Θ be the parameter space, which holds the unknown parameters on which the distribution of X depends. For any experiment outcome ω, $X(\omega)$ is referred to as the data. In the following, we will identify the observed samples $x \in \chi$ as its realization. $(P_\theta^X)_{\theta\in\Theta}$ is a family of probability distributions. Further, let $\mathcal{P}(\Theta)$ be a power set of Θ.

Now we want to specify the experiment. First, let $\chi_1 \subseteq \chi$, $\chi_1 \subset \mathbb{R}$ be our sample and $\theta = (\mu, \sigma_0^2) \in \Theta \subset \mathbb{R} \times \mathbb{R}$ holding the unknown mean μ of χ and for the moment assumed to be known variance σ_0^2. The random variables $X_i \in \chi_1$ are independent and identically $\mathcal{N}(\mu, \sigma_0^2)$-distributed. For each μ we are using the two-tailed Gaussian test, to test if the null hypothesis $H_{0_\mu} = \{\mu\}$ and the alternative hypothesis $H_{1_\mu} = \{b : b \neq \mu\}$ satisfy

$$\Phi_\mu(x) = \begin{cases} 1, & \frac{\sqrt{n} \cdot |\overline{x}_n - \mu|}{\sqrt{\sigma_0^2}} > u_{\frac{\alpha}{2}} \\ 0, & \frac{\sqrt{n} \cdot |\overline{x}_n - \mu|}{\sqrt{\sigma_0^2}} \leqslant u_{\frac{\alpha}{2}} \end{cases} .$$

For any $x_i \in \chi_1$ the mean $\overline{x}_n$ of all the measurements is defined as

$$\overline{x}_n = \frac{1}{n} \sum_{i=1}^{n} x_i \, .$$

Since we are given a test with significance level $\alpha \in (0, 1)$, one has

$$1 - \alpha = W_{(\mu, \sigma_0^2)}(\{x : \Phi_\mu(x) = 0\}) = W_{(\mu, \sigma_0^2)}(\{x : |\overline{x}_n - \mu| \leqslant \sqrt{\frac{\sigma_0^2}{n}} u_{\frac{\alpha}{2}}\}) \, .$$

Hence, we define

$$c_1(x) = \overline{x}_n - \sqrt{\frac{\sigma_0^2}{n}} u_{\frac{\alpha}{2}} \quad \text{and} \quad c_2(x) = \overline{x}_n + \sqrt{\frac{\sigma_0^2}{n}} u_{\frac{\alpha}{2}} \, . \tag{3.1}$$

and get for any $\mu \in \mathbb{R}$:

$$W_{(\mu, \sigma_0^2)}(\{x : c_1(x) \leqslant \mu \leqslant c_2(x)\}) = 1 - \alpha \, .$$

By consequence, any unknown value of μ is located with a probability of $1 - \alpha$ in the interval $[c_1(x), c_2(x)]$. Notice that the values for $u_{\frac{\alpha}{2}}$ in general are obtained from a precomputed table for Gaussian distributed values like in [50].

Since we do not know the true mean and variance of our performance tests, one has to exchange the Gaussian two-tailed test with the two-tailed t-test. Again, the random variables $X_i \in \chi_1$, $1 \leqslant i \leqslant n$, are independent and identically $\mathcal{N}(\mu, \sigma^2)$-distributed. In this case the variance σ is unknown. The null hypothesis is, again, given as H_0 and the alternative hypothesis as H_1. To determine a confidence interval, H_0 and H_1 must satisfy

$$\Psi_\mu(x) = \begin{cases} 1, & \frac{\sqrt{n} \cdot |\overline{x}_n - \mu|}{\sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (\overline{x}_n - \mu)}} > f(t_{n-1}, \frac{\alpha}{2}) \\ 0, & \frac{\sqrt{n} \cdot |\overline{x}_n - \mu|}{\sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (\overline{x}_n - \mu)}} \leqslant f(t_{n-1}, \frac{\alpha}{2}) \end{cases} .$$

Thus, one has

$$\begin{aligned} c_1'(x) &= \overline{x}_n - \sqrt{\frac{\sum_{i=1}^{n}(x-\overline{x_n})^2}{n(n-1)}} f(t_{n-1}, \frac{\alpha}{2})\,, \\ c_2'(x) &= \overline{x}_n + \sqrt{\frac{\sum_{i=1}^{n}(x-\overline{x_n})^2}{n(n-1)}} f(t_{n-1}, \frac{\alpha}{2})\,, \end{aligned} \tag{3.2}$$

whereas one has for any $\mu \in \mathbb{R}$ and any $\sigma^2 > 0$:

$$W_{(\mu,\sigma_0^2)}(\{x : c_1'(x) \leqslant \mu \leqslant c_2'(x)\}) = 1 - \alpha\,.$$

Notice that the values for $f(t_{n-1}, \frac{\alpha}{2})$ in general are obtained from a precomputed table for Student's t-distributed values like in [50].

Example for calculating a confidence interval

After introducing the confidence interval mathematically, we now translate theory into practice. When using a common probability measure $\Pr[c_1 \leqslant \mu \leqslant c_2) = 1 - \alpha$ for an unknown mean μ and the variance σ, one has exactly the definition used by Georges, Buytaert, and Eeckhout [18]. μ and σ are unknown, because we neither know the true mean of a method's performance nor the standard deviation. Thus, we have to estimate, whether our measured mean and standard deviation are in range of μ and σ or not. Georges, Buytaert, and Eeckhout [18] distinguish in their method to calculate confidence interval between the sizes of a given sample. If a sample's size is $n \geqslant 30$, they assume a Gaussian distribution of the data, due to the central limit theory. Then the confidence interval can be estimated by Equation 3.1. In case of $n < 30$ they assume a Student's t-distribution and the confidence intervals can be calculated by Equation 3.2. Since the microbenchmarking framework JMH (see Section 3.2.2) exclusively uses the Student's t-test and hence the t-distribution to compute confidence intervals [48], we can solely give an example using Equation 3.2.

We are given a sample of 30 executions of a performance test (the sample's size is $n = 30$) and the average runtime $\overline{x}_n = 24.378\,\text{ns}$ (nanoseconds), standard deviation $\sigma = 1.228\,\text{ns}$, and we define a confidence level $\gamma = 0.95$. Hence, one has the level of significance $\alpha = 1 - \gamma = 0.05$. Then one has the following confidence interval for the confidence level γ:

$$\begin{aligned} c_1 &= \overline{x}_n - f(t_{n-1}, \frac{\alpha}{2}) \cdot \frac{\sigma}{\sqrt{n}} = 24.378 - f(29, 0.025) \cdot \frac{1.228}{\sqrt{30}} \\ &\approx 24.378 - 1.699 \cdot 0.224 \\ &\approx 23.997\,, \end{aligned}$$

$$c_2 = \overline{x}_n + f(t_{n-1}, \frac{\alpha}{2}) \cdot \frac{\sigma}{\sqrt{n}} = 24.378 + f(29, 0.025) \cdot \frac{1.228}{\sqrt{30}}$$
$$\approx 24.378 + 1.699 \cdot 0.224$$
$$\approx 24.759\,.$$

By conclusion, the true mean of the performance test's runtime is with a probability of 95% within the confidence interval [23.997, 24.759]. Notice that we obtained the value $f(t_{n-1}, \frac{\alpha}{2}) = 1.699$ from the precomputed table for the Student's t-distribution in [50] with confidence level $\gamma = 0.95$ and $n - 1 = 29$ degrees of freedom.

3.2 Utilized Technologies

In this master's thesis, we report performance tests in a CI environment build upon infrastructure in the programming language *Java*. Thus, we assume a good knowledge of Java. In the follow, we briefly introduce the utilized frameworks.

3.2.1 The Integrated Development Environment Eclipse

Eclipse [16] is an IDE released in 2001 by, in the present known as, the Eclipse Foundation and is one of the most used Java IDEs. The Eclipse software development kit (SDK) is free and open-source software, released under the terms of the Eclipse Public License [16]. It is operating cross platforms, e.g., Linux, macOS, Solaris, and Windows. A selectable base workspace and an extensible plugin system allows to customize the environment. Although Eclipse is mostly written in Java and is mostly used to develop applications in Java, it may also be used to develop applications in other programming languages, e.g., C, C++, PHP and Erlang. Its run-time system is based on Equinox [42]. Equinox is an implementation of the OSGi R5[1] core framework specification [42]. With the release of version 4.2, called Juno, Eclipse was restructured to allow the development of custom client applications. We introduce the Eclipse Rich-Client-Platform in Section 6.1. The old IDE was migrated into an compatibility layer to allow old plugins to still run on new Eclipse versions. This is described in more detail in Section 6.1.

3.2.2 The Java Benchmarking Harness (JMH)

In Section 3.1.2 we mentioned different pitfalls in microbenchmarking. The open-source framework Java Benchmarking Harness (JMH) [45] is part of the open-source framework OpenJDK and is a Java harness for building and running benchmarks. Different modes like *throughput* and *time per task* can be measured. It supports a granularity of time in nano-, micro-, milli-, and seconds. Due to its annotation based concept, the customization for performance benchmarking is very flexible. Developers can, e.g., specify the number

[1] `https://www.osgi.org`

of separated execution environments, the number of executions, the run mode and the measured time unit. After performing a benchmark different statistics are computed and attached to the run results. We utilize the build-in method to compute the confidence intervals, which we described in Section 3.1.4. JMH utilizes the library Apache Commons Math [1] to calculate confidence intervals from the measurements. Notice that JMH uses the Student's t-distribution[2] to compute confidence intervals for a given confidence level. Furthermore, it does not eliminate outliers before computing the confidence interval. By default a confidence level of 99, 9% is reported by JMH. However, obtain the confidence interval for every confidence level we pass to JMH.

```
1  @Warmup(iterations=5)
2  @Measurement(iterations=50)
3  @Fork(2)
4  @BenchmarkMode(Mode.AverageTime)
5  @OutputTimeUnit(TimeUnit.NANOSECONDS)
6  @State(Scope.Benchmark)
7  @Thread(1)
8  public class PerformanceTester {
9
10    private final Random random = new Random();
11    private final SecureRandom secureRandom = new SecureRandom();
12
13    @Benchmark
14    public void testPerformanceRandom(BlackHole blackHole) {
15       blackHole.consume(random.nextInt());
16    }
17
18    @Benchmark
19    public void testPerformanceSecureRandom(BlackHole blackHole) {
20       blackHole.consume(secureRandom.nextInt());
21    }
22 }
```

Listing 3.3. Configuring a benchmark with JMH

In Listing 3.3 we introduce JMH to our previous example of benchmarking the generation of random numbers. This setup considers some warm-up iterations (line 1) and a number of measurements to perform (Line 2). Further, the random number generation is integrated into methods we perform benchmarks on. Each annotation *@Benchmark* (Line 12&17) defines

[2] http://hg.openjdk.java.net/code-tools/jmh/file/25d8b2695bac/jmh-core/src/main/java/org/openjdk/jmh/util/AbstractStatistics.java

a benchmark to perform. Notice that unlike the *Random*[3] class the class *SecureRandom*[4] generates cryptographically secure random numbers. Furthermore, it is important to consume the computed random numbers. Otherwise, the JVM possibly optimizes the random number generation based on the observation that the benchmark code does not make use of these [51]. Thus, JMH provides an object named `Blackhole`, which is used when it is not convenient to return a single object from a benchmark method [45]. In Section 7.3 we describe how to write benchmarks by an example we use for our feasibility evaluation in Chapter 8.

3.2.3 The Pipe-and-Filter Framework TeeTime

The Pipe-and-Filter (P&F) framework TeeTime [57] was developed at the University of Kiel. It is a framework to support the development of applications based on this style of architecture, e.g., applications processing streams of data. Users can create custom stages (filters), pipes, ports and whole configurations in an easy way. Additionally, it contains many primitive and composite ready-to-use stages. Teetime supports the possible performance improvement of P&F architectures, since it allows a single-threaded execution, with no overhead, or a multi-threaded execution, with minimal overhead [56]. The architecture of the abstract and generic filters makes it easy to implement our own stages and to connect them with other filters, in a type-safe way. Stages support incoming and outgoing ports. Pipes connect an outgoing port of one stage with the incoming port of a following stage. A configuration defines which filters are connected. In Section 3.2.5 we describe a framework we use for performance testing which is based on TeeTime.

3.2.4 The Monitoring Framework Kieker

The Kieker Monitoring Framework [22] was developed at the University of Kiel and is a framework for monitoring and analyzing the runtime behavior software systems. To monitor already compiled applications, Kieker uses aspect-oriented programming [37] in its configurations. The monitored data is persisted in files containing probes. A key requirement for Kieker is a preferably small overhead. Therefor, a performance testing tool for the overhead was introduced by Waller [54] named MooBench. Since Kieker is actively used and continuously developed, we write performance tests to test its performance automatically with RadarGun. We evaluate RadarGun by using performance tests for Kieker in Chapter 7.

3.2.5 The Performance Testing Framework RadarGun

The performance testing framework RadarGun [20] was developed in a student research project by Henning [19] at the University of Kiel. The testing mechanisms are based

[3] `https://docs.oracle.com/javase/8/docs/api/java/util/Random.html`

[4] `https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html`

on microbenchmarking with JMH. After performing microbenchmarks, the results are compared with predefined assertions regarding to a time interval the test should be performed in. The key advantage over other frameworks is the hardware-dependent performance testing. RadarGun allows the configuration, in YAML files, of hardware-dependent assertions and automatically detects the hardware the tests are performed on. The assertions have a similar purpose like in JUnit and consist of a lower and upper bound for the average runtime. If the performance test's average runtime does not meet the time interval given in the assertion, the test fails. The results are exported including the performance test's name, the average runtime, the assertion interval, and whether the test finished successfully or has failed, to CSV files. The test results are importable and can be plotted by external tools. In [19] the Jenkins plugin [13] visualizes the test results in a line plot, including lower and upper bounds. As shown in Figure 3.1 RadarGun

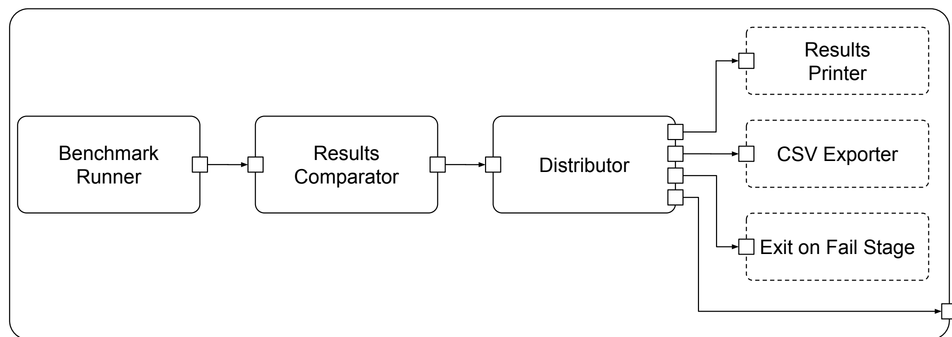


Figure 3.1. The architecture of the performance testing framework RadarGun [19]

is based on a P&F architecture. This architecture is realized with the P&F framework TeeTime (see Section 3.2.3). The `Benchmark Runner` stage executes all tests, before it passes the results further to the `Results Comparator` stage. This stage looks up the assertions for each performance test and compares the results with the assertions. Subsequently, a complete test result for each performance test is created and written in multiple outputs files, regarding the overall result, whether it has failed or not. Hence, no proper progress monitor can be shown.

Although Henning, Wulf, and Hasselbring [21] argue that RadarGun's performance test configurations are separated from the benchmark configurations, this is not well-implemented. Performance test are configured in a text file using the data format standard YAML [6]. In Listing 3.4 we show an example configuration. This file can contain several different configurations. Each configuration is separated by — (line 1), followed by three parameters. The `Identifier` (line 2) takes one of the possible identification strategies presents in Table 3.1. Since the performance tests are hardware-dependent, this identification is required. Identifiers such as the `MacAddressIdentifier`, require specific parameters, which are defined in `Parameters` (line 3), e.g., `['aa:bb:cc:dd:ee:ff']`. Since these parameters are

Table 3.1. Usable machine identifiers in performance test configurations.

Machine Identifier	*Required Parameter(s)*
`NetworkAddressIdentifier`	[IP or host name]
`MacAddressIdentifier`	[Mac Address]
`WindowsComputernameIdentifier`	[Windows computer name]
`WildcardIdentifier`	[]
`DismissIdentifier`	[]

defined in an array, more than one parameter can be set. Hence, performance tests can be defined for several machines. Performance tests are defined as a list in `tests` (line $4-7$). Notice, that the benchmark's name is its full qualified name and each benchmark starts in a separate line and is tab-indented. Assertions are described by closed intervals [lower bound, upper bound].

```
1 ---
2 identifier: WildcardIdentifier
3 parameters: []
4 tests:
5     java.example.benchmarkA: [15, 17]
6     java.example.benchmarkB: [19, 21]
7     java.example.benchmarkC: [14, 16]
```

Listing 3.4. A performance test configuration for RadarGun

3.2.6 The Continuous Integration Environment Jenkins

The main goal of this master's thesis is to support the quality management in a continuous integrate environment. Continuous integration is a development practice [17] helping software engineers to develop software. An overview of the process of CI is illustrated in Figure 3.2. Developers add their source code to a source control system, e.g. GIT. The commit triggers an automatic build process. While building the software, functional testing starts automatically, to verify the correctness of a method and its result. If a test fails, the whole build process stops. After the build and testing process, developers receive a result, whether the build has failed or finished successfully. If the build has failed, the developer receives an E-Mail containing the test results. In 2004 Sun Microsystems published Hudson [46], an extensible continuous integration server. Jenkins is a fork of the CI tool Hudson and an open-source platform [53]. The advantage of Jenkins over other CI systems, e.g. Bamboo, is that Jenkins is community driven under the Creative Commons Attribution

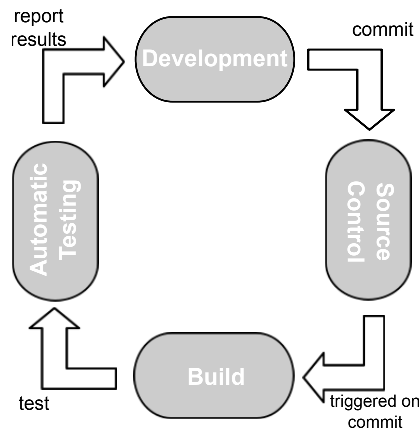


Figure 3.2. Workflow in CI

Share-Alike license and everyone can easily add custom plugins. It is built upon a plugin architecture and allows the installation of custom plugins. Thus, each CI environment customizable and configured as one desires. Due to its architecture, Jenkins is usable by developers that write products in different programming languages than Java, e.g., C++, C, Ruby etc.. The community of Jenkins is the largest in the context of CI environments and continuously enhances Jenkins [53]. Daily new features, bugs fixes, and plugin updates are released. In Chapter 5 we describe how we utilize Jenkins for performance testing.

3.2.7 The Javascript Plotting Framework CanvasPlot

The plotting framework CanvasPlot [29] is build upon the plotting framework D3 [10], which is a JavaScript library for producing dynamic, interactive data visualizations in web browsers and makes use of the SVG, HTML5, and CSS standards. Using CanvasPlot, one can choose between scatter, time series, vector time series, and group plots. The key advantage why we use this framework is the feature that allows one to zoom-in and zoom-out. This way, the visualized data can be expanded or limited in the web-browser as desired, without changing any parameters. Additionally, data can be inserted or removed dynamically. In Figure 3.3 a time series plot with two data sets is shown.

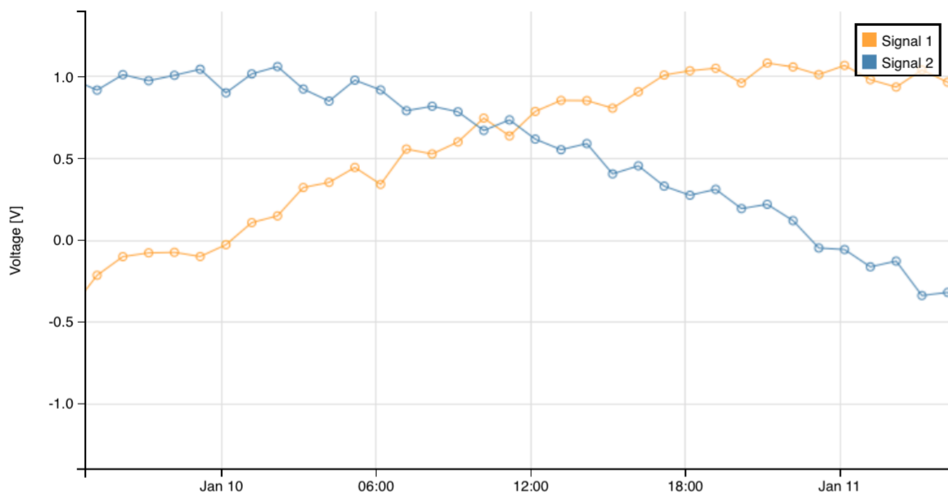


Figure 3.3. Example of a time series plot using CanvasPlot (source: [29])

Chapter 4

Enhancing the Performance Testing Framework RadarGun

While exploring RadarGun's prototype [20], we experienced some issues that impair the usability of this performance testing tool. First of all, performance tests are reported only after all benchmarks were executed, instead of performing the benchmark, testing it against the assertions, and reporting each performance test one after another. Hence, we do not receive the reports in real time and thus there is no proper progress monitoring. Secondly, the export writer is a built-in parser using the data format CSV and exports solely the fields that are hard coded into the parser. We are unable to report results different from the average runtime. Furthermore, the results of performance tests can not be properly reconstruct from the exported data, due to the missing informations regarding the run mode and measured time unit. However, when visualizing the results the run mode and measured time unit are important to interpret a performance test's result correctly (see Section 3.1.3). Thirdly, the configuration of performance tests is not completely uncoupled from the benchmark's configuration that is done in the Java classes. Since this separation is intended by Henning, Wulf, and Hasselbring [21], the configuration mechanism needs some improvements, e.g., to indicate the time unit and run mode a performance test is defined for. Finally, one single unexpected event during a benchmark run may corrupts the whole test result, since a statistically analysis is missing. Thus, it does not take care of outliers that influence the final results too much.

In the following, we enhance RadarGun by improving these four aspects. To enhance RadarGun, we follow the design principle separation of concerns. By abstracting the different tasks in different parts of RadarGun, we achieve a refined modularization.

4.1 Improving the Pipe-And-Filter Architecture

RadarGun's entire architecture is build upon a P&F architecture utilizing the generic P&F framework TeeTime (see Section 3.2.3). In Figure 3.1 the architecture of RadarGun's prototype is shown. Since the `Benchmark Runner` stage blocks the process until all benchmarks are performed, this restrains RadarGun's potential to be applied in the development process by software engineers. If and only if all benchmarks finished, the data is analyzed and reported in the following stages. Hence, a proper progress monitoring is missing.

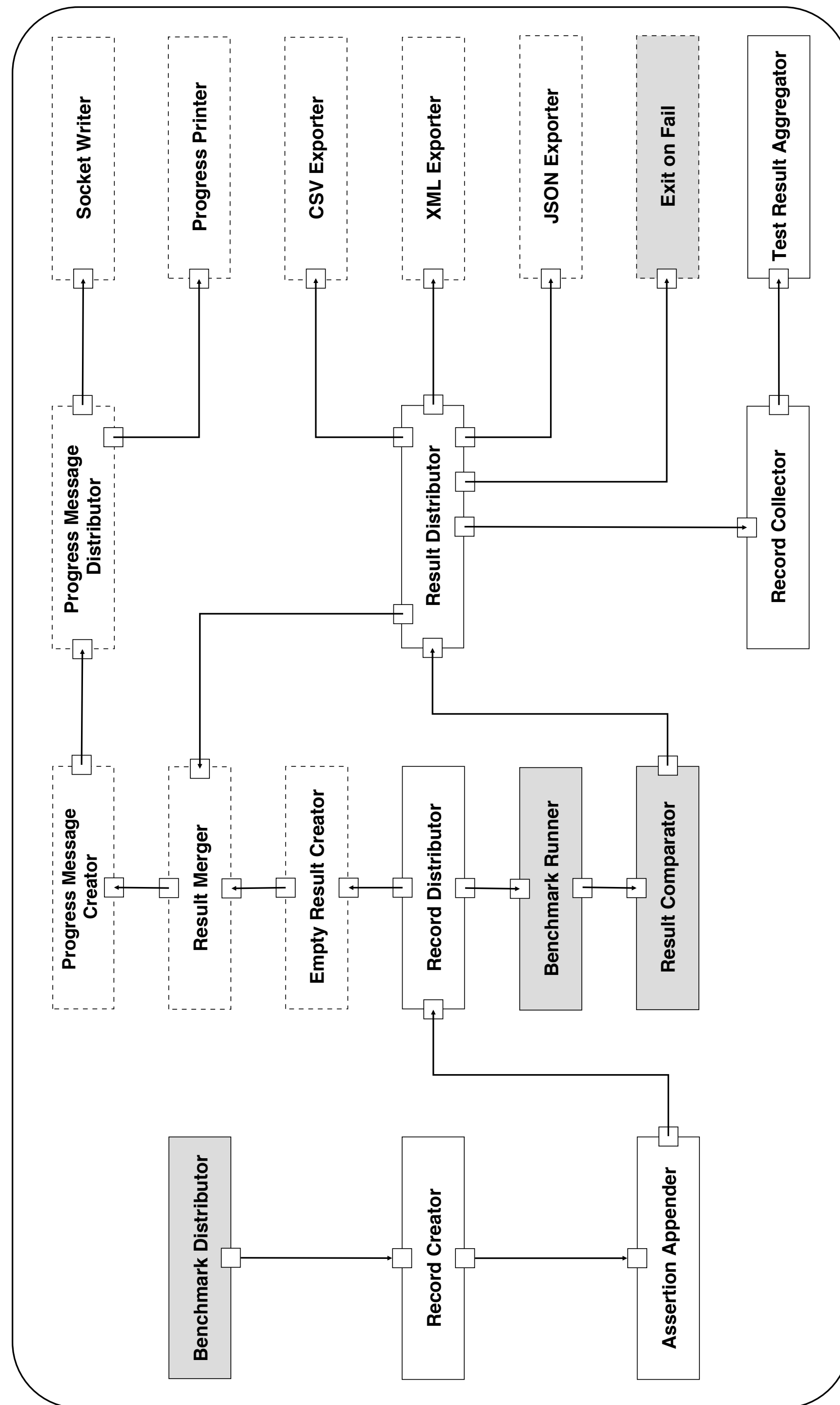


Figure 4.1. RadarGun's improved P&F architecture

In Figure 4.1 we illustrate how we split up RadarGun's P&F configuration. The stages with dashed borders are optional and can be included to the execution configuration via parameters on start up. All other stages are essential for the proper passing of benchmarks and performance test results. RadarGun's initial P&F configuration builds an execution pipeline consisting of four filters: (a) retrieve the list of benchmarks provided by JMH, (b) execute all benchmarks, (c) compare the results with assertions afterwards, (d) and report all results at once subsequently. Consequently, these filters do a lot of different computations at once. These filters are still part of our architecture and are visualized in gray. Nevertheless, the idea and potential of P&F is to abstract the architecture of programs to produce a high modularity [56]. When splitting the `Benchmark Runner` stage, we abstract the steps of having a list of benchmarks and executing them all, before reporting the results. Thus, we create the `Benchmark Distributor` stage. This stage contains the list of all benchmarks found by JMH. Instead of passing the whole list at once, we pass each entry one by one. To separate a performance test's configuration from the benchmark's configuration, we enrich a benchmark with its corresponding assertion before the benchmark is performed. Therefor, we utilize a container object named `Record` that temporarily holds all the data that is passed from stage to stage. Following, the `Benchmark Distributor` passes each benchmark from the list of benchmarks to the `Record Creator` stage one by one. For each passed benchmark this stage creates a `Record` object containing its name. Afterwards, the `Record` is passed to the stage `Assertion Appender`. If an `Assertion` exists for the passed benchmark, the `Assertion` is appended to the `Record`. If no `Assertion` exists for a benchmark, this benchmark is still be executed. However, the benchmark can not be compared against an `Assertion` and thus, the performance test fails in the end. `Assertions` are looked up in the configuration file based on the benchmarks' names. If the performance test's configuration differs from the benchmark's configuration, the `Benchmark Runner` stage that executes the passed benchmark, reconfigures the benchmark options to the run mode and timeunit passed in the assertion. When finished, the results are saved as `RunResult` in the `Record`. Since JMH computes some statistics for each result, we can utilize these computed statistics to obtain the confidence interval for the confidence level predefined in the assertions (see Section 3.2.2). From these results the `Result Comparator` stage creates the final `TestResult` object, which includes whether the test finished successfully or failed. All test results are collected in the `Record Collector` stage. After all performance tests finished, the `Record Collector` stage forwards the collected `TestResults` to the stage `Test Result Aggregator`. Here all performance test results are aggregated to one single object that is exported as XML file and represents the entire session of executed performance tests. Notice that there is no stage to start a statistically analysis of the performance test run, since JMH already contains all the methods to calculate confidence intervals from the results. This statistics are part of the `RunResult` object contained by the `Record`. The confidence level is important in the `Test Result Comparator` stage. A performance test is successful, if and only if the confidence interval is within the given assertion's lower and upper bound. Each performance test result is then exported in the data format that is

specified in the parameters by the user.

In summary, we obtain a P&F configuration as shown in Figure 4.1. However, we did not mention all the stages we include in the splitting process of stages, e.g., the stages `Record Distributor` and `Result Distributor`. To implement a proper progress monitoring, we include the stage `Record Distributor` to pass the benchmarks, which are about to be executed to all stages that are in charge to report the progress. When passing a `Record`, the monitoring stage reports which performance test is about to be executed. For example, when a benchmark is about to be executed, the stage `Progress Message Creator` creates and passes an object named `Progress Message` with the parameter `Started` and an object of the type `EmptyTestResult`. This object solely contains the name of the benchmark that is about to be executed and is only created to report that a benchmark's execution started. We describe the progress monitoring in more detail in Section 4.4. After all stages that are to the `Record Distributor` stage reported the progress, the `Record` is passed to `Benchmark Runner` stage which executes the benchmark. To distribute the results, we include the `Result Distributor` stage, whose only job is to receive a `Record` containing a `TestResult` and distribute it to all stages that are connected to one of the distributor's output ports. For example, all stages that export data are connected to the stage `Result Distributor`.

Additionally, we integrate a Transmission Control Protocol (TCP) stage named `Socket Write` to export the results via a TCP connection. This stage represents a server sending data to connected clients in a non-blocking way. If the parameter `-tcp-output` is set, RadarGun waits for an incoming connection by a TCP client. This stage communicates via `Progress Message` objects. This `Progress Message` is sent (1) right after a client has connected, (2) when a benchmark is chosen to be executed, (3) when the performance test finishes, (4) and finally when all performance tests are finished. The first `Progress Message` after a TCP client has connected, contains a list with all benchmark that are about to be executed. Thereby, the connected client receives a progress status and can produce a custom progress monitoring. This stages are implemented, especially, to establish a data transfer link between our Eclipse plugin and RadarGun (see Section 6.3).

4.2 Separating Performance Test Configurations from Benchmark Configurations

In Section 3.1.3 we emphasize the difference between performance tests and benchmarks. Benchmarks are a part of performance tests. However, without assertions to be compared against, benchmarks do not represent a complete performance test. To isolate the benchmark execution done by JMH from the testing done by RadarGun, we separate the configuration of performance tests from the benchmarks' configuration written in Java classes. Thus, the whole configuration of performance tests is assembled in a configuration file imported by RadarGun and not inside the written benchmarks. Thereby, developers do not need to recompile the benchmarks. Nevertheless, they can still configure a different run mode or

measured time unit. Although, the idea of separating performance test configurations from benchmark configurations is mentioned in Henning, Wulf, and Hasselbring [21], it is not well-implemented. Solely the assertions can be defined, but neither the run mode, nor the time unit, nor the confidence level.

In Listing 3.4 an example performance test configuration for RadarGun's prototype is illustrated. We define configurations in a text file in the data format YAML [6]. On start up, configurations are imported via the parameter `-cp-assertions`, which takes the classpath to the configuration file. Notice that this parameter also handles a list of comma separated entries and thus we are able to import several configuration files. We modify the performance test configuration by adding two additional fields to the array. As shown in Listing 4.1,

```
1 ---
2 identifier: WildcardIdentifier
3 parameters: []
4 tests:
5    teetime.benchmark.Port2PortBenchmark: [35, 45, 'ns/op',  95]
6    teetime.benchmark.Port2PortWithTermInstanceofCheckBenchmark: [18, 26,
         'ns/op', 95]
7    teetime.benchmark.Port2PortWithTermReferenceCheckBenchmark: [15, 20,
         'ns/op', 95]
```

Listing 4.1. A RadarGun configuration including the run mode and confidence level.

the new array represents `[lower bound, upper bound, timeunit/per operation (or vice versa), confidence level]`. Hence, we instantly see for each benchmark which assertion is used, which run mode is set, and which time unit is measured. Furthermore, one does not need to recompile the benchmarks, when adjusting the run mode or measured time unit. For example, let us assume a benchmark is configured to measure the number of operations per nanoseconds (`ops/ns`), yet the performance test is configured to measure `ns/op`. As a consequence, the `Benchmark Runner` stage (see Figure 4.1) changes the mode and time unit of the corresponding benchmark to the required mode and time unit defined in the performance test's configuration. This happens autonomously, before a benchmark is about to be executed.

4.3 Creating an Import/Export Model for Performance Test Results

RadarGun's prototype (see Section 3.2.5) allows merely the export of a test's name, its score, the predefined assertion, and whether the test was successful or has failed. Although, the

configuration reader uses an open-source YAML parser, named SnakeYAML[1], to import the performance test configurations, the export writer is custom-tailored and restricted to export only a few fields in the data format CSV. To allow more customization and improve the usability, we universalize the import/export strategy and include Jackson [15], a JSON processor framework for Java, for all exports and imports. Internally Jackson processes all data as JSON objects, yet it can export and import the data in up to 9 other data formats, e.g., CSV, XML, YAML, ect.. In order to use Jackson, we add annotations, provided by Jackson, to the Plain-Old-Java-Objects (POJOs) we aim to export. The framework serializes and deserializes objects by references to the annotated fields and names. Built-in mappers deserialize the objects from files or serializes the data to write it to files. This way we provide an interface to export the performance tests in different data formats. Thereby, software engineers, which are using our tool, are able to import the performance test result in the standard data format they require for further processing. In the following, we demonstrate how we create our model of `TestResults` to make it exportable with Jackson. In Figure 4.2 we give an overview of the `TestResult` creation model used by RadarGun. The interface `TestResult` contains all the methods that are accessible for devel-

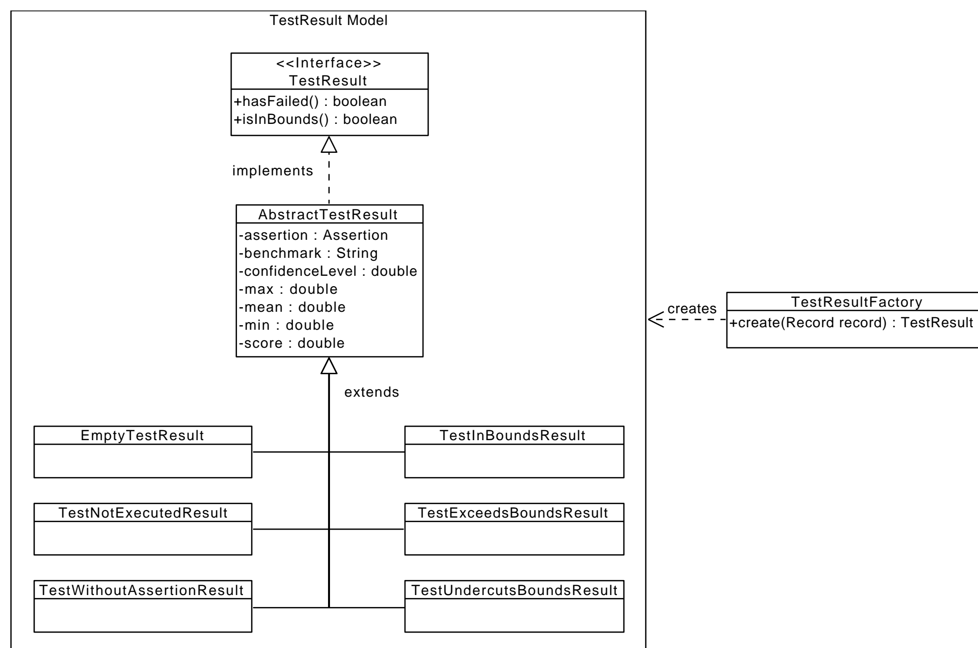


Figure 4.2. The model of RadarGun for the creation of TestResult objects.

opers. Notice that we do not display the getter and setter methods used for the attributes

[1] `www.snakeyaml.org/`

shown in the `AbstractTestResult` class. The abstract class `AbstractTestResult` implements the interface and is generalized by the classes `EmptyTestResult`, `TestExceedsBoundsResult`, `TestInBoundsResult`, `TestUndercutsBounds`, `TestNotExecutedResult`, and `TestWithoutBounds`. All classes that inherit from `AbstractTestResult` implement the methods `isInBounds()` and `hasFailed()`. However, the only `TestResult` object that returns `True` when calling the method `isInBounds()` is the class `TestInBoundsResult`. All `TestResults`, except `EmptyTestResult`, are created in the stage `Result Comparator` using the factory pattern implemented in `TestResultFactory`. This factory creates these objects depending on the data a `Record` contains. An overview of which object is created in which case is given in Table 4.1. In summary, we categorize the test results in three categories:
(1) `Failed`, if the test result exceeds or undercuts the bounds;
(2) `No Result`, if a benchmark was not performed or contains an empty result;
(3) `Successful`, if the test result is within the bounds.
`EmptyTestResult` objects are solely created by the stage `Empty Result Creator`. This objects represent a dummy test result for progress monitoring and solely contain a performance test's name. This object is required to report the benchmark that is about to be executed. When objects are serialized to persist performance test results in files, for example XML files,

Table 4.1. Overview of which concrete TestResult object is produced by the factory.

Produced TestResult	*Case*
TestNotExecutedResult	The score of a test is `NaN`.
TestWithoutAssertionResult	No assertion is appended to the `Record`.
TestExceedsBoundsResult	The confidence interval exceeds the upper bounds of the assertion.
TestUncercutsBoundsResult	The confidence interval undercuts the lower bounds of the assertion.
TestInBoundsResult	The test performed successfully.

the output looks like in Listing 4.2. The header (Line 1) represents the object `TestResult` and contains its concrete implementation saved in the field `type` (see Table 4.1). Notice that the prototype of RadarGun [20] exports assertions, too. However, we add annotations to the POJO `Assertion` to export the run mode and measured timeunit as `timeunit` and the confidence level as `confidenceLevel`. Hence, the fields are serialized like in Lines $2-7$. Furthermore, we add the fields `confidenceInterval` (Lines $9-12$), `max` (Line 13), `mean` (Line 14), and `min` (Line 15) to the serialization model. Although the fields `benchmark` (Line 8), `score` (Line 16), `wasSuccesful` (Line 17), and `hasFailed` (Line 18) are already exported by RadarGun's prototype. However, we annotate these fields for export. Notice that the field `confidenceInterval` is an array of the type `double[]` and thus each entry is named `confidenceInterval`, too.

```
1  <TestResult type="TestInBoundsResult">
2  <assertion>
3     <lowerBound>15.0</lowerBound>
4     <upperBound>17.0</upperBound>
5     <timeunit>ns/op</timeunit>
6     <confidenceLevel>0.95</confidenceLevel>
7  </assertion>
8  <benchmark>teetime.benchmark.Port2PortBenchmark.queue</benchmark>
9  <confidenceInterval>
10    <confidenceInterval>16.250478935663516</confidenceInterval>
11    <confidenceInterval>16.424796507032468</confidenceInterval>
12 </confidenceInterval>
13 <max>16.62222454799239</max>
14 <mean>16.337637721347992</mean>
15 <min>16.035154771665955</min>
16 <score>16.337637721347992</score>
17 <wasSuccessful>true</wasSuccessful>
18 <hasFailed>false</hasFailed>
19 </TestResult>
```

Listing 4.2. Serialization of the POJO TestInBoundsResult

By including RadarGun into any framework, this framework can access the model to deserialize data from files exported by RadarGun. In Listing 4.3 we use the interface POJO TestResult to define the fields to export with Jackson. Only a few annotations are required to configure the whole output. In the following, we describe which annotation in Listing 4.3, Listing 4.4, and Listing 4.5 generate which output in Listing 4.2. By using @JsonRootName("TestResult") (Line 1 in Listing 4.3), we define the root element. Since TestResult is an interface and thus, can not be instantiated, a concrete type from Table 4.1 is passed as subtype. These subtypes are passed as the value type to the root element. The value of type corresponds to the class name of the concrete type and is configured via the annotation JsonTypeInfo(...) (Line 2). Each subtype is declared in the list of subtypes. This list is created by @JsonSubTypes(...) and each subtype is defined by @JsonSubTypes(...) (Lines 3 – 9). @JsonPropertyOrder({...}) sets the order in which the elements are exported in. Since Jackson automatically uses getter and setter methods for the annotated variables, we do not need to annotate or even define these getters and setters, expect for the data we aim to export without declaring and annotating a concrete field. In Lines 19 – 24 two methods are annotated with @JsonGetter(...). By using this annotation, the fields <hasFailed>...</hasFailed> and <isInBounds>...</isInBounds> are exported, e.g., Lines 17 – 18 in Listing 4.2, although there are not any fields failed or succesful declared. Thus, @JsonGetter(...) allows us to export the methods hasFailed and isInBounds without

saving this values to a field. However, the classes that implement the interface `TestResult` have to implement this methods.

```
1  @JsonRootName("TestResult")
2  @JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include =
       JsonTypeInfo.As.PROPERTY, property = "type")
3  @JsonSubTypes({@JsonSubTypes.Type(value = TestExceedsBoundsResult.class,
       name = "TestExceedsBoundsResult"),
4  @JsonSubTypes.Type(value = TestInBoundsResult.class, name =
       "TestInBoundsResult"),
5  @JsonSubTypes.Type(value = TestNotExecutedResult.class, name =
       "TestNotExecutedResult"),
6  @JsonSubTypes.Type(value = TestUndercutsBoundsResult.class, name =
       "TestUndercutsBoundsResult"),
7  @JsonSubTypes.Type(value = TestWithoutAssertionResult.class, name =
       "TestWithoutAssertionResult"),
8  @JsonSubTypes.Type(value = EmptyTestResult.class, name =
       "EmptyTestResult") })
9  @JsonPropertyOrder({ "type", "assertion", "benchmark",
       "confidenceInterval", "max", "mean", "min", "score", "wasSuccessful"
       })
10 public interface TestResult {
11
12     public Assertion getAssertion();
13     public String getBenchmark();
14     public double[] getConfidenceInterval();
15     public double getMean();
16     public double getMin();
17     public double getMax();
18     public double getScore();
19
20     @JsonGetter("hasFailed")
21     public boolean hasFailed();
22
23     @JsonGetter("wasSuccessful")
24     public boolean wasSuccesfull();
25 }
```

Listing 4.3. Jackson annotations in the POJO TestResult.

Except for the fields `hasFailed` and `isInBounds` all exported fields are declared in the abstract class `AbstractTestResult`. In Listing 4.4 the POJO with the annotations to export fields

is shown. Each exportable field is annotated by `@JsonProperty(...)`. This property contains the name the field has on serialization. The classes that generalize the `AbstractTestResult` and are declared as subtypes in `TestResult`, are annotated by `@JsonTypeName(...)`, e.g., Line 1 in Listing 4.5. Objects deserializable on import by annotating its class' constructor with `@JsonCreator` (Line 8). This constructor receives and binds the fields, declared by `@JsonProperty(...)`, to variables (Line 9) which is usable by the framework, later. Additionally, these classes implement the methods `hasFailed` (Lines 14 – 16) and `isInBounds` (Lines 19 – 21) which are annotated with `@JsonGetter` in Listing 4.3.

```
1  public abstract class AbstractTestResult implements TestResult {
2     @JsonProperty("assertion")
3     private final Assertion assertion;
4
5     @JsonProperty("benchmark")
6     private final String benchmark;
7
8     @JsonProperty("confidenceInterval")
9     private final double[] confidenceInterval;
10
11    @JsonProperty("max")
12    private final double max;
13
14    @JsonProperty("mean")
15    private final double mean;
16
17    @JsonProperty("min")
18    private final double min;
19
20    @JsonProperty("score")
21    private final double score;
22    [...]
23 }
```

Listing 4.4. Jackson annotations in the POJO AbstractTestResult.

All in all, we created a model to export and import the performance tests done by RadarGun. Supplementary to the model of `TestResults`, all classes contained in the package `de.cau.se.radarun.shared`, e.g. `Assertion` and `ProgressMessage`, are serializable and deserializable by Jackson. Thus, we provide an interface to process performance test results by external tools. Furthermore, we provide an interface to import the data in different data formats, since Jackson supports up to 9 data formats, e.g., CSV, JSON, XML, YAML

etc.. However, RadarGun still exports the performance tests only in the data formats XML, JSON, and CSV.

```
1  @JsonTypeName(value = "TestInBoundsResult")
2  public class TestInBoundsResult extends AbstractTestResult {
3
4     public TestInBoundsResult(Record record) {
5        super(record);
6     }
7
8     @JsonCreator
9     public TestInBoundsResult(@JsonProperty("assertion") final Assertion
          assertion, @JsonProperty("benchmark") final String benchmark,
          @JsonProperty("max") final double max, @JsonProperty("mean") final
          double mean, @JsonProperty("min") final double min,
          @JsonProperty("confidenceInterval") final double[] confidenceInterval,
          @JsonProperty("score") final double score) {
10       super(assertion, benchmark, max, mean, min, confidenceInterval, score);
11    }
12
13    @Override
14    public boolean hasFailed() {
15       return false;
16    }
17
18    @Override
19    public boolean isInBounds() {
20       return true;
21    }
22 }
```

Listing 4.5. Jackson annotations in the POJO TestInBoundsResult.

4.4 Supporting Progress Monitoring

Due to the improved architecture, RadarGun supports different types of progress monitoring. External tools, e.g. Jenkins and Eclipse, can utilize the results exported by RadarGun for progress monitoring. Furthermore, RadarGun produces three kinds of progress monitoring on its own, accessible in a console. In the following, we describe this progress monitoring provided by RadarGun. The prototyp of RadarGun uses the stage `Result Printer` to print all performance test results to a console at once, after all performance tests finished. For each performance test there are three possible outcomes:

(1) `Failed`, (2) `No Result`, (3) or `Successful`.

The improved architecture enables RadarGun to report performance tests one by one and thus, a progress monitoring in real time. One type is generated by TeeTime that logs, as shown in Listing 4.6, the stages it creates (Lines $1-2$) and the connected outgoing and incoming ports (Line 4) on start up. When the producer stage `Benchmark Distributor` starts to pass each benchmark one by one, a start up message is logged (Line 6). This progress is already monitored in the prototype of RadarGun [20].

```
1 13:28:55.871 [main] DEBUG
      radargun.lib.teetime.stage.basic.distributor.Distributor:Distributor-1
      - numOpenedInputPorts (inc): 1
2 13:28:55.873 [main] DEBUG
      radargun.benchmarks.centralindex.RecordCreatorStage:RecordCreatorStage-0
      - numOpenedInputPorts (inc): 1
3 [...]
4 13:28:55.960 [main] DEBUG
      radargun.lib.teetime.framework.scheduling.pushpullmodel.A3PipeInstantiation
      - Connected (unsynch)
      radargun.lib.teetime.framework.OutputPort@197d671 and
      radargun.lib.teetime.framework.InputPort@402e37bc
5 [...]
6 13:28:55.968 [Thread for BenchmarkDistributor-0] DEBUG
      radargun.benchmarks.BenchmarkDistributor - Executing runnable stage...
```

Listing 4.6. Progress monitoring by TeeTime on start up.

The progress monitoring we add, reflects the status of the performance tests. An example log for three performance tests is shown in Listing 4.7. Each log starts with the current time the log was created and which is then followed by the status and further informations. On start up, RadarGun reports the status `[START UP]` followed by the number of benchmarks that were found by JMH (Line 1). Before a benchmark is executed, RadarGun logs the status `[STARTING]` followed by the name of the benchmark that is about be performed (Line 2). After this benchmark finished, the status `[FINISHED]` followed by the performance

test's result is logged (Line 3). This result contains the score, the computed confidence interval for the predefined confidence level, and the predefined assertion. Performance tests can still finish with one of three possible results `Failed` (Line 3), `No Result` (Line 5), or `Successful` (Line 7). When all performance tests finished "`[SHUTDOWN] Finished all performance tests`" (Line 8) is logged and RadarGun stops.

```
1 14:18:08 [START UP] Found 3 benchmarks.
2 14:18:08 [STARTING] teetime.benchmark.Port2PortBenchmark.queue is running
      now
3 14:18:30 [FINISHED] teetime.benchmark.Port2PortBenchmark.queue [FAILED]
      Score: 17.473846460811227 CL: 0.95 CI: [17.260543850863304,
      17.68714907075915] (Bounds: [15.0, 17.0])
4 14:18:30 [STARTING]
      teetime.benchmark.Port2PortWithTermInstanceofCheckBenchmark.queue is
      running now
5 14:18:51 [FINISHED]
      teetime.benchmark.Port2PortWithTermInstanceofCheckBenchmark.queue [NO
      RESULT] Score: NaN CL: 0.95 CI: [NaN, NaN] (Bounds: [19.0, 21.0])
6 14:18:51 [STARTING]
      teetime.benchmark.Port2PortWithTermReferenceCheckBenchmark.queue is
      running now
7 14:19:11 [FINISHED]
      teetime.benchmark.Port2PortWithTermReferenceCheckBenchmark.queue
      [SUCCESSFULL]  Score: 15.654463695356986 CL: 0.95  CI:
      [15.540353056806708, 15.768574333907264] (Bounds: [14.0, 16.0])
8 14:19:11 [SHUTDOWN] Finished all performance tests
```

Listing 4.7. Progress monitoring in the console by RadarGun.

To generate this progress monitoring, three stages (see Figure 4.1) are necessary. Before executing a benchmark, the stage `Empty Message Creator` passes a `Record` that contains an `EmptyTestResult` object. This `Record` is processed by the `Progress Message Creator` stage, which creates a `Progress Message` object that is reported by the stage `Progress Printer`, afterwards. The `Progress Printer` receives the number of performance tests to print it on start up. When this stage terminates it logs the shutdown status.

The third type of progress monitoring is generated by JMH, if the parameter `-jmh-output` is set on start up. JMH then produces a log as shown in Listing 4.8. It logs the benchmark and its configuration that is about to be started (Lines $1-8$), the currently executed fork (Line 11), the warmup iterations (Lines $12-14$), all single benchmark executions (Lines $15-18$), and the results (Lines $20-23$). When the benchmark finishes, its overall results are logged in a compact overview (Lines $27-29$). Notice that JMH by default always prints a confidence interval of $99,9\%$ (Line 23).

In summary, we allow developers to track the execution of benchmarks and performance tests in detail. The output by RadarGun and by JMH can be turned off and on via parameters on start up, independent from each other.

```
1  [...]
2  # Warmup: 5 iterations, 1 s each
3  # Measurement: 15 iterations, 1 s each
4  # Timeout: 10 min per iteration
5  # Threads: 2 threads (1 group; 1x "add", 1x "remove" in each group), will
       synchronize iterations
6  # Benchmark mode: Average time, time/op
7  # Benchmark: teetime.benchmark.Port2PortBenchmark.queue
8  # Parameters: (capacity = 1024)
9
10 # Run progress: 0,00% complete, ETA 00:00:20
11 # Fork: 1 of 1
12 # Warmup Iteration   1: 16,959 ns/op
13 # Warmup Iteration   2: 17,254 ns/op
14 [...]
15 Iteration   1: 16,427 ns/op
16       add:    16,405 ns/op
17       remove: 16,449 ns/op
18 [...]
19
20 Result "teetime.benchmark.Port2PortBenchmark.queue":
21 16,338 ±(99.9%) 0,168 ns/op [Average]
22 (min, avg, max) = (16,035, 16,338, 16,622), stdev = 0,157
23 CI (99.9%): [16,169, 16,506] (assumes normal distribution)
24 [...]
25 # Run complete. Total time: 00:00:20
26
27 Benchmark                        (capacity)  Mode  Cnt   Score   Error  Units
28 Port2PortBenchmark.queue               1024  avgt   15  16,338 ± 0,168 ns/op
29 [...]
```

Listing 4.8. Progress monitoring in the console by JMH.

Chapter 5

Reporting Performance Tests in Jenkins

Introducing the performance testing framework RadarGun [21] to Jenkins [33] and thus, reporting performance tests in a CI environment is the main goal of this master's thesis (see Chapter 2). Hence, we develop a plugin to make RadarGun part of the build process and to visualize the performance tests in Jenkins. Jenkins already provides a plot plugin, named *Plot Plugin* [13] that is able to plot the data produced by RadarGun. This plugin was used by Henning, Wulf, and Hasselbring [21] to visualize the output of RadarGun's prototype in their evaluation. However, this plugin is unable to dynamically react to inputs. To visualize the results a plot has to be configured for each performance test separately. Hence, if we miss to add a performance test's plot to the configuration, it is not visualized. Additionally, plots in *Plot Plugin* are configured as post build step in a job's build configuration. Thus, the data format to visualize is predefined for all builds. For example, if a performance test's measured time unit or run mode has changed between two builds, then this plugin can not distinguish between these two units. Thus, it is unable to visualize the data correctly and draws both performance tests in the same plot. Furthermore, the number of visualized results is predefined in the configuration. One can decide whether all or only a limited number of builds is shown. However, it is not possible to limit the viewed results dynamically.

When reporting performance tests, we expect an overview of all performance tests and their results. Since RadarGun only executes performance tests and exports the results, we develop a plugin that reports and visualizes the data in Jenkins. This plugin creates a build history containing all performance test results. The plots are generated dynamically and are not a part of the job configuration. Additionally, we can zoom in and out to adjust the number of the viewed results in each plot, as described in Section 3.2.7. In the following, we describe how performance tests are reported, gathered, and visualized in Jenkins. Therefor, we create a post build step that imports the data reported by RadarGun.

5.1 Understanding the Stapling of Pages in Jenkins

Jenkins is written in the programming language Java and thus, is a set of Java classes, which model the concepts of a build system [52]. Some classes exactly model the purpose they are named after, e.g., `Job` and `Run`. A `Job` is a particular task in our build process, such as compiling the source code and running unit tests [53]. A `Run` object represents the

current build that is executed. Additionally, there are classes and interfaces that model different build process steps, such as performing a build, e.g., `Maven` to perform a Maven build. The root object is `Jenkins`. All other model objects are subordinates to the root. To extend Jenkins' model objects, extension points are provided.

To use Jenkins, users access Jenkins via a browser. Therefor, Java objects are bound to URLs by the HTTP request handling engine *Stapler* [36]. The root object `Jenkins` is instantiated as a singleton object and is bound to the context root URL, e.g., `"/"`. All other object are bound according to their reachability from this root object and define an URL subspace. Thus, the framework "staples" all children to the parent object, by creating an URL hierarchy. To recursively determine how to process any given URL, the framework uses reflection [35].

Let us demonstrate how *Stapler* processes URLs by means of the following example URL:

```
http://domain.de/jenkins/job/exampleJob/7/radargun/packagename/testname/
\_______________________/\_____________/\/\_______/\__________/\_______/
             |                     |        |     |            |          |
      Jenkins root                Job      Run  BuildAction Package      Test
```

Stapler assign for the corresponding object hierarchy an URL subspace as shown in Figure 5.1. Jenkins' root element is bound to the URL `http://domain.de/jenkins/`. When

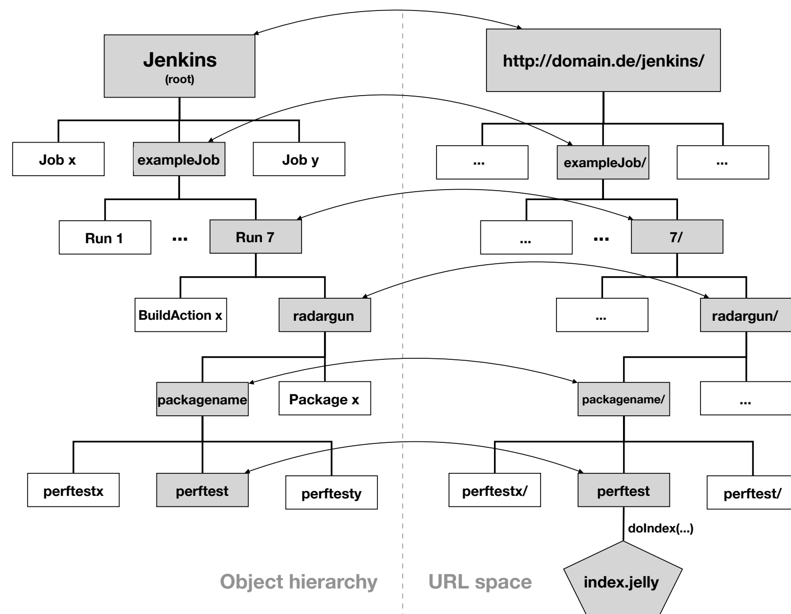


Figure 5.1. Stapler assigns URL subspaces with regards to the object hierarchy.

processing the path segment `/job/exampleJob/` there are different ways how *Stapler* could process this URL segment:

- A method `getJob(String job)` is defined on the `Jenkins` object. Then *Stapler* passes the parameter `exampleJob` to this method. The object returned, has a method named `doIndex(...)`. This method gets called and renders the response.
- A method `getDynamic(String urlspace)` is defined on the `Jenkins` object. Then *Stapler* passes the parameter `/job/exampleJob/` to this method. Jenkins looks up in its list of objects, whether the URL subspace of any object equals the URL space `/job/exampleJob/`. The returned object has a method named `doIndex(...)`. This method gets called and renders the response.

We use the latter approach in our plugin, to process URLs and render the returned object. Nevertheless, there are further possibilities how this URL could be processed, depending on the implementation of methods on objects. However, the URL path segment `/7/radargun/packagename/testname/` is still left. If a job with the name `exampleJob` exists, *Stapler* uses the `getDynamic(...)` method to obtain the `Run` object with the build number 7. If RadarGun is configured as post build step for this job, the `getDymanic(...)` method is called to look up a `Build` object with the path segment `/radargun/`. If this objects exists, the `getDymanic(...)` method is executed to look up a package bound to the URL space `packagename/`. If this package exists and it contains a `PerformanceTestResult` object bound to the name `testname`, than its `doIndex(...)` method is called and the object gets rendered. This URL can be processed recursively by defining the mathematical function $evaluate(node, url)$ [32], too. The evaluation process looks like in the following:

evaluate(<root object>, "/job/exampleJob/7/radargun/packagename/testname")

→ evaluate(<root object>.getJob("exampleJob"), "/7/radargun/packagename/testname")

→ evaluate(<exampleJob object>, "/7/radargun/packagename/testname")

→ evaluate(<exampleJob object>.getDynamic(7), "/radargun/packagename/testname")

→ evaluate(<Run number 7>, "/radargun/packagename/testname")

→ evaluate(<Run number 7>.getDynamic("radargun"), "/packagename/testname")

→ evaluate(<RadarGun build action>, "/packagename/testname")

→ evaluate(<RadarGun build action>.getDynamic(packagename"), "/testname")

→ evaluate(<packagename object>, "/testname")

→ evaluate(<packagename object>.getDynamic("testname"), "")

→ evaluate(<testname object>, "")

→ <testname object>.doIndex(...)

5.2 Understanding the Rendering of Objects in Jenkins

Jenkins uses a simple UI pattern. On the left hand side of the page, see Box #1 in Figure 5.2, a sidepanel that contains menus with several navigable links and command links, helps users to navigate through Jenkins [34]. The content of this links is then shown in the main panel (see Box #2). To contribute to the UI and bind objects to URLs via *Stapler*, we implement

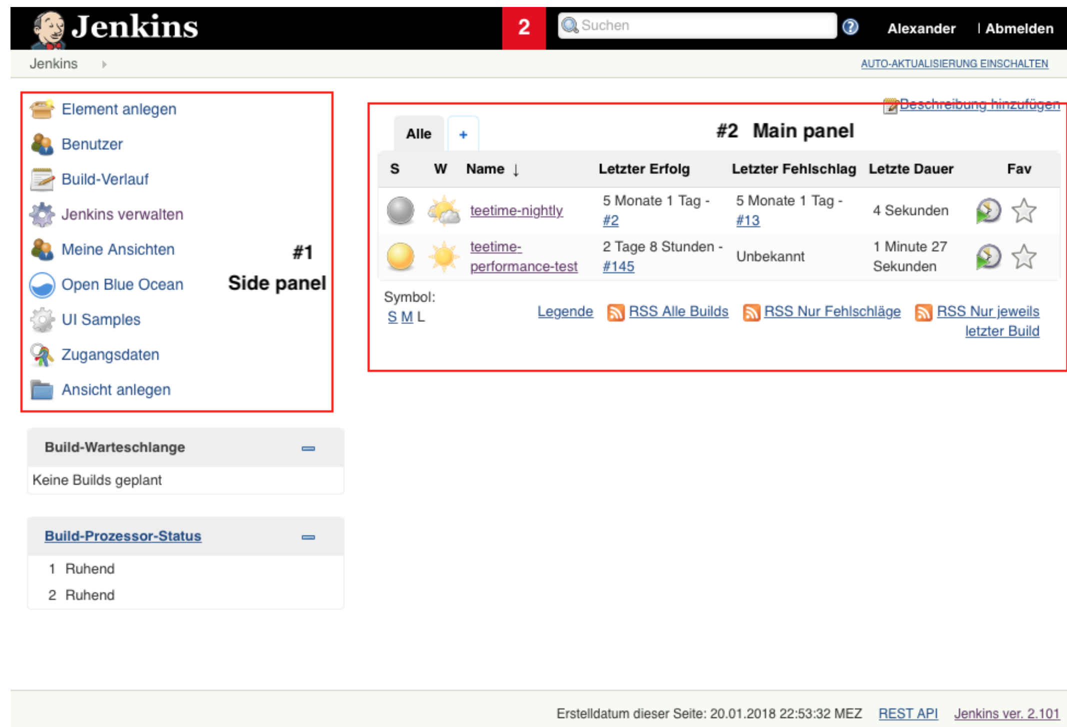


Figure 5.2. Jenkins' UI front page

the interface `Action` in our model objects. Implementing this interface in a model object, creates an additional URL subspace to the parent's model object. `Actions` define at least three methods, which are required to render objects: (1) an icon, (2) the display name of the object, (3) and the URL subspace the object is bound to. The menu shown in Box #1 changes depending on the model object that is currently rendered. For a `Job` a different menu is displayed than for a single `Run`. As described in Section 5.1, the method `doIndex(...)` is returned to render the object. *Stapler* is able to render the template formats *Jelly* [2] and *Groovy*[1]. In our plugin we use *Jelly* to write templates. *Jelly* is a Java and XML based scripting and processing engine [2], which accesses Java methods and renders HTML output. Tem-

[1] `http://groovy-lang.org`

plates are placed in the source folder `src/main/resources/packagestructure/objectname/`. By default the `doIndex(...)` method renders the file `index.jelly`. A Jelly file is an XML documents that gets parsed into a script. The advantage of using Jelly scripts over Groovy script is that by using *Jelly* XML elements can be bound to Java code. Hence, we implement a dynamic processing of our data, yet use a familiar syntax. Jelly files are tied directly to classes and hence, we are able call methods on those classes. By including the XML Schema `jelly:stapler` (Line 2 in Listing 5.1) we are able to write and include views. Such a view we include to our rendering, is the current build's menu (Line 4). The view that renders this menu is written in the `sidepanel.jelly` file. Since the content depends on the action that is rendered, the view is bound to an `Action` object. Similar to Java's keyword `this`, which references the current object, the `it` keyword in Jelly files references the object it is tied to. To call Java code, we use the dollar sign and curly-braces, e.g., `${it.run}`, which calls the method *getRun()* in the tied Java class.

```
1   <?jelly escape-by-default='true'?>
2   <j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler"
        xmlns:l="/lib/layout" xmlns:g="glide">
3   <l:layout>
4   <st:include it="${it.run}" page="sidepanel.jelly" />
5   <l:main-panel>
6   <h1>RadarGun Performance Tests</h1>
7   <h2>${%benchmark(it.displayName, it.run.number)}</h2>
8   <table>
9   <tr>
10  <td>${%status}</td>
11  <td>${it.wasSuccessful()}</td>
12  </tr>
13  <tr>
14  <td>${%units}</td>
15  <td>${it.testResult.assertion.timeunit}</td>
16  </tr>
17  [...]
18  </table>
19  </l:main-panel>
20  </l:layout>
21  </j:jelly>
```

Listing 5.1. A Jelly script to render the results of a performance test.

The main content is written between the tags `<l:main-panel>...</l:main-panel>` (Line 5 and Line 19). To layout templates, we use HTML standard elements. We add the ti-

tle (Line 6) and subtitle (Line 7) at the top of the content panel. The performance test results are displayed in a table (Lines 8 – 18). Each row consists of two columns. The first column contains the label, the second contains the corresponding performance test result data field. We annotate labels by `${%xyz}`, which is a placeholder to allow an internationalization depending on the language configured in Jenkins. Therefor, we create the files `index.properties` and `index_de.properties`. The first property file contains the default values for the labels. If `index_de.properties` exists and Jenkins' configured language is German, the labels are replaced by the values defined in that file. Additionally, it is possible to define a label and pass one or more values as parameter(s). Thereby, the value can be placed anywhere in the label's output. In Line 7 we pass a benchmark name and the build number to the label `$%benchmark(...)`. The output is rendered as '`Performance Test: BenchmarkName for Build #Number`'. The second column contains the data retrieved from the tied Java class. We can call a getter method, such as `$it.isInBounds()` in Line 11 or use `$it.testResult.assertion.timeunit` (Line 15), which is equal to `this.getTestResult().getAssertion().getTimeunit()`).

How this Jelly script is rendered for a performance test is shown in Figure 5.6. Since we only report performance tests, all our renderings consist of the included sidepanel, a title and subtitle in the header, and a table to display the data. The renderings that plot the data additionally include the D3 based Javascript library *CanvasPlot* [29] to visualize the data.

5.3 Providing a Build Pipeline Step

Jenkins' object model is extensible and thus, provides different extension points that are accessible via interfaces. `Job` objects model a project to combine and connect different steps in a build. Together they form a build pipeline, e.g., → pull the project source code from Git → compile it with Maven → run unit tests → report the results afterwards. Loosely speaking, build steps can be categorized in build and post build steps. Since Jenkins provides a build step interface that is able to execute shell scripts, we use this shell script interface to execute RadarGun and to run performance tests during the build process. Hence, we do not develop a custom build step to add the execution of RadarGun to the build pipeline. However, the performance test results exported by RadarGun are still not imported in Jenkins. Hence, we define a post build step that is executed every time a build finishes. By utilizing the extension point `Publisher`, we create a post build step to process a task after a build completes. A `Recorder` is a kind of `Publisher`, yet additionally collects statistics from the build. Furthermore, by extending an object of the type *Recorder*, we are able to mark builds as was successful, has failed, or contains no results. This ensures that builds are marked according to their results, before notifications are sent to developers via `Notifiers`. Due to the newly established pipeline configuration provided by the Blue Ocean plugin and UI, we wrap the execution of our `Publisher` object into a `Step` object that starts the post build step on execution.

Our `Recorder` extension is named `RadarGunPublisher` (Line 1). Each `RadarGunPublisher`

is bound to a `Run` object that represents the current build (Line 3). The constructor (Line 6) is annotated with `@DataBoundConstructor` (Line 5). This annotation signalizes that this constructor takes the parameters one may defines for the post build step. Since we do not provide any configurable parameters, the constructor is empty. When performing this post build step, the method `perform(...)` (Line 9 – 23) is called. This method takes, among other parameters, the current run and the workspace a `Job` belongs to. Thus, we read the data that was exported by RadarGun during the previous build step. This data is imported as an `Action` object, named `RadarGunBuildAction` (Line 11), and added to the current build (Line 12). The object `RadarGunBuildAction` is important to report the performance test result. Since, this object implements the interface `Action`, it allocates an URL subspace to report a single build. We discuss this in detail in Section 5.5. Depending on the number of missing results and failed tests the build is either marked as `FAILURE` (Line 16), `UNSTABLE` (Line 18), or `SUCCESS` (Line 20). The `RadarGunPublisher` is used as post build step. However, it is still not addable to the build pipeline. To be selectable, Jenkins provides extension points for `Descriptor` objects. Thus `RadarGunPublisher` contains an inner class `DescriptorImpl` that extends `BuildStepDescriptor<Publisher>` (Line 32). The `@Extension` annotation lets Jenkins know that this is a plugin extension of the Jenkins core. The `DescriptorImpl` class represents a configurable object for the `RadarGunPublisher`.

```
1  public class RadarGunPublisher extends Recorder implements
       SimpleBuildStep {
2
3     private Run<?, ?> run;
4
5     @DataBoundConstructor
6     public RadarGunPublisher() { }
7
8     @Override
9     public void perform(Run<?, ?> run, FilePath workspace, Launcher
           launcher, TaskListener listener) throws InterruptedException,
           IOException {
10       this.run = run;
11       RadarGunBuildAction buildAction = new RadarGunBuildAction(run,
             workspace);
12       run.addAction(buildAction);
13
14       if (buildAction != null) {
15          if(buildAction.getNumberOfErrorTests() > 0) {
16             run.setResult(Result.FAILURE);
```

```
17           } else if(buildAction.getNumberOfFailedTests() > 0) {
18              run.setResult(Result.UNSTABLE);
19           } else {
20              run.setResult(Result.SUCCESS);
21           }
22        }
23     }
24
25     @Override
26     public DescriptorImpl getDescriptor() {
27        return (DescriptorImpl) super.getDescriptor();
28     }
29
30     [...]
31     @Extension
32     public static final class DescriptorImpl extends
           BuildStepDescriptor<Publisher> {
33
34        public DescriptorImpl() {
35           load();
36        }
37
38        public boolean isApplicable(Class<? extends AbstractProject>
              aClass) {
39           return true;
40        }
41
42        public String getDisplayName() {
43           return "Report Performance Test Results (RadarGun)";
44        }
45
46        @Override
47        public boolean configure(StaplerRequest req, JSONObject formData)
              throws FormException {
48           save();
49           return super.configure(req, formData);
50        }
51     }
52  }
```

Listing 5.2. Create a post build step to import performance test results by utilizing the extension point Publisher.

When `RadarGunPublisher` calls its `getDescriptor()` method, a `DescriptorImpl` object is returned. This object defines the displayed name (Lines 13 – 15) and persists configured parameters (Lines 17 – 21). As mentioned previously, we do not configure anything for this post build step.

In summary, by utilizing the `Publisher` extension point, we are able record performance tests done by RadarGun as post build step. Thereby, we provide an option to the configuration of `Jobs`. This option is either selectable via the button `Add Post Build Action`, shown in Figure 5.3, or by including it in the pipeline configuration as described in Section 5.4. When adding our plugin as post build step, new links are added to the UI of the jobs.

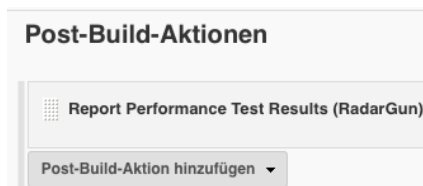


Figure 5.3. Add RadarGun as post build step to a job.

Thereby, our plugin then reports single builds, as described in Section 5.5. Additionally, it presents a history of all builds, as described in Section 5.6.

5.4 Configuring a Build Pipeline

There are several ways to configure build pipelines for jobs in Jenkins. A newly established way is to configure pipelines using the integrated Blue Ocean UI. This plugin uses text files to configure build pipelines for jobs. We either can write a configuration file or use the UI to connect several steps to a build pipeline. When saving the configuration in the UI, a configuration file is created automatically in the job's repository. In the following, we demonstrate how to configure a project to utilize RadarGun to execute and report performance tests. Since the configuration via the UI creates a configuration file and vice versa, we describe the connection of build steps by reference to Figure 5.4. The parameters we set for each build step are described by reference to Listing 5.3.

First of all, we create a Git multipipeline job in Jenkins, which links to a Git repository containing our performance test and pulls the performance tests from the repository when starting the build. In Figure 5.4 this step is represented by `start`. After the repository has been pulled, the project has to be built in order to retrieve all performance test. This build step is a shell script step that starts a `Maven` build. It is important to compile the performance tests with JMH before starting RadarGun, to receive the latest compiled benchmarks and performance test configuration. The step `Run RadarGun` (Lines 9 – 13) is a shell script step. As shown in Listing 5.3, this step executes Java with several parameters (Line 11). First we declare the classpath to RadarGun, which is located in our plugin's `lib`

folder, and afterwards the classpath to the performance tests. Secondly, we declare the main class to execute. The parameter `-cp-asertions` declares the path to the performance test configuration files. Notice that we do not declare an export path to the test result, since Jenkins uses the default export path `/target/radargun-reports/` defined by RadarGun.

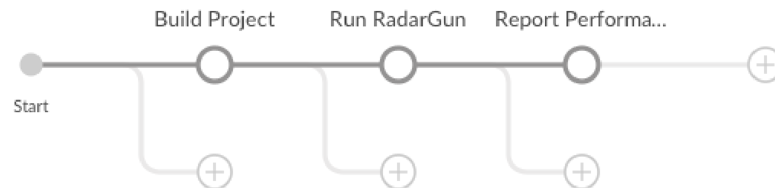


Figure 5.4. Jenkins Pipeline Configuration in the BlueOcean UI.

```
1  pipeline {
2     agent any
3     stages {
4        stage('Build Project') {
5           steps {
6              sh 'mvn clean package -DskipTests'
7           }
8        }
9        stage('Run RadarGun') {
10          steps {
11             sh 'java -Djmh.ignoreLock=true -cp
                   ${JENKINS\_HOME}/plugins/radargun/WEB-INF/lib/radargun-
                    2.0.0-SNAPSHOT.jar:${WORKSPACE}/target/benchmarks.jar
                   radargun.RadarGun --cp-assertions
                   assertions/se-jenkins.yaml'
12          }
13       }
14       stage('Report Performance Tests') {
15          steps {
16             radargunreporting()
17          }
18       }
19    }
20 }
```

Listing 5.3. A configuration file that includes RadarGun in the build process.

The last step is labeled as `Report Performance Tests` (Line 14) and includes the post build

step named `radargunreporting` (Line 16). This step executes the post build step we defined in Section 5.3. This step imports the performance test created by RadarGun and creates the model objects to persist and render the performance test results in Jenkins' UI.

5.5 Reporting a Single Build

In Section 5.4 we create a build pipeline to build a job named `teetime-benchmark`. Its build pipeline contains a build step to start the performance testing using a shell script and a post build step to import the results by the `RadarGunPublisher`. After the performance tests finished, the post build step imports the file from `FullTestResults.xml`, which is exported to RadarGun's default output path. Each post build step creates a `RadarGunBuildAction` object that represents the collection of test results in a build and binds this object to the run. Each imported performance test result creates a `PerformanceTestResult` object that utilizes the extension point `Action` to create an URL subspace for each performance test result. Performance test results from the same package are merged to a `PackageResult` object, which implements the `Action` interface and creates an URL subspace, too.

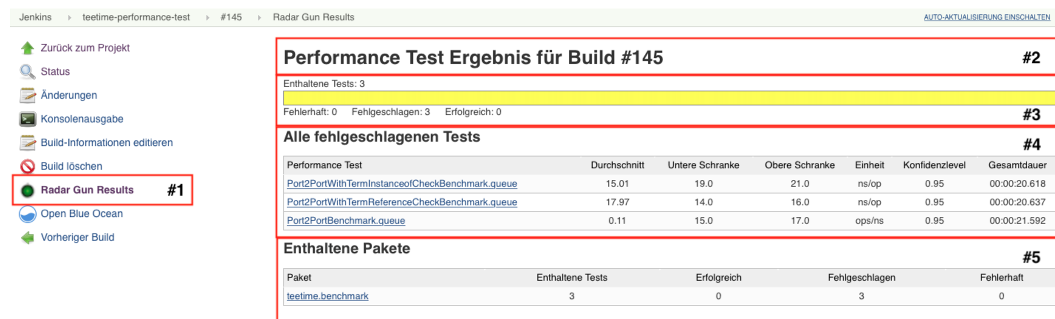


Figure 5.5. Overview page for all performance tests in a single build

Each build that includes our post build step, provides a link to the `RadarGunBuildAction` object (Box #1 in Figure 5.5), which represents an overview page of all performance test results. This URL subspace is rendered by a Jelly script, as described in Section 5.2. Box #2 shows the page's title including the build number. A result bar at the page's head (Box #3) visualizes how many tests were executed and how many finished successful, have failed, or contained no results. Performance tests that finished successfully, are visualized in green. Failed performance tests are visualized in yellow. Performance tests containing no results are visualized in red. Performance tests that had failed or contained no results, are presented at the top and reference to the results' details (Box #4). They are not categorized by its package names. A package overview is shown in Box #5. This overview shows how many tests are in the different packages and what their results are. Package results are represented similar to the items on the overview page. Packages are not displayed in a hierarchical,

but in a flattened way. When clicking on a package, only those performance tests are listed which contain to that specific package. By clicking on a performance tests name, we

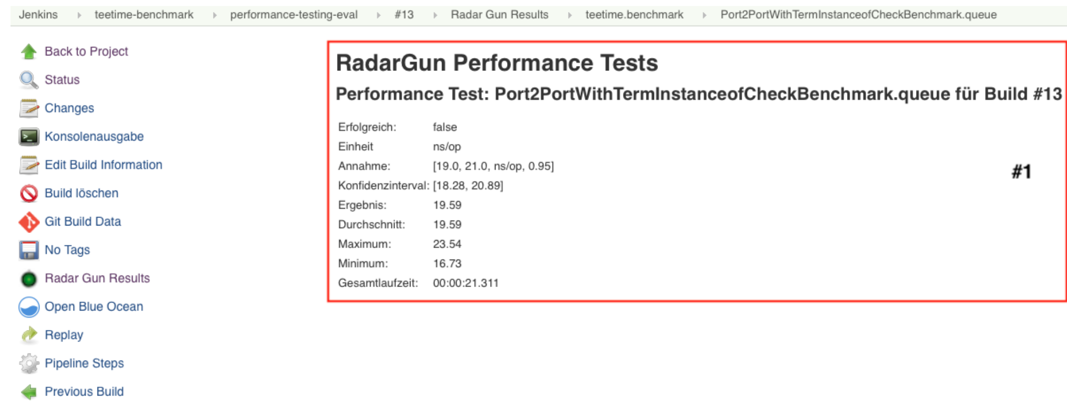


Figure 5.6. Details of a performance tests

navigate to the results for this performance test in detail. In Figure 5.6 we illustrate a single result output for the performance test `Port2PortWithTermInstanceofCheckBenchmark.queue` in Build #13. Box #1 presents the results in a table as show in the Jelly script in Listing 5.1.

5.6 Reporting a Build History

Each build is a model object and attached to a job. When including RadarGun as post build step, each run contains the recorded performance tests. We iterate over all builds to create a history of all performance tests for all builds. As shown in Figure 5.7, this history can be navigated through the menu on the sidepanel (Box #1). A build history is created for a job (Box #4), whereas the overview of single builds can be navigated through the list of all builds in Box #2. The overview page collects and displays all packages in the corresponding project that contains the performance tests (Box #3). Different from the package results of single builds, the package result of our history contains aggregated performance test results. Furthermore, in this package we are able to compare the performance test results in plots, as shown in Figure 5.8. If we click on the checkbox (Box #2), the plot appears in Box #1. Performance tests are aggregated to the same performance test, if and only if the names, the run mode and the measured time unit are equal. If two performance tests equal in name, yet differ in the time unit or run mode, they are aggregated separately (see Box #3). Two Performance tests are only compared with each other, if the units are equal, e.g., *ns/op*. For example, the performance tests `Port2PortWithTermInstanceofCheckBenchmark` in Figure 5.8 represents the aggregated results of the test `Port2PortWithTermInstanceofCheckBenchmark`. By clicking on it, we receive an overview of all builds. This overview is shown in Figure 5.9. The plot is displayed in Box #1. The scores for the specific performance test are visualized

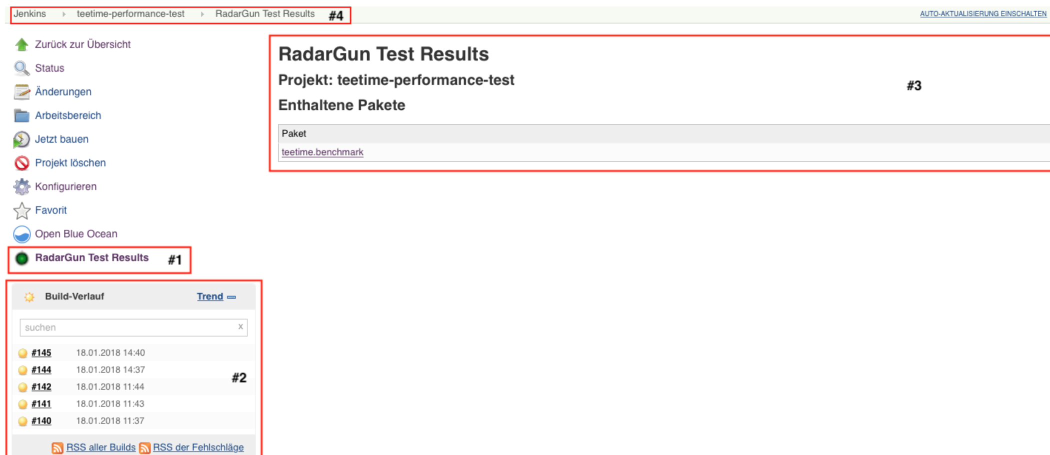


Figure 5.7. Package overview for the build history

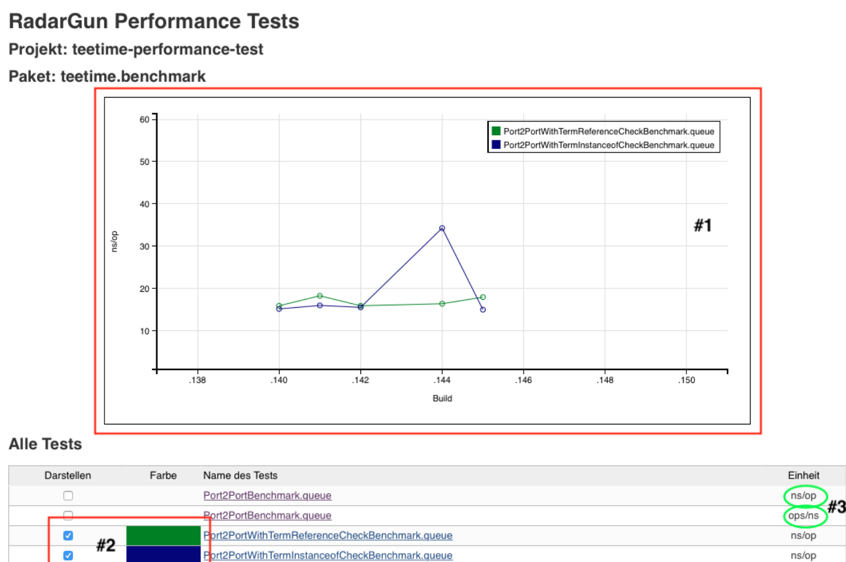
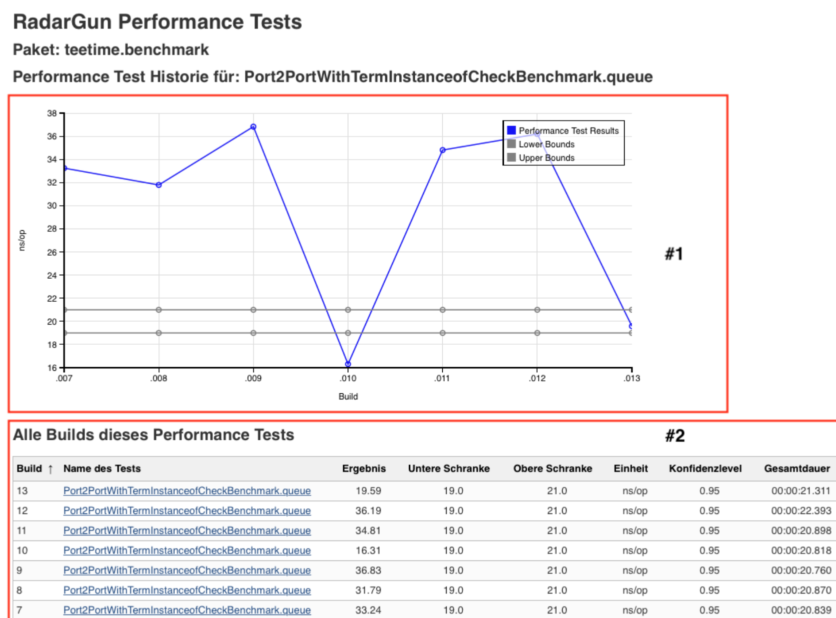


Figure 5.8. Comparing the results of two different performance tests

as blue plot and the assertions' lower and upper bounds are visualized in gray. The list of the performance test results for each build is sorted by the build number in descending order (Box #2). By clicking on the performance test, we visualize the test results for a specific build as shown in Figure 5.6.



Build ↑	Name des Tests	Ergebnis	Untere Schranke	Obere Schranke	Einheit	Konfidenzlevel	Gesamtdauer
13	Port2PortWithTermInstanceofCheckBenchmark.queue	19.59	19.0	21.0	ns/op	0.95	00:00:21.311
12	Port2PortWithTermInstanceofCheckBenchmark.queue	36.19	19.0	21.0	ns/op	0.95	00:00:22.393
11	Port2PortWithTermInstanceofCheckBenchmark.queue	34.81	19.0	21.0	ns/op	0.95	00:00:20.898
10	Port2PortWithTermInstanceofCheckBenchmark.queue	16.31	19.0	21.0	ns/op	0.95	00:00:20.818
9	Port2PortWithTermInstanceofCheckBenchmark.queue	36.83	19.0	21.0	ns/op	0.95	00:00:20.760
8	Port2PortWithTermInstanceofCheckBenchmark.queue	31.79	19.0	21.0	ns/op	0.95	00:00:20.870
7	Port2PortWithTermInstanceofCheckBenchmark.queue	33.24	19.0	21.0	ns/op	0.95	00:00:20.839

Figure 5.9. A performance test's build history presents a list of results for each build.

Chapter 6

Reporting Performance Tests in Eclipse

The IDE Eclipse [16] is one of the most used tools for Java development. We aim to support software engineers during the development process. Therefor, we introduce RadarGun into the Eclipse environment by providing a plugin, which reports performance tests done by RadarGun, in an Eclipse view.

6.1 Understanding the Eclipse Rich-Client-Platform

New Eclipse versions are released by the Eclipse Foundation annually. By releasing Eclipse 4.2, named Juno, the entire platform architecture was restructured. Before Juno, the Eclipse platform was designed to serve as an open tools platform. Since the release of Juno, the major API and architecture break allows users to build custom client application. Solely a minimal set of core plugins is needed to build a rich client application. This set of core plugins is known as the Eclipse Rich Client Platform (RCP). Rich client applications are still based on a dynamic plugin model. Furthermore, the UI is built using the same toolkits and extension points. However, the layout and functionality of the workbench is under control of the plugin developer and thus is customizable [40]. Hence, the Eclipse 4.x API allows to integrate independent software components. For these components most of the data processing occurs on the client side, which is the Eclipse 4.x core.

The IDE Eclipse is reimplemented on top of the Eclipse 4.x API and runs in a legacy mode. This legacy mode is called *compatibility mode*. Plugins developed for Eclipse 3.x still run on Eclipse 4.x platforms, yet in the compatibility mode. As illustrated in Figure 6.1, the Eclipse 3.x API implementation uses the Eclipse 4.x API and provides the Eclipse 3.x Interface that is used by Eclipse 3.x API based plugins. Plugins developed using the Eclipse 3.x API, utilize and define extension points in the configuration file `plugin.xml`. The compatibility layer converts the relevant extension point information into an application model. By including the package `org.eclipse.platform` in a project, the file `LegacyIDE.e4xmi` is included, which defines the initial window and some model Addons of the Eclipse 3.x IDE [38]. In Eclipse 4.x the Eclipse Modeling Framework (EMF) allows to store the model content via the EMF persistence framework that provides an XML Metadata Interchange (XMI) or XML persistence provider. By default EMF uses XMI, which is a standard for exchanging metadata information via XML [39]. Thus, new elements are configured in an `e4xmi` file instead of the `plugin.xml`, which still points to the `e4xmi` file. However, not all

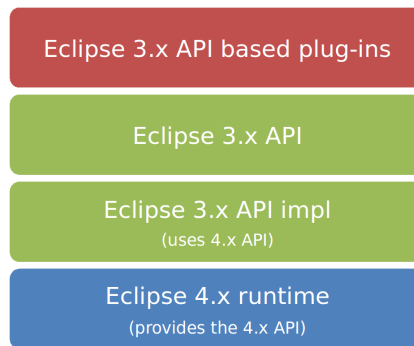


Figure 6.1. The Eclipse 3.x API still works in the Eclipse 4.x runtime via the compatibility layer (source: Lars Vogel [38])

programming concepts of Eclipse 4 work using the compatibility layer. New programming concepts of Eclipse 4, such as dependency injection, merely work for model objects that are declared in an XMI file, e.g., parts, handlers, and commands.

Although the Eclipse platform team plans to support the compatibility layer for an unlimited period of time [38], we implement as many parts as possible using the Eclipse 4.*x* API. The concept of the application model and dependency injection makes our plugin code more concise and flexible. Additionally, we are able to utilize the event service and the extensible Eclipse context hierarchy that provides better ways to communicate within our application. Furthermore, we are more independent of changes in new Eclipse versions. Therefor, we use a `fragment.e4xmi` file to contribute model elements to the legacy model application and try to avoid the Singleton objects of the Eclipse platform, e.g., `Platform` or `PlatformUI`. To design the RadarGun view we use the Standard Widget Toolkit (SWT) [14]. SWT is an open source widget toolkit for Java and is used in Eclipse to provide portable access to the UI facilities of the operating systems on which Eclipse is running [14]. Nevertheless, not all parts can be implemented purely using the Eclipse 4.*x* API. Parts, such as the launch configurations in Section 6.2, can not be implemented without the extension point `org.eclipse.debug.core.launchConfigurationTypes`, which is solely configurable in the `plugin.xml` file. In the following, we mark the parts that are implemented using the Eclipse 3.*x* API.

6.2 Providing a RadarGun Launch Configuration in Eclipse

One of the parts we are unable to implement using the Eclipse 4.*x* API, is the launch configuration to execute RadarGun in Eclipse. We still have to utilize the extension points `org.eclipse.debug.core.launchConfigurationTypes`, `org.eclipse.debug.ui.launchShortcuts`, and `org.eclipse.debug.ui.launchConfigurationTabGroups` in the `plugin.xml` file. These ex-

tension points provide two ways to execute RadarGun. One way is to execute a launch shortcut, which is accessible via *Right click on project → Run As → RadarGun Performance Tests*. This shortcut consists of three different components. In order to always execute the latest benchmarks, a default Maven launch configuration is created for the project one aims to execute. This configuration compiles the benchmarks. The second component is the RadarGun launch configuration that executes RadarGun and thus, executes the performance tests. The third component is a group launch configuration that merges the Maven and the RadarGun launch configurations to one launch configuration that executes the contained elements one after another, in a blocking way. Otherwise, RadarGun could start while the benchmarks still compile. Another way is to simply start the RadarGun launch

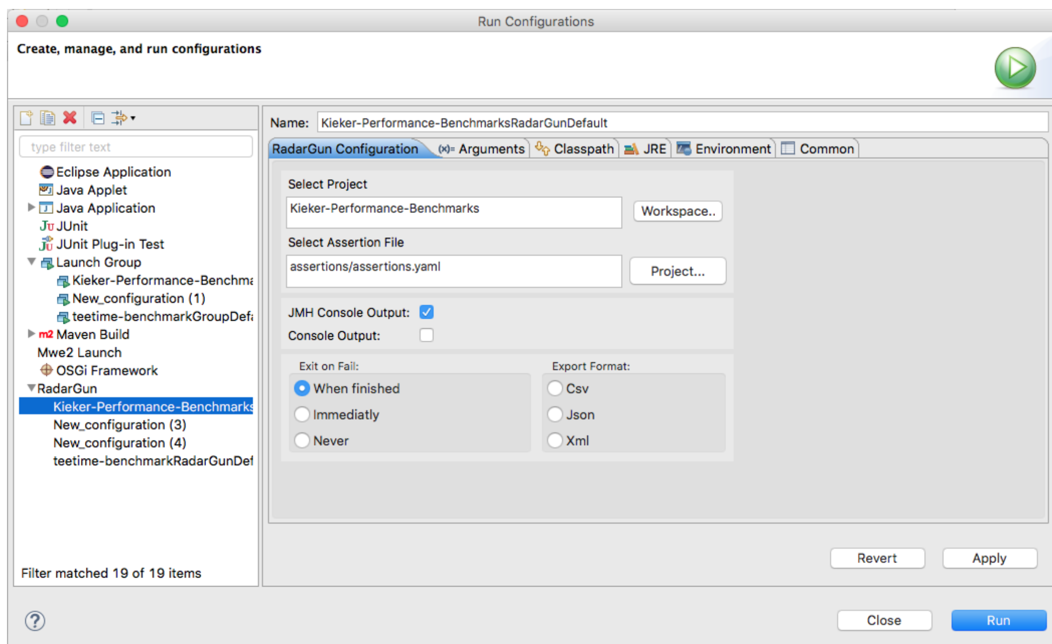


Figure 6.2. RadarGun launch configuration

configuration via the launch configuration menu. In the launch configuration menu we are able to modify the parameters RadarGun is executed with. Such a launch configuration is shown in Figure 6.2. A launch configuration contains the workspace of the project this configuration is created for and the location to the assertions. This locations are mandatory to launch RadarGun. The file panel to select the workspace and the assertion location are elements using the Eclipse 3.x API. Optional parameters can be turned on or off via checkboxes and radio buttons. The parameter `-tcp-output` is set internally by default and can not be turned off by the user. This parameter is mandatory to exchange data between RadarGun and the Eclipse view that visualizes the results. We describe this data exchange

in Section 6.3.

6.3 Reporting Performance Test Results in Eclipse

The reporting of performance tests in Eclipse is the most challenging part of our plugin. RadarGun is indented to be used for automatic testing in a CI environment and hence to export performance test results to files. Applications can import these files and report the data, afterwards. Due to the missing progress monitoring in RadarGun's prototype [20], a progress monitoring in external tools is not possible using the prototype. That is why, we enhance RadarGun in Chapter 4 to report a progress monitoring in real time (see Section 4.4). A trivial approach to exchange data is to use a file listener that triggers an event to read the file every time a new files was created. This listener triggers an event for a newly created file, yet this file mays not contains any data or not the complete output. Consequently, the imported performance test results were flawed. To exchange performance test results properly, we implement a Server/Client pattern and exchange messages via an unilateral connection from RadarGun to Eclipse. Thereby, we receive the results correctly and can visualize them.

To establish a connection between RadarGun and Eclipse different steps have to be handled. Since Eclipse starts RadarGun in a VM and passes the classespaths as parameter, we add a launch listener that triggers a start up event on RadarGun's launch. Therefor, we implement the interface `LaunchListeners` provided by Eclipse. If the listener triggers a start up event, a client, which connects to RadarGun, starts. If RadarGun is started with the parameter `-tcp-output`, a semaphore blocks the performance test execution until a client connects. After a client is connected, the performance tests are executed. Subsequently, each `ProgressMessage` (described in Section 4.4) is passed to the `SocketWriter` stage that sends the data to all connected clients. Clients receive the data and handle them. First, the messages are deserialized to `TestResults`. Afterwards, each `TestResult` is sent to the view, which visualizes the result in the Eclipse GUI.

To communicate between the different parts of our plugin, we utilize the built-in `EventBroker` provided by Eclipse via dependency injection. This broker implements a publish-subscribe pattern and thereby, we implement loosely coupled parts. The used publish-subscribe pattern is illustrated in Figure 6.3. We create a domain `RADARGUN/` that contains three subdomains: (1) LaunchConfig, (2) Client, (3) and Data. Objects can subscribe to event topics with regards to these subdomains and publish or receive events. The data format of events is predefined. Since not all objects can use the dependency injection, we provide an interface named `RadarGunEventHandler`. Thereby, we allow parts, which can not use dependency injection, to publish events for our RadarGun plugin. On termination all parts unsubscribe all topics. The subdomain `LAUNCHCONFIG` has two event topics, namely `STARTED` and `FINISHED`. The launch listener triggers `STARTED` to notify the subscribers that RadarGun launches and triggers `FINISHED`, if RadarGun terminates. The object `PerformanceTestRunSession` subscribes the whole topic `RADARGUN/LAUNCHCONFIG/*`.

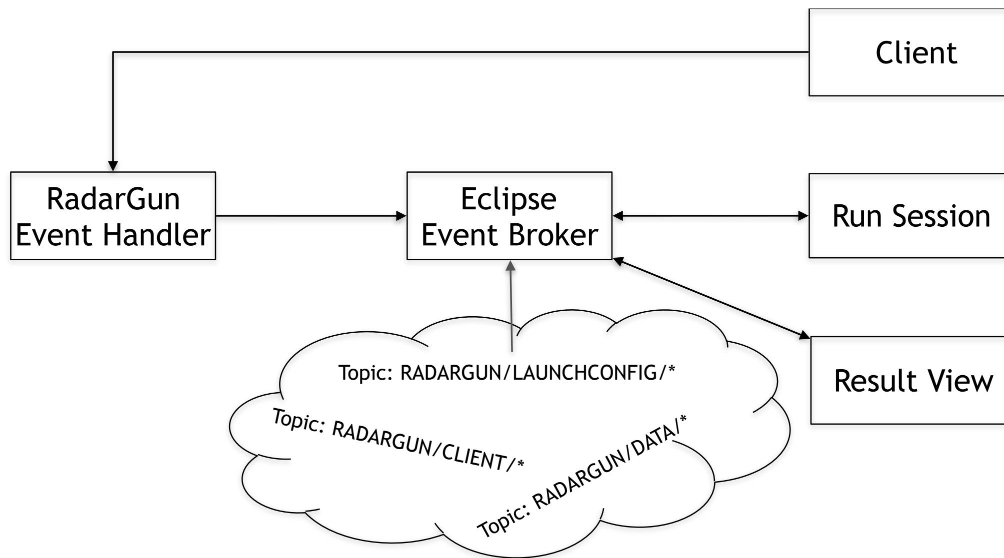


Figure 6.3. Utilizing the event broker to use the publish subscribe pattern.

`PerformanceTestRunSession` creates or terminates a client depending on whether RadarGun starts or terminates. If RadarGun starts, the event `STARTED` triggers. The event `CLOSED` triggers after the client was terminated. To prevent several parallel launches, only one client at a time is running. Additional launches get discarded. A client publishes two events topics: `INIT` and `UPDATE`. Both event topics contain a `ProgressMessage` sent by RadarGun. `INIT` is published by a client, if a `ProgressMessage` is flagged as `Begin`. Then the message represents a list containing all performance tests. An `Update` event triggers, if the progress messages flag is `Started` or `Finished` and the client forwards the `TestResult` object. The view subscribes the topic `RadarGun/DATA/*` and processes the received events. In Section 6.4 we describe how the GUI handles `ProgressMessages`.

6.4 Visualizing Performance Test Results in Eclipse

The RadarGun view in Eclipse opens automatically, if the start up event is triggered. The view in Figure 6.4 is implemented as part using the Eclipse 4.*x* API and thus, utilizes dependency injection to subscribe for the topic `RADARGUN/DATA/*` after construction. When receiving the event `RADARGUN/DATA/INIT`, which contains a list of performance tests, these performance tests' names are added to the table, without the score and assertion. The performance test that is currently executed displays `Running` in the field named `Score`. The event `RADARGUN/DATA/UPDATE` triggers, if a progress message's flag is `Stared` or `Finished`. Then the client forwards the `TestResult` object and updates the GUI's content depending on

whether the `TestResult` is an instance of `EmptyTestResult` or one of the others in Table 4.1. For an `EmptyTestResult` the assertion of the corresponding performance test is updated. Otherwise, the `Score` field is updated from `Running` to the performance test's score.

Running or failed performance tests are marked in yellow. Successfully finished performance tests are marked in green. Performance tests having no results are marked in red. The progress bar is updated dynamically. The differently colored sections visualize the different results. The GUI is built using solely SWT and JFace[1] elements. Thereby, the view accesses the UI facilities of the operating systems on which Eclipse is running. Thus, the view's style harmonizes with the other parts of the Eclipse IDE.

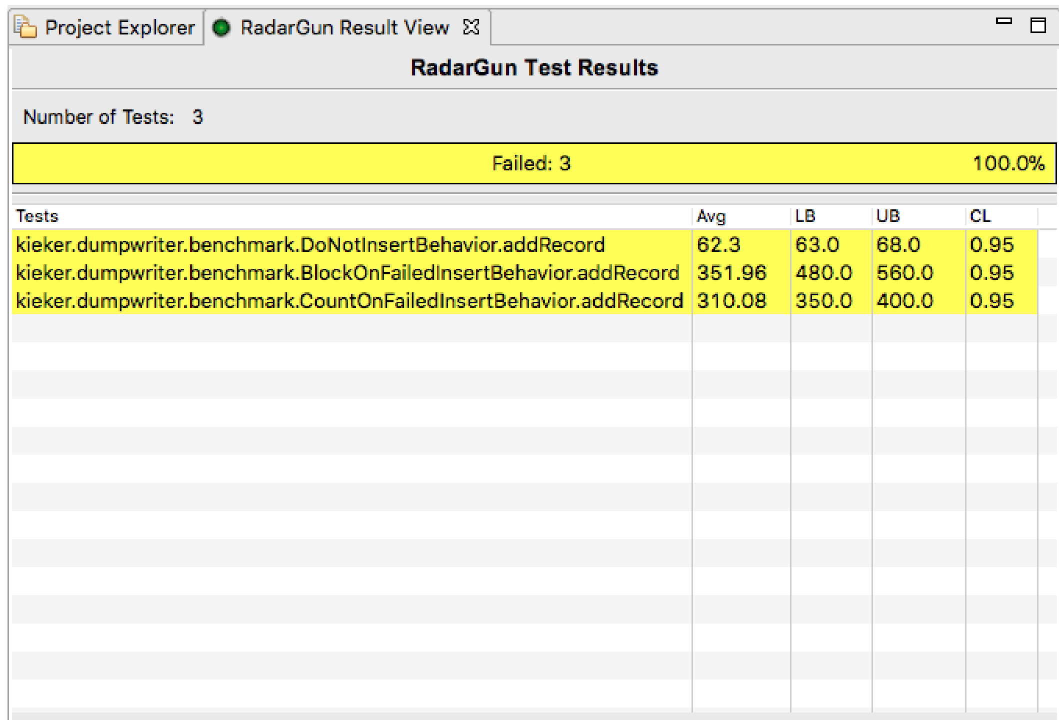


Figure 6.4. RadarGun view visualizing three failed performance tests. Their actual result values are all below the lower bounds.

[1]`http://www.vogella.com/tutorials/EclipseJFace/article.html`

Chapter 7

Application Example of RadarGun

We enhance the performance test framework RadarGun in Chapter 4 and develope two tools to execute performance tests in a CI environment and an IDE. The prototype of RadarGun was evaluated by Henning, Wulf, and Hasselbring [21] with different microbenchmarks for the P&F framework TeeTime. In our feasibility evaluation in Chapter 8, we rerun the tests with TeeTime using our enhanced RadarGun version. However, to evaluate a second framework, that is, to increase the external validity, we write performance tests for the monitoring framework Kieker (see Section 3.2.4) and execute them with RadarGun to accomplish Goal 4 in Chapter 2. Therefor, we introduce how to configure benchmarks using JMH and how to define performance test configurations in RadarGun. Finally, we illustrate how to write a performance test for the Kieker framework.

7.1 Understanding the Benchmark Configuration by JMH

Similar to functional tests in JUnit, performance tests in RadarGun are defined in Java classes. However, the annotations to configure a benchmark are provided by JMH. The methods to test are annotated by `@Benchmark`. Sometimes benchmarks need initialized objects before they are executable. However, if this initialization is not part of the method, it must not be measured and thus should be outsourced from the performance test. Such objects are called "state" objects. Using JMH, these state objects are declared in special state classes, which are annotated with `@State`. To initialize state objects before passing them to benchmarks, we annotate state methods with `@Setup` and `@TearDown`. The @Setup ensures that this method is called to setup the state object, before it is passed to the benchmark method. The *@TearDown* annotation ensures that this method is called to clean up the state object, after the benchmark has been executed. On execution, an instance of that state class is then provided as parameter to the benchmark method. State objects can be reused across multiple calls to benchmark methods. Table 7.1 shows and describes which scopes are provided for state objects by the `Scope` class in JMH. Similar to JUnit, it is also possible to use parameterized tests by declaring the parameters with the `@Param` annotation in a state class.

In addition to the annotations that are used to setup a benchmark, JMH provides annotations to refine the measurements. In Section 3.1.1 we describe different influences that may corrupt the measurements of program executions. To handle these influences, we

Table 7.1. Configurable scopes using JMH

`Scope.`	*Where state object can be reused*
`Thread`	For each thread running the benchmark its own instance of the state object is created.
`Group`	For each each thread group running the benchmark its own instance of the state object is created.
`Benchmark`	All threads running the benchmark share the same state object.

adjust the benchmark execution. The annotation `@Fork` declares the number of separated execution environments. Warmup cycles provide the opportunity to the JVM to optimize the code before the measurement starts.The number of warmup iterations is defined by `@Warmup`. `@Measurement` defines the number of benchmark measurements in each fork. As described in Section 3.1.3, there are different metrics to measure the performance of benchmarks. JMH offers five different benchmark modes, which are selectable using the annotation `@BenchmarkMode`. Table 7.2 shows and describes which constants are provided by the class `Mode` to configure which run mode. The output's time unit can be defined

Table 7.2. Run modes to configure benchmark using JMH

Run Mode	*What is measured*
Throughput	The number of times a benchmark method was executed in a given timeunit.
Average Time	The average time it takes for the benchmark method to execute once.
Sample Time	How long time it takes for the benchmark method to execute.
Single Shot Time	How long time a single benchmark method execution takes to run.
All	Measures all of the above.

via `@OutputTimeUnit`. The class `Timeunit` provides the following constants: (1) `NANOSECONDS`, (2) `MICROSECONDS`, (3) `MILLISECONDS`, (4) `SECONDS`, (5) `MINUTES`, (6) and `HOURS`. JMH is able to measure multi-threaded benchmarks. With the annotation `@Thread`, we declare how many threads are used for the execution of benchmarks. We demonstrate how to write a benchmark in Section 7.3.

7.2 Defining Performance Tests in RadarGun

We present the enhanced performance test configuration for RadarGun in Section 4.2. Since one of our goals is to evaluate our tools by using performance tests in Kieker, see Chapter 2, we demonstrate how to write a new performance test configuration for Kieker. In Chapter 8 we execute this performance tests on an external Jenkins server and on a local machine hosting Jenkins. The external machine uses the `WildcardIdentifier` and thus the parameter field is an empty array `[]`. For the performance tests on our local machine we use `MacAddressIdentifier` as `Identifier` (Line 2) and hence we put our MAC-Address in the parameter array `['00:50:b6:45:f0:e0']` (Line 3). Performance tests are defined in a list written in the data format YAML. All elements in the list `tests` (Line 4) are tab-indented (Line 5 – 7). Each performance test represents the full qualified name of the Java class representing the benchmark. For each declared performance test an array, consisting of exactly four elements, is set. This array is interpretable as `[lower bound, upper bound, timeunit/per operation (or vice versa), confidence level]`. We set the run mode and timeunit to `ns/op` and use a confidence level of 95%. Notice, that we determine the upper and lower bounds by some sample runs of the benchmarks. Therefor, we used our Eclipse plugin in Chapter 6. Since we do not know how fast the `MonitoringController` performs in general, we need these runs to find an first interval for each performance test.

```
1   ---
2   identifier: WildcardIdentifier
3   parameters: ['00:50:b6:45:f0:e0']
4   tests:
5   kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior: [480, 560,
        'ns/op',  95]
6   kieker.dumpwriter.benchmark.CountOnFailedInsertBehavior: [300, 350,
        'ns/op',  95]
7   kieker.dumpwriter.benchmark.DoNotInsertBehavior: [60, 68, 'ns/op',  95]
```

Listing 7.1. A Kieker performance test configuration

7.3 Writing Performance Tests for the Kieker Framework

In the follow, we write a benchmark for the performance test `kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior`. This performance tests is written to test the monitoring controller used by Kieker. A `MonitoringController` has five strategies[1] to write data to a queue. As shown in Figure 7.1, a queue is shared between two threads. The application

[1] https://github.com/kieker-monitoring/kieker/blob/stable/kieker-monitoring/src/kieker/monitoring/core/controller/WriterController.java

thread writes monitoring probes to the queue via the `MonitoringController`. A writer thread reads the entries from the queue. The monitoring controller has five strategies to write the probes to the queue. We use a `DumpWriter` to read the queue. If the queue is blocked, due to the reading of the `DumpWriter`, the `DumpWriter` is too slow. Since the method, which reads a probe and is supposed to write it, does not contain any logic[2] this `DumpWriter` can not block the queue. Consequently, a queue is never full and blocks only, if the operation system, which executes the benchmark, does not schedule both threads fairly.

In our performance test the benchmark, represents the probe and the execution of this benchmark the application thread. Consequently, we measure the minimal overhead that is caused by creating a probe and write it to a queue. The written benchmark is presented in Listing 7.2. Although this strategies could be easily parametrized via `@Param(...)` (Line 12), we have to write three separate benchmarks. Otherwise we receive the score over all three parametrized executions and not three performance tests. JMH executes the parametrization one after another without writing three separate benchmarks to the list of benchmarks used by RadarGun. Thus, we can not write assertions for this parametrization. Consequently, we can not run performance tests for this parametrization. We configure three forks for the benchmark `kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior` (Line 4). Each fork runs five warmup iterations (Line 5) and 30 measurements (Line 6). The benchmark measures the average time per execution in nanoseconds (Line 2). To run a benchmark, the state object `MonitoringController` is configured in the setup (Lines $14 - 21$). JMH executes the benchmark in a single thread (Line 7).

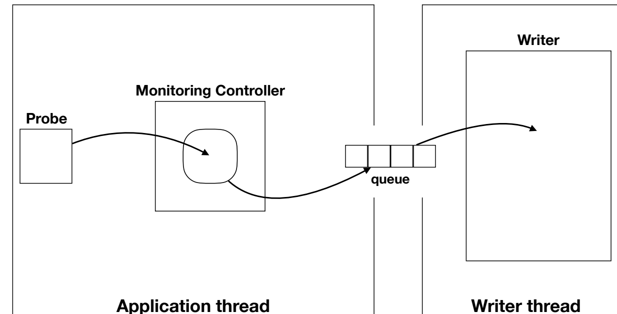


Figure 7.1. We test the minimal overhead of Kieker's monitoring controller.

[2]`https://github.com/kieker-monitoring/kieker/blob/stable/kieker-monitoring/src/kieker/monitoring/writer/dump/DumpWriter.java`

```
1  @State(Scope.Thread)
2  @BenchmarkMode(Mode.AverageTime)
3  @OutputTimeUnit(TimeUnit.NANOSECONDS)
4  @Fork(3)
5  @Warmup(iterations = 5)
6  @Measurement(iterations = 30)
7  @Thread(1)
8  public class BlockOnFailedInsertBehavior {
9
10    private MonitoringController monCtrl;
11    // @Param({"1","2","3","4","5"})
12    private int queueBehavior = 1;
13
14    @Setup
15    public void initConfiguration() {
16       Configuration configuration =
             ConfigurationFactory.createSingletonConfiguration();
17       configuration.setProperty(ConfigurationFactory.WRITER_CLASSNAME,
             "kieker.monitoring.writer.dump.DumpWriter");
18       String key = kieker.monitoring.core.controller.WriterController.
             PREFIX+kieker.monitoring.core.controller.WriterController.
             RECORD_QUEUE_INSERT_BEHAVIOR;
19       configuration.setProperty(key, queueBehavior);  // <-- varying parameter
20       this.monCtrl = MonitoringController.createInstance(configuration);
21    }
22
23    @Benchmark
24    public void benchmark() {
25       EmptyRecord record = new EmptyRecord();
26       monCtrl.newMonitoringRecord(record);
27    }
28 }
29 }
```

Listing 7.2. Benchmark for the performance test `BlockOnFailedInsertBehavior` used by Kieker.

Chapter 8

Feasibility Evaluation

Due to the lack of automatic tests to evaluate the results and the visualization, we evaluate (1) the enhanced performance testing framework RadarGun, (2) the RadarGun plugin for Jenkins, (3) and the RadarGun plugin for Eclipse, in a feasibility evaluation. From Henning, Wulf, and Hasselbring [21] we inherit the scenarios that were used to evaluate RadarGun's prototype. However, these scenarios do not cover all functions we implemented to utilize RadarGun in a continuous integration environment. That is why, we expand the evaluation and evaluate additional features. Our experimental setup is illustrated in Figure 8.1. To increase the validity of our evaluation, we run the evaluation on two

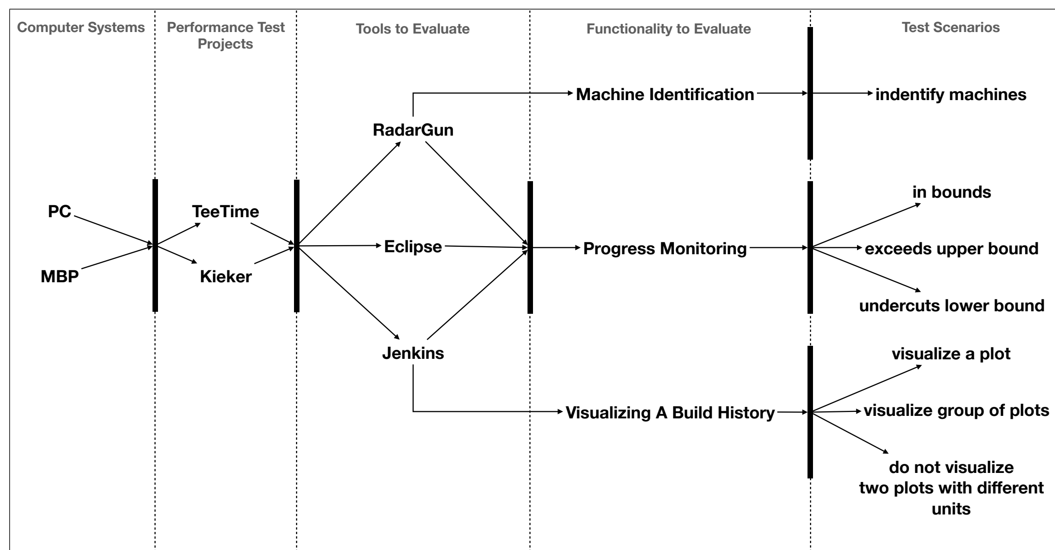


Figure 8.1. Overview of the methodology used for the feasibility evaluation

different environments. The system specifications are shown in Table 8.1. On both systems Eclipse Oxygen is installed. Jenkins runs in a Docker[1] container and uses the latest Jenkins Long-Term-Support[2] (LTS) version, which is version 1.89.3.

[1] https://www.docker.com/what-docker

[2] https://jenkins.io/download/lts/

Table 8.1. The specifications of the computer systems we used in the evaluation.

MacBook Pro 15" (MBP)	*Desktop PC (PC)*
2,4 GHz Intel Core i5	3,2 GHz Intel Core i5
8 GB 1067 MHz DDR3	8 GB 1600 MHz DDR3
Intel HD Graphics 288 MB	NVIDIA GeForce GTX 750 TI 2GB
macOS High Sierra	Windows 10 (Edu)
Eclipse Oxygen (1.a)	Eclipse Oxygen (1.a)
Jenkins LTS (1.89.3)	Jenkins LTS (1.89.3)

Since Henning, Wulf, and Hasselbring [21] use performance tests written for the P&F framework TeeTime to evaluate RadarGun's prototype, we repeat this evaluation on each of our systems for our enhanced RadarGun version. Instead of execution only a single benchmark as Henning, Wulf, and Hasselbring [21] did, we execute all three performance tests provided by TeeTime. To increase the external validity, we additionally conduct this evaluation for the three performance tests we write for the monitoring framework Kieker in Chapter 7. On each system we evaluate the enhanced performance testing framework RadarGun, the Eclipse plugin, and the Jenkins plugin. Both plugins utilize RadarGun to execute and report performance tests.

In this feasibility evaluation we simulate scenarios that cover different cases. These scenarios test the different tools and functions we presented in this master's thesis. To present the results in an oversee-able way, the scenarios are categorized in three categories: (1) Machine identification, (2) Progress Monitoring, (3) and build history visualization in Jenkins. In the following, we describe each category in detail.

8.1 Evaluating the Machine Identification

RadarGun executes hardware-dependent performance tests. To map the assertions to a corresponding benchmark correctly, machine identifiers are used. These machine identifiers are presented in Table 3.1.

8.1.1 Methodology and Test Scenarios

We evaluate whether all machine identifiers identify the correct machine or not. This complements the feasibility evaluation by Henning, Wulf, and Hasselbring [21]. Since this function was never evaluated, we evaluate whether all identifiers bind the assertions to the benchmarks correctly or not. If one of the identifiers does not work, then the performance tests are not compared against assertions. Else our plugins reports the test

results compared against the identified assertions. Thus, the machine identification is evaluated while evaluating our tools. Notice that the `WildcardIdentifier` always returns `true`. Performance test configurations, which use that identifier, return always the list of configured performance tests. Exactly the opposite does the `DissmissIdentfier`. This identifier returns `false` for all performance test configurations. Consequently, we do not evaluate both identifiers. We use the performance test configuration which is shown in Listing 4.1 to evaluate TeeTime. The performance test configuration of Kieker is shown in Listing 7.1. However, we modify the machine identifies and the corresponding parameters for the system that use these configurations.

MI1: MacAddressIdentifer is used The assertions of the performance tests in Jenkins are bound to a Mac-Address. Thus the configurations uses the `MacAddressIdentifer` and contains the Mac-Addresse of the corresponding Docker container as parameter(s).

For a correct behavior, we expect that on both systems Jenkins visualizes for each performance tests, the corresponding assertions and the result status `Successful`, `Failed`, or `No Result`.

MI2: WindowsComputernameIdentifer is used Our *PC* uses the `WindowsComputernameIdentifier` to execute the performance tests in Eclipse. The computer name is given as parameter.

For a correct behavior, we expect that the Eclipse view shows three performance tests for TeeTime and three performance tests for Kieker. These tests contain the corresponding assertions we defined for the PC.

MI3: NetworkAddressIdentifer is used The `NetworkAddressIdentifier` is used by our *MBP* in Eclipse. The network address is given as parameter.

For a correct behavior, we expect that the Eclipse view shows three performance tests for TeeTime and three performance tests for Kieker. These tests contain the corresponding assertions we defined for the MBP.

8.1.2 Results and Discussion

While evaluating our tools, we used the different identifiers provided by RadarGun to configure the performance tests for the different systems and tools. To identify the system correctly, we fixed bugs in the two Identifiers `MacAddressIdentifier` and `WindowsComputernameIdentifier`. Afterwards, all identifiers properly identified the corresponding system and the assertions were bound to the benchmarks correctly.

8.1.3 Threats to Validity

We evaluated our Jenkins plugin on two different local systems. To increase the validity of our results our plugin should be tested in another environment with different hardware,

e.g., on the Jenkins server of the Software Engineering Group at the University of Kiel. Our performance testing framework is not feasible for cloud systems. Cloud systems allocate resources dynamically and we are not able detect or consider this during performance testing. Since the machine identifiers request parameters and merely compare them against system variables they can access, unit tests should be used to test these identifiers.

8.2 Evaluating the Progress Monitoring

The progress monitoring is implemented to report the results of performance tests in real time. When executing RadarGun, or one of the plugins that utilize RadarGun, different notifications and visualizations are generated. Using RadarGun we receive for each performance test a notification right before the corresponding benchmark is about to be executed. After a benchmark was executed, we receive the benchmark's result and whether the performance test was successful or not. Since reporting the result `Successful`, `Failed`, or `No Result` (see Section 4.3) is done at the end of each benchmark execution, we evaluate whether each stage in RadarGun works correctly or not. The exporting stage is evaluated when reporting the results in Jenkins. The `SocketWriter` stage, which reports the results via a TCP connection to our Eclipse plugin, is evaluated by the Eclipse plugin.

8.2.1 Methodology and Test Scenarios

To evaluate our enhanced RadarGun, we repeat the test scenarios used by Henning, Wulf, and Hasselbring [21]. However, the result comparison differs from RadarGun's prototype, due to the confidence intervals that were introduced to RadarGun.

S1: The score is within bounds A performance test result is within the bounds, if and only if the whole confidence interval is within the assertion's lower and upper bound.

For a correct behavior, we expect that RadarGun produces a console output, which confirms that the performance test was successful. In Eclipse we expect that the performance test is colored in green. In Jenkins we expect that the build finished with the status `Success`. Notice that a build only finishes with the status `Success`, if the results of all performance tests are within the corresponding bounds.

S2: The score is lower than the lower bound A performance test result undercuts the lower bounds, if the confidence intervals undercuts the assertion's lower bound.

For a correct behavior, we expect that RadarGun produces a console output, which confirms that the performance test has failed. In Eclipse we expect that the performance test is colored in yellow. In Jenkins we expect that the build finished with the status `Failure`. Notice that a build finishes with the status `Failure`, if at least one performance tests result was not within the corresponding bounds.

S3: The score is greater than the upper bound A performance test result exceeds the assertion's upper bound, if the confidence interval exceeds the assertion's upper bound.

For a correct behavior, we expect that RadarGun produces a console output, which confirms that the performance test has failed. In Eclipse we expect that the performance test is colored in yellow. In Jenkins we expect that the build finished with the status `Failure`. Notice that a build finishes with the status `Failure`, if at least one performance tests result was not within the corresponding bounds.

To evaluate the progress monitoring provided by RadarGun, without using our plugins, we execute RadarGun for TeeTime and Kieker in a console. Therefor, we use the command:

```
java -cp RadarGun/target/radargun-2.0.0-SNAPSHOT.jar:teetime-benchmark-
performance-testing/target/benchmarks.jar radargun.RadarGun -cp-assertions
assertions/assertions.yaml -jmh-output .
```

In Eclipse we use our plugin for each project once. The result view shows the progress in real time for each performance test. In Jenkins our plugin uses the pipeline configuration file shown in Listing 5.3. Therefor, we create two Jobs. One job imports the Git repository of the TeeTime benchmarks[3]. The other job imports the Gitlab repository of the Kieker benchmarks[4]. The results are visualized for each job after a build finished.

In the following, we measure for each performance test the number nanoseconds per operation (ns/op).

8.2.2 Results and Discussion

We divide the presentation of our results into three subsections, for each tool one subsection. First, We present the results executed on the MBP. Afterwards, the results obtained on our PC are presented. To avoid too much redundancy, we present the MBP's results for TeeTime and the PC's results for Kieker. If the results differ from each other, we report them for both systems. However, the missing results are presented in the Appendix. All executions measured the number of nanoseconds per operation (ns/op).

Progress Monitoring in RadarGun

We executed RadarGun on our MBP using a console. RadarGun finds the assertions in the file `assertions/assertions.yaml` and properly binds them to the corresponding benchmarks. On execution we received the progress log shown in Listing 8.1. First, all stages are initialized (Lines $1-3$), as described in Section 4.4. On start up all three benchmarks were found and reported by the corresponding progress message (Line 4). The first executed performance test was `Port2PortBenchmark` (Line 5). This benchmark's score was 124.00 (Line 7). Since the asserted interval was $[15.0, 19.0]$ (Line 7), the benchmark's result exceeded the

[3] https://build.se.informatik.uni-kiel.de/teetime/teetime-benchmark.git
[4] https://build.se.informatik.uni-kiel.de/kieker/kieker-performance-tests.git

assertion's upper bound. Thus, the benchmark failed, which was reported by RadarGun correctly (Line 7). Consequently, Scenario $S2$ is shown for RadarGun. For the second performance test, namely `Port2PortWithTermInstanceofCheckBenchmark`, RadarGun reports the start (Line 8) and that the performance test finished successfully (Line 10). The score was 15.63 and the confidence interval for the confidence level 0.95 was $\approx [15.19, 16.07]$. Thus, the confidence interval is within the bounds and Scenario $S1$ is accomplished. The third performance test to execute was `Port2PortWithTermReferenceCheckBenchmark` (Line 11). This performance test failed (Line 13), since it undercut the assertion's lower bound. Its score was 17.98, yet the assertion was $[18.0, 20.0]$. Thus, RadarGun successfully accomplished Scenario $S3$ for the TeeTime performance tests.

```
1  19:38:52.773 [main] DEBUG
       radargun.lib.teetime.stage.basic.distributor.Distributor:Distributor-0
       - numOpenedInputPorts (inc): 1
2  19:38:53.391 [main] DEBUG
       radargun.lib.teetime.stage.basic.distributor.Distributor:Distributor-1
       - numOpenedInputPorts (inc): 1
3  [...]
4  19:38:53.492 [START UP] Found 3 benchmarks.
5  19:38:53.495 [STARTING] teetime.benchmark.Port2PortBenchmark is running
       now
6  [...]
7  19:39:14.434 [FINISHED] teetime.benchmark.Port2PortBenchmark [FAILED]
       Score: 124.00638708349331 CL: 0.95 CI: [121.21520524285819,
       126.79756892412843] (Bounds: [15.0, 19.0]) ns/op
8  19:39:14.435 [STARTING]
       teetime.benchmark.Port2PortWithTermInstanceofCheckBenchmark is
       running now
9  [...]
10 19:39:35.099 [FINISHED]
       teetime.benchmark.Port2PortWithTermInstanceofCheckBenchmark
       [SUCCESSFULL] Score: 15.63788229611647 CL: 0.95 CI:
       [15.198543895212682, 16.077220697020255] (Bounds: [13.0, 17.0]) ns/op
11 19:39:35.099 [STARTING]
       teetime.benchmark.Port2PortWithTermReferenceCheckBenchmark is running
       now
12 [...]
13 19:39:55.769 [FINISHED]
       teetime.benchmark.Port2PortWithTermReferenceCheckBenchmark [FAILED]
       Score: 17.985656328996914 CL: 0.95 CI: [17.837935676755137,
       18.13337698123869] (Bounds: [18.0, 20.0]) ns/op
```

```
14 19:39:55.769 [SHUTDOWN] Finished all performance tests
```

Listing 8.1. Reported progress by RadarGun on executing performance tests for TeeTime on our MBP

Due to its similarity, we relinquish to report the results we obtained on our PC in this section, yet the missing results are presented in Listing 1. In the following, we report the progress monitoring by RadarGun for the performance tests in Kieker on our PC.

Listing 8.2 shows that RadarGun found the assertions in the file `assertions/assertions.yaml` and properly bound them to the corresponding benchmarks. First, all stages are initialized (Line $1-3$), as described in Section 4.4. On start up all three benchmarks were found and reported by the corresponding progress message (Line 4). The first executed performance test was `BlockOnFailedInsertBehavior` (Line 5). The benchmark's score was 329.44 (Line 7). Since the asserted interval was $[480.0, 560.0]$ (Line 7), the benchmark's score undercut the assertion's upper bound. Thus, the benchmark failed, which was reported by RadarGun correctly (Line 7). Consequently, Scenario $S3$ is shown for RadarGun executed on the PC. For the second performance test, named `CountOnFailedInsertBehavior`, RadarGun reports the start (Line 8) and that the performance test failed (Line 10). The score was 332.95 and the confidence interval for the confidence level 0.95 was $\approx [331.75, 334.15]$ (Line 10). Although the score was within the assertion's bounds ($[322.0, 332.0]$), the performance test failed. The confidence interval exceeded the assertion's upper bound. Thus, the Scenario $S2$ is accomplished. The third performance test to execute was `DoNotInsertBehavior` (Line 11). This performance test finished successfully (Line 13), since the score and the confidence interval are within the assertion's bounds. Its score was 37.08 and its assertion was $[35.0, 40.0]$. Thus, RadarGun successfully accomplished Scenario $S1$ for the Kieker performance tests.

```
1 15:51:26.678 [main] DEBUG
      radargun.lib.teetime.stage.basic.distributor.Distributor:Distributor-0
      - numOpenedInputPorts (inc): 1
2 15:51:26.990 [main] DEBUG
      radargun.lib.teetime.stage.basic.distributor.Distributor:Distributor-1
      - numOpenedInputPorts (inc): 1
3 [...]
4 15:51:27.171 [START UP] Found 3 benchmarks.
5 15:51:27.317 [STARTING]
      kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior is running now
6 [...]
7 15:51:48.169 [FINISHED]
      kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior [FAILED]
      Score: 329.44060131618085 CL: 0.95 CI: [328.19397304312554,
      330.68722958923615] (Bounds: [480.0, 560.0]) ns/op
8 15:51:48.180 [STARTING]
```

```
        kieker.dumpwriter.benchmark.CountOnFailedInsertBehavior is running now
 9  [...]
10  15:52:08.793 [FINISHED]
        kieker.dumpwriter.benchmark.CountOnFailedInsertBehavior [FAILED]
        Score: 332.95550983887625 CL: 0.95 CI: [331.7533883547323,
        334.1576313230202] (Bounds: [350.0, 400.0]) ns/op
11  15:52:08.794 [STARTING] kieker.dumpwriter.benchmark.DoNotInsertBehavior
        is running now
12  [...]
13  15:52:29.235 [FINISHED] kieker.dumpwriter.benchmark.DoNotInsertBehavior
        [SUCCESSFULL] Score: 37.085234391494666 CL: 0.95 CI:
        [37.01250715748098, 37.15796162550835] (Bounds: [35.0, 40.0]) ns/op
14  15:52:29.238 [SHUTDOWN] Finished all performance tests
```

Listing 8.2. Reported progress by RadarGun on executing TeeTime performance tests on our PC

Again, we relinquish to report the progress monitoring of RadarGun executed on our MBP, due to the similarity of the logs. Nevertheless, we present the missing results in Listing 2.

All in all, the console output is in accordance with our expectations. We proofed that our enhanced version of RadarGun reports the results in real time. Additionally, it provides more information than its prototype. Furthermore, we do identify the machines correctly. However, RadarGun should report the identified machine and the properly mapped assertions and benchmarks. Currently, a software engineer, who did not write the performance test, can not be sure, whether the machine was identified correctly or not.

Progress Monitoring in Eclipse

Figure 8.2 shows the results for TeeTime's performance tests in Eclipse executed on our MBP. The view, which is provided by our Eclipse plugin, received at start up the three benchmarks. The currently executed benchmark `Port2PortWithTermReferenceCheckBenchmark` is marked as `Running` in the `Score` field. The performance tests `Port2PortWithTermInstanceofCheckBenchmark` finished successfully and thus, is colored in green. The performance test `Port2PortBenchmark` has failed, since it undercuts its assertion's lower bound. Hence, this performance test is colored in yellow. Notice, that performance tests, which are currently executed, are colored in yellow, too. The result bar shows that 67% of the performance tests were completed. Hence, the result bar works correctly. After all performance tests were completed, our view looked like in Figure 8.4. The performance test `Port2PortWithTermInstanceofCheckBenchmark`, which was marked as `Running` in Figure 8.4, exceeded the assertions upper bound. Thus, this performance test has failed and is colored in yellow. Consequently, we proofed that all three scenarios worked correctly for TeeTime using our Eclipse plugin on the MBP. The corresponding result on our PC are shown in Figure A.2.

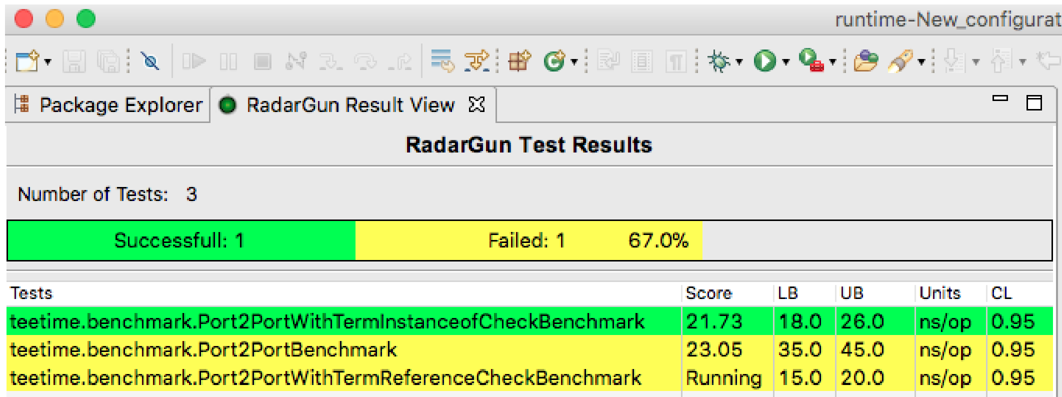


Figure 8.2. On our MBP two of three performance tests for TeeTime have finished and one is still running.

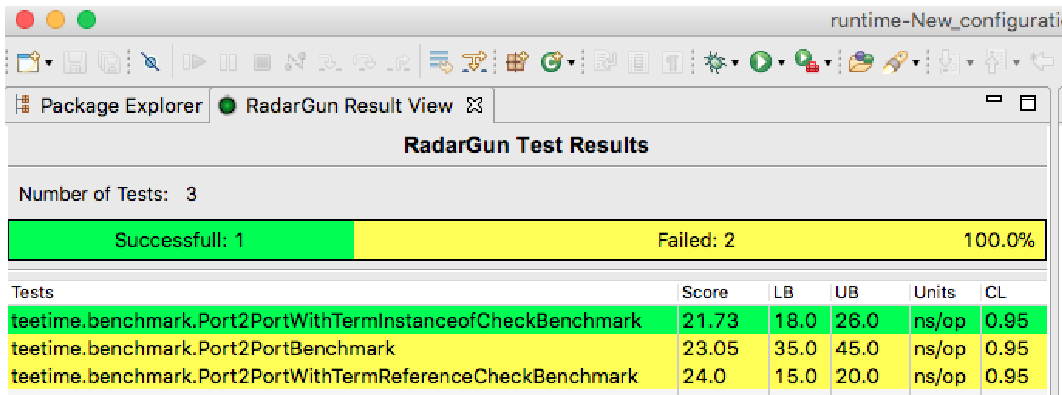


Figure 8.3. All performance tests for TeeTime have finished on our MBP.

Figure 8.4 shows the results for the Kieker performance tests in Eclipse on our PC. The view, which is provided by our Eclipse plugin, received at start up three benchmarks. The currently executed benchmark `DoNotInsertBehavior` is marked as `Running`. The other two performance tests already finished. The performance test `BlockOnFailedInsertBehavior` undercut its assertion's lower bound and hence, has failed. Thus, it is correctly colored in yellow. Although the score of the performance test `CountOnFailedInsertBehavior` is within its assertion's bounds, it has failed. The corresponding confidence level was not within the assertions bounds. Consequently, this performance tests is colored correctly. The result bar shows that 67% of the performance tests were completed. After all performance tests were completed, our view looked like in Figure 8.4. The performance test `kieker.dumpwriter.benchmark.DoNotInsertBehavior`, which was marked as `Running` in

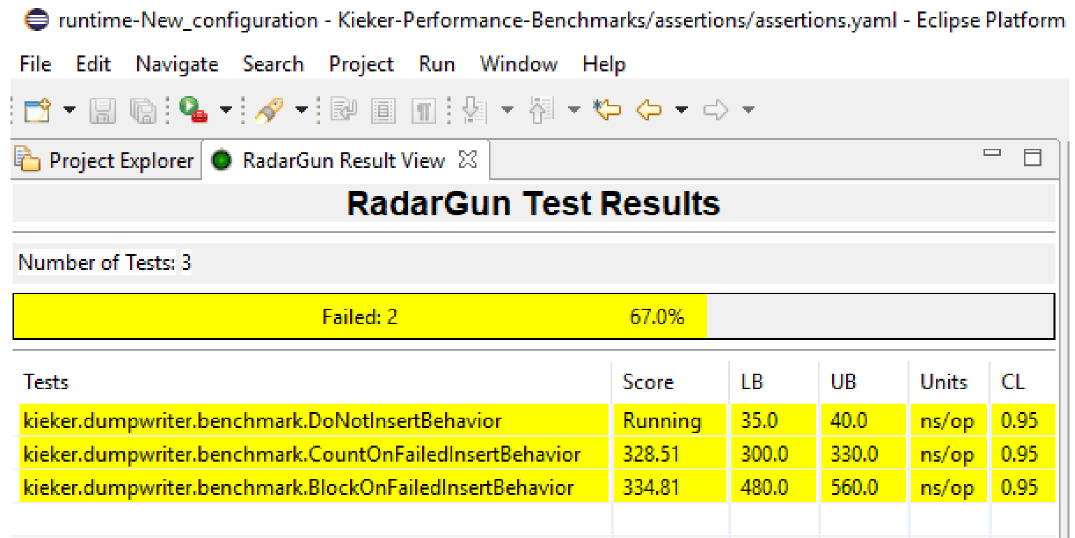


Figure 8.4. On our PC two of three performance tests for Kieker have finished and one is still running.

Figure 8.4, finished successfully. The score 37.09 is within the corresponding bounds [35, 40] and thus, is colored in green. The corresponding result on our MBP are shown in Figure A.1.

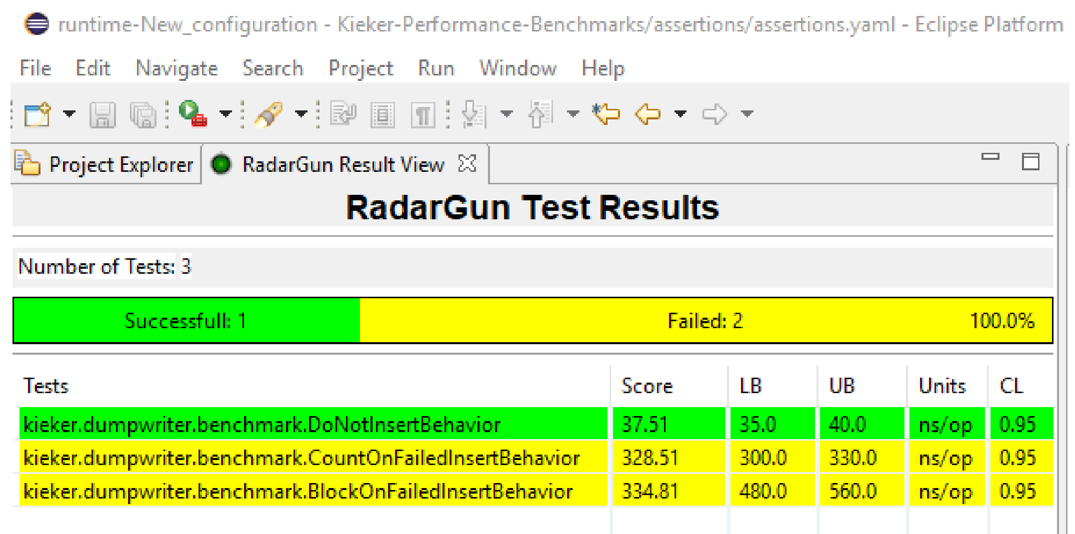


Figure 8.5. All performance tests finished on our PC.

In summary, the execution and visualization of performance tests in our Eclipse plugin is in accordance with our expectations. The progress was shown in real time. Due to the coloring, we easily see how a performance test performed. The GUI fits in the IDE on both operating systems.

Nevertheless, this plugin needs improvements. When starting two performance tests in parallel, Eclipse throws exceptions we did not handle properly. Due to the exceptions, the process does not quite automatically and one has to abort it manually. Moreover, when executing a second performance test session, without closing the view of the previous session, both views receive the data sent by RadarGun. Additionally, the view is still shown after Eclipse was restarted, yet empty. The colors are too bright and the currently running benchmark should be distinguishable from the failed performance tests. If a performances test's score is within the bounds, yet the confidence level is not within the bounds, then a software engineer can only guess the reason why the test failed. Thus, the GUI needs to provide more details of the performance test results.

Progress Monitoring in Jenkins

Jenkins' progress monitoring is a time approximation that depends on the runtime of the previous builds. If the previous build took two minutes to build the job, then the progress bar shows that 50% of the build have finished after one minute. However, RadarGun prints the progress to Jenkins' console. Apart from that, we report the performance test results for each build after the build finished.

Figure 8.6 shows the performance tests results for build number 39 in the job `performance-testing-eval-mac`. This job builds a branch that Jenkins pulled from Git for the evaluation of TeeTime on our MBP.

Performance Test Ergebnisse für Build #39

#1

Enthaltene Tests: 3

Fehlerhaft: 0 Fehlgeschlagen: 0 Erfolgreich: 3

#2

Alle fehlgeschlagenen Tests

Performance Test	Ergebnis	Untere Schranke	Obere Schranke	Einheit	Konfidenzlevel	Gesamtdauer
Port2PortBenchmark	92.75	85.5	100.0	ns/op	0.95	00:00:13.175
Port2PortWithTermReferenceCheckBenchmark	91.64	86.0	100.0	ns/op	0.95	00:00:12.865
Port2PortWithTermInstanceofCheckBenchmark	92.11	86.0	100.0	ns/op	0.95	00:00:12.906

#3

Enthaltene Pakete

Paket	Enthaltene Tests	Erfolgreich	Fehlgeschlagen	Fehlerhaft
teetime.benchmark	3	3	0	0

Figure 8.6. A build's performance tests were all successful.

The progress bar in Box #1 indicates that all performance test have finished successfully, since it is fully colored in green. However, the collection of all failed performance tests in

Box #2 lists all contained performance tests. This is a bug in an if-statement and thus, this table will display always all performance tests. The package result in the package overview (Box #3) shows three successful performance tests. Thus, the build finished with the status `SUCCESS`. Consequently, we proofed Scenario $S1$ for our jenkins plugin on the MBP.

Figure 8.7 shows the performance tests results for build number 41 Each build contains

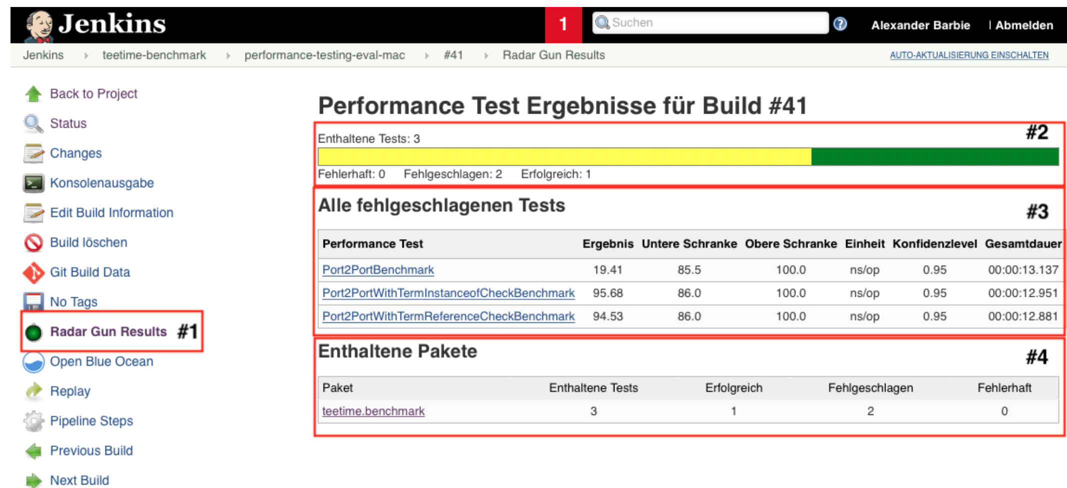


Figure 8.7. Each build contains an overview page for the performance test results.

a menu entry named `RadarGun results` (Box #1). This link references to the result page that was created by our Jenkins plugin. The result bar (Box #2) shows that 2 of 3 tests failed and one finished successfully. However, in Box #3 three test are listed as failed. As mentioned before, this is a bug. To see the detailed results of a failed performance test, we click on the test `Port2PortWithTermInstanceofCheckBenchmark` in Box #3. This link references to the detailed performance test results, as shown in Figure 8.8. Otherwise, one has to navigate through the package results first. The performance test has a score of 95.68 and the assertion was $[86, 100]$. Since the confidence interval $[90.83, 100.52]$ exceeds the assertion's upper bound, the performance test has failed. Since the performance test `teetime.benchmark.Port2PortBenchmark` undercut its assertion's lower bound, this performance test failed, too. Thus, the build finished with the status `FAILURE`. Consequently, we proofed Scenario $S2$ and $S3$ for our Jenkins plugin.

All in all, the performance test result visualization for TeeTime in a single build in Jenkins is in accordance with our expectations. We proofed that our Jenkins plugin imported and visualized the reported data by RadarGun correctly.

As shown in Figure 8.9 the build 66 finished successfully with the status `SUCCESS` for Kieker executed on our PC. The status bar is fully colored in green (Box #2). The build results are referenced by the link "*RadarGun results*" (Box #1). Since all performance tests finished

RadarGun Performance Tests

Performance Test: Port2PortWithTermInstanceofCheckBenchmark für Build #41

Erfolgreich:	false
Einheit	ns/op
Annahme:	[86.0, 100.0, ns/op, 0.95]
Konfidenzinterval:	[90.83, 100.52]
Ergebnis:	95.68
Durchschnitt:	95.68
Maximum:	107.54
Minimum:	90.28
Gesamtlaufzeit:	00:00:12.951

Figure 8.8. Performance test results for the build number 41 of the job Teetime-benchmarks.

successfully, the displayed table in Box #3 contains a bug. By clicking on Box #4 we a navigated to the package overview page. Consequently, we proofed Scenario *S*1 for the Jenkins plugin on our PC.

Jenkins 1 Suchen Alexander Barbie | Abmelden

Jenkins › kieker-performance-tests › master › #66 › Radar Gun Results AUTO-AKTUALISIERUNG EINSCHALTEN

Back to Project
Status
Changes
Konsolenausgabe
Edit Build Information
Build löschen
Git Build Data
No Tags
Radar Gun Results #1
Open Blue Ocean
Replay
Pipeline Steps
Previous Build

Performance Test Ergebnisse für Build #66

Enthaltene Tests: 3 #2

Fehlerhaft: 0 Fehlgeschlagen: 0 Erfolgreich: 3

Alle fehlgeschlagenen Tests #3

Performance Test	Ergebnis	Untere Schranke	Obere Schranke	Einheit	Konfidenzlevel	Gesamtdauer
CountOnFailedInsertBehavior	939.15	820.0	1100.0	ns/op	0.95	00:02:01.411
DoNotInsertBehavior	406.64	390.0	560.0	ns/op	0.95	00:02:01.281
BlockOnFailedInsertBehavior	939.39	820.0	1100.0	ns/op	0.95	00:02:01.711

Enthaltene Pakete #4

Paket	Enthaltene Tests	Erfolgreich	Fehlgeschlagen	Fehlerhaft
kieker.dumpwriter.benchmark	3	3	0	0

Figure 8.9. A build finished successfully.

In Figure 8.10 the package overview page for the package `kieker.dumpwriter.benchmark` is shown. This is the only package that contains performance tests for Kieker. In build 57 the performance test `DoNotInsertBehavior` has failed, since its score (677.36) exceeds the corresponding assertion's bounds ($[390, 560]$). The other two tests finished successfully. Due to the failed performance test, the build finished with the status `FAILURE`. Consequently, we have shown Scenario *S*2 for the Jenkins plugin on our PC.

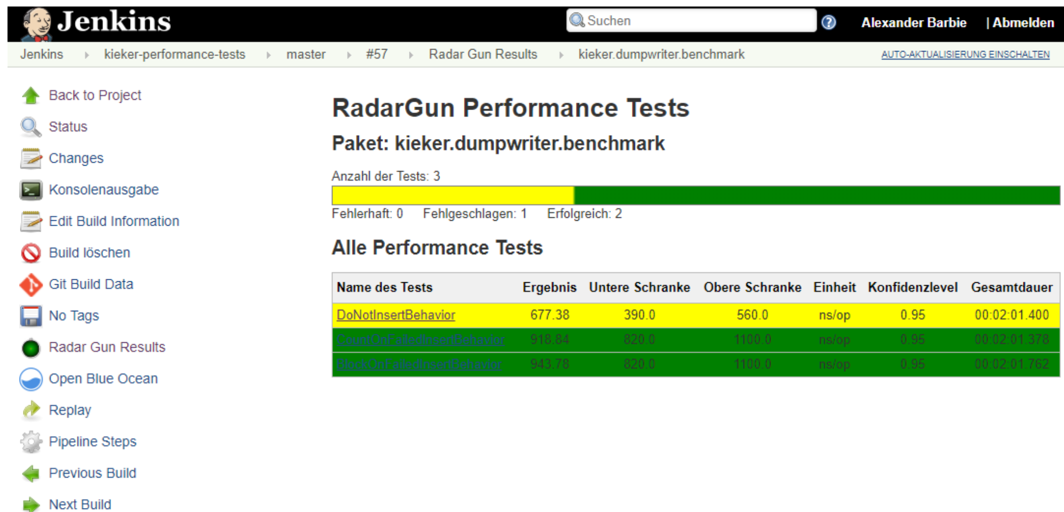


Figure 8.10. A package's performance test result overview page.

All in all, the performance test result visualization for Kieker in a single build in Jenkins is in accordance with our expectations. We proofed that our Jenkins plugin imports and visualized the reported data by RadarGun. However, this visualization still contains a bug in the collection of failed performance tests. The colors of the performance tests in the package overview in Figure 8.10 are too bold. Thus, the links are only readable with an enormous effort. In general, the page layouts should be beautified.

8.2.3 Threats to Validity

Currently, a software engineer, who did not write the performance test, can not be sure whether the machine was identified correctly or not. Although we test in this feasibility evaluation whether RadarGun's performance testing is working correctly or not, the correctness of RadarGun can be tested with unit tests. The P&F framework TeeTime provides an internal Domain-Specific-Language (DSL) to test the stages in unit test. Thus, an automatic unit testing can replace the feasibility evaluation. This way a CI system could build and automatically test RadarGun.

Our Eclipse plugin is missing a proper exception handling. Hence, we can not guarantee this plugin runs stable on all systems.

Jenkins does not report the machines the performance tests were executed on. If the results were executed on two different nodes, they are may visualized in the same job.

8.3 Evaluating the Visualizing of a Build History

This master's thesis main goal is to report performance tests in a continuous integration environment. In Chapter 5 we develop a plugin to utilize RadarGun in Jenkins and report the results. We describe three requirements for this plugin in Section 1.1. In the following, we evaluate, whether our Jenkins plugin satisfies this requirements or not.

8.3.1 Methodology and Test Scenarios

To evaluate these requirements, we simulate the following three scenarios.

R1: Visualize the build history for a performance test To visualize a build history, we have at least two builds. We navigate to a performance test in the build history.

For a correct behavior, we expect that a performance tests is plotted in a chart. Additionally, each single build is presented in a table.

R2: Visualize a group of performance tests In the performance test overview of the build history we compare two performance tests by plotting them.

For a correct behavior, we expect that two performance tests are plotted in the same chart.

R3: Performance tests that were executed with two different run modes can not be compared We execute the benchmark `Port2PortBenchmark` using the changed units `ops/ns`. Afterwards, we navigate to the build history.

For a correct behavior, we expect that this performance test is aggregated as two separate tests. If the visualization works correctly, four tests are presented. Furthermore, we can not compare the two different `Port2PortBenchmark` performance tests, due to the different units. A pop-up message indicates that the units are different, if we try to compare them.

We repeat the methodology used by Henning, Wulf, and Hasselbring [21] to evaluate RadarGun's prototype and run 20 executions for TeeTime and for Kieker. First, we perform ten executions for Scenario $S1$. Afterwards, one execution for Scenario $S2$, followed by six executions for Scenario $S1$ again. Thereafter, one execution for Scenario $S3$ follows. At the end, we execute Scenario $S1$ two times. To simulate deviations in performance, we intentionally decelerate the benchmarks by using JMH's blackhole[5]. It burns CPU cycles according to the given workload value, in our case by a value of 50. To decelerate a performance test to show Scenario $S2$, we use the value 100 in the blackhole. To accelerate a performance test to show Scenario $S3$, we remove the blackhole.

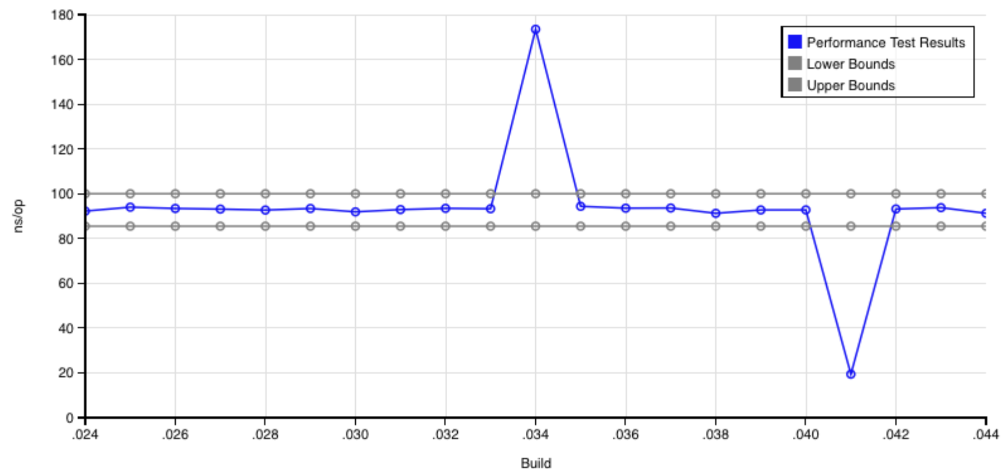
The benchmark configurations in TeeTime and Kieker are different. In TeeTime each benchmark is configured, similar to the evaluation by Henning [19], with one fork, three

[5] `org.openjdk.jmh.infra.Blackhole.consumeCPU(longtokens)`

warm-up iterations and nine measurements. However, in Kieker we use three forks, ten warm-iterations and 30 measurements. This way we evaluate the impact of the hardware and the computed confidence intervals.

8.3.2 Results and Discussion

We executed 20 builds in Jenkins on each system for Teetime and Kieker. The results for TeeTime on the MBP are shown in Figure 8.11. In Figure 8.12 we show the results on our PC.



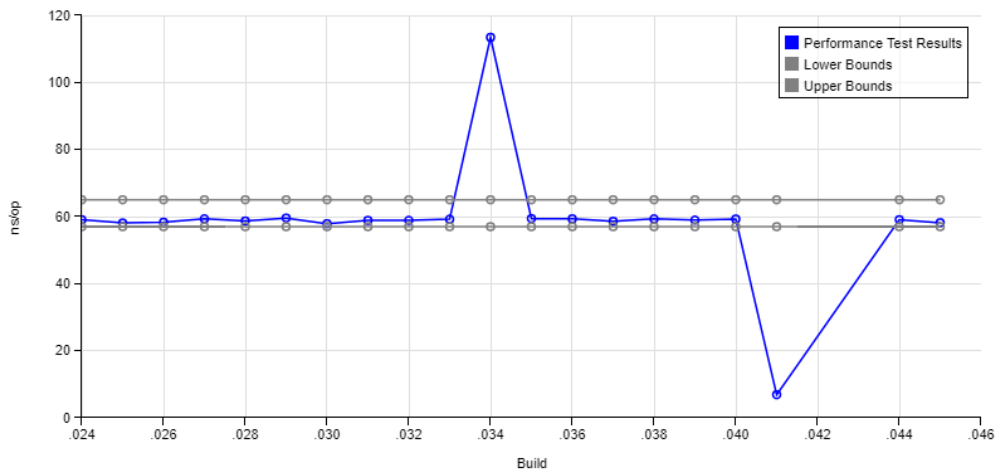
Alle Builds dieses Performance Tests

Build ↑	Name des Tests	Ergebnis	Untere Schranke	Obere Schranke	Einheit	Konfidenzlevel	Gesamtdauer
44	Port2PortBenchmark	91.25	85.5	100.0	ns/op	0.95	00:00:13.018
43	Port2PortBenchmark	93.78	85.5	100.0	ns/op	0.95	00:00:13.151

Figure 8.11. The visualized build history of the performance test Port2PortBenchmark on our MBP

Notice that the build 42 and 43 in Figure 8.12 are missing. We removed both builds, due to a background process that was started by our anti-virus program on our PC during both builds.

On both systems we executed 23 builds to determine the upper and lower bounds for each performance test. Hence, build 24 is the first build we present. The plots look similar to the results presented by Henning, Wulf, and Hasselbring [21]. We decelerated the build 34 to test scenario *S*2. As expected, the score exceeded the corresponding assertion's upper bound on both systems. Analogue, the build 41 undercuts the lower bound on both systems. This build was accelerated to test scenario *S*3. Furthermore, all builds a listed in a



Alle Builds dieses Performance Tests

Build ↑	Name des Tests	Ergebnis	Untere Schranke	Obere Schranke	Einheit	Konfidenzlevel	Gesamtdauer
45	Port2PortBenchmark	58.17	57.0	65.0	ns/op	0.95	00:00:12.657
44	Port2PortBenchmark	59.05	57.0	65.0	ns/op	0.95	00:00:12.564

Figure 8.12. The visualized build history of the performance test Port2PortBenchmark on our PC

table. For both systems the results are as excepted. Consequently, we proofed the Scenario $R1$ for TeeTime on both systems.

However, the results for Kieker are not as expected. Different from the benchmark configurations for Teetime, we configured three forks with each executing 30 measurements. Figure 8.13 shows that the results of the performance test `DoNotInsertBehavior` significantly deviate from each other. The builds 19, 22, 23, 24, and 26 undercut the bounds, although we did not accelerate the benchmark. Furthermore, build number 21 exceeded the upper bound, although we did not decelerate the benchmark. The deviation of the fastest and slowest performance test run is approximately 800 ns/op. Due to the deviation, the confidence intervals of almost every run exceeded or undercut the bounds. However, this is not displayed in this overview. Notice that Figure 8.15a displays 1970 at the origin of coordinates. This is a bug. On the contrary, the builds on our PC were stable. The results of the performance test `DoNotInsertBehavior` executed on our PC are shown in Figure 8.14. Build 57 exceeded the upper bound, since we slowed down the benchmark. Although we did not accelerate the benchmark in the builds 58, 59, 60, 61, 65, and 65, they do not significantly deviate from the accelerated benchmark in build 64. Nevertheless, the performance tests executed on our PC caused less deviation than on the MBP. Hence, we assume that the MBP's hardware is to slow to handle all the executions and additionally the interfering

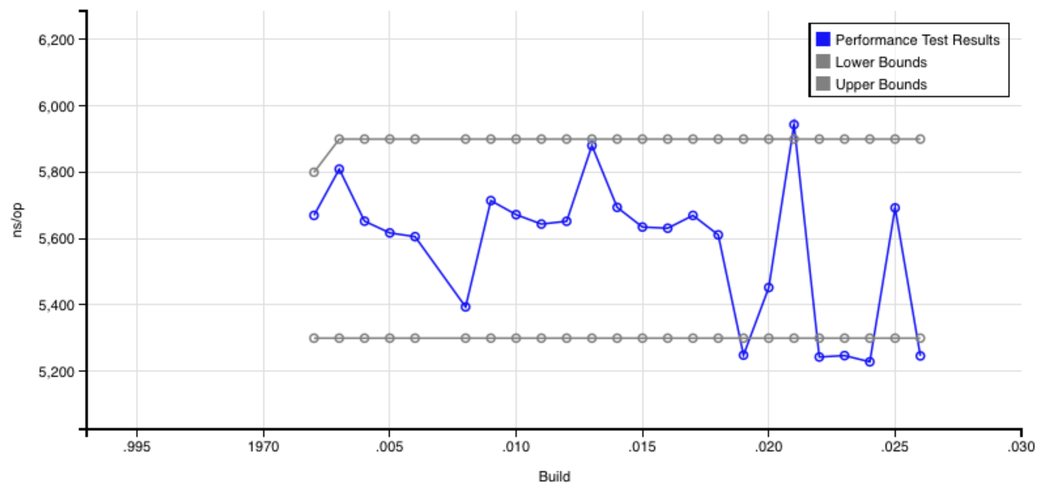


Figure 8.13. Compare three performance tests that implement a different strategy.

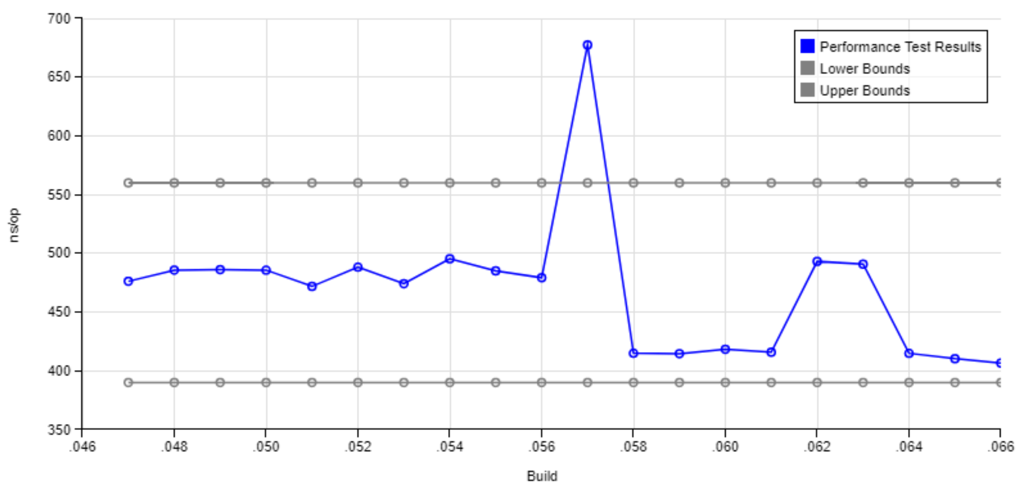


Figure 8.14. Compare three performance tests that implement a different strategy.

events, e.g., caused by the operation system (OS). Due to the execution time, the OS' scheduler may interrupts the performance test execution for a short period of time to run another process. Some executions benefit from the scheduling, whereas others detriment from the scheduling. Furthermore, we assume that we used to few CPU burns on the PC. Thus, we were not able to proof Scenario $R1$ for Kieker on both system, due to a faulty experimental setup.

In Figure 8.15a we compare the performance tests `CountOnFailedInsertBehavior` and `BlockOnFailedInsertBehavior`, which were executed in an extra build on the MBP, to each other. Both performance tests deviated in the execution times from run to run. Whereas, the execution times of the same performance tests executed on the PC, shown in Figure 8.15b, did not significantly deviate from each other.

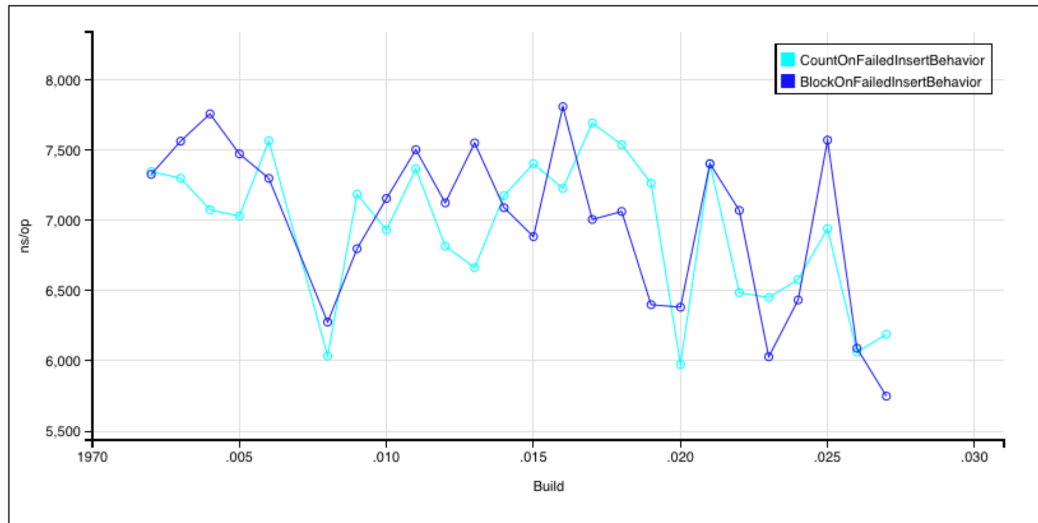
The performance tests are plotted as expected. A performance tests gets plotted by clicking on the corresponding checkbox, as we describe in Section 5.6. Thus, we showed that our plugins fulfill the Scenario $R2$.

We performed an additional build in TeeTime and changed the run mode and run unit of the benchmark `Port2PortBenchmark` to measure the number of operations per nanosecond (ops/ns). As shown in Figure 8.16, both tests are separately listed (Box #3). Thus, our aggregation mechanism distinguished between different units correctly. Additionally, a pop-up window displayed the message "*You can only compare performance tests for the same units*" (Box #1) when we tryed to compare both tests (Box #2). Consequently, we proofed Scenario $R3$ on both systems.

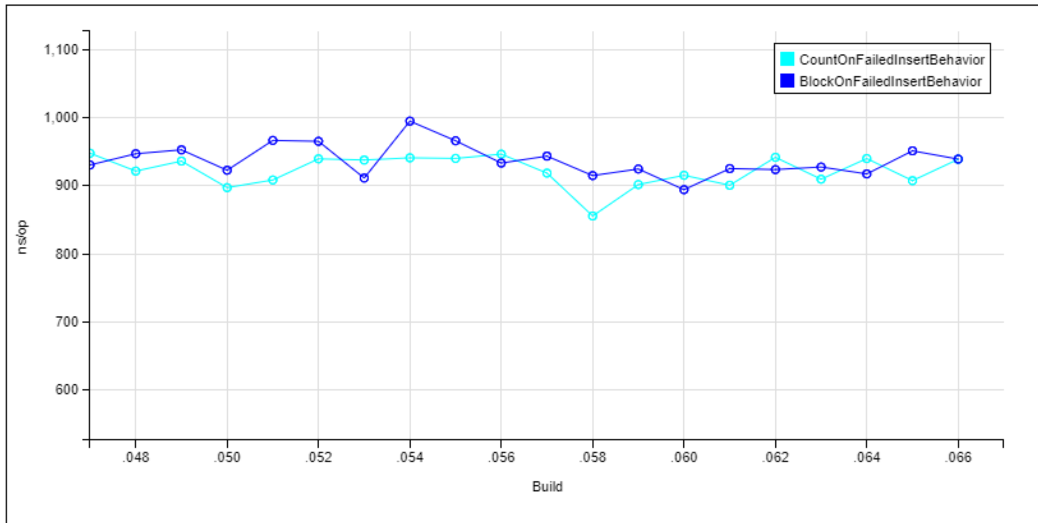
In summary, the visualized build history of performance tests by our Jenkins plugin is in accordance with our expectations. Again, the visualizations should be beautified.

8.3.3 Threats to Validity

We evaluated our Jenkins plugin on two different local systems. To increase the validity of our results our plugin should be tested another environment with different hardware, e.g., on the Jenkins server of the Software Engineering Group at the University of Kiel. Since Jenkins builds could be performed on different nodes, our plugin may fails to visualize the results with regards to the used nodes. RadarGun detects different nodes, yet the visualization does not. Furthermore, the build history for projects, which contain far more performance tests, should be evaluated. The Scenario $R1$ for Kieker should be repeated in another environment.



(a) Comparing two performance tests executed on the MBP.



(b) Comparing two performance tests executed on the PC.

Figure 8.15. Comparing two performance tests for TeeTime that implement a different strategy.

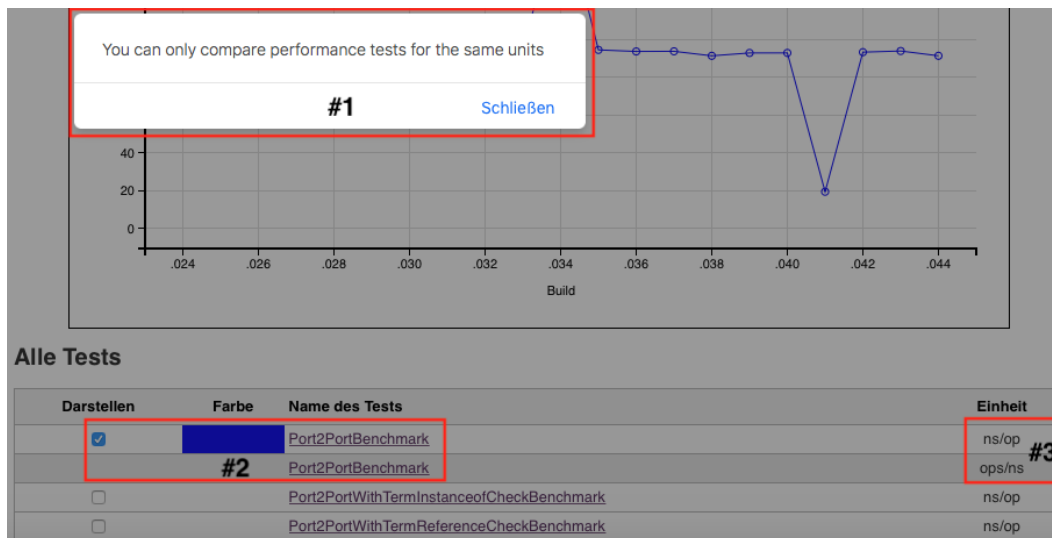


Figure 8.16. Get a pop-up message on trying to compare tests with different units.

Chapter 9

Related Work

We combine several frameworks to enhance and integrate a performance testing framework into a CI environment. A crucial factor for this framework is the free licensing and development possibilities in further research. Hence, we studied open source frameworks, instead of developing all tools from the scratch.

The number of different system for continuous integration is growing since it was proposed the first time by Fowler and Foemmel [17]. The most popular systems are Bamboo [4], Hudson [46], and *Jenkins* [33]. Bamboo is a commercial CI system by Atlassian [4] and is connectible with all the other products by Atlassian. Hudson was found by employees of Sun Microsystems, which left the company after the acquisition by Oracle. Today, Hudson is developed by the Eclipse Foundation and is licensed under the Eclipse Public License [46]. The former developers of Hudson continued on their own, after a dispute with Oracle about the direction of Hudson and created a fork of Hudson called Jenkins. Jenkins is licensed under the MIT-license and open source [33]. The open source plugin Daley and Nielsen [13] visualizes given data in different graphs. However, this plugin plots data in a statically way and users are unable to interact with this data dynamically. Furthermore, each plot has to be configured as post build step in Jenkins. Johanson [29] developed a JavaScript plotting library based on D3.js [10] for visualizing large data sets. This library is named CanvasPlot [29] and was developed to visualize interactive plots of OceanTEA [30]. Johanson et al. [30] developed OceanTea, to interactively explore and analyze climate-relevant time series data, gathered by Ocean observation systems. The advantage over other frameworks is that we are able to dynamically zoom-in or zoom-out in the plots.

In advance of the decision for our approach, we studied several Java frameworks that are claiming to be performance test frameworks. One of these frameworks we studied is JMeter [3] by the Apache Software Foundation [3]. JMeter is designed to load test functional behavior and measure performance [3]. Thus, JMeter is not a framework to execute performance tests that succeed or fail with regards to their runtime. One promising framework is ContiPerf2 by Bergmann [7] that extends JUnit tests and measures runtimes of these tests. Users are able to define limits, such as max and average, for runtimes and also a confidence level the measurements have to meet. The disadvantage of ContiPerf2 is the lack of hardware-depending assertions for tests. The frameworks JUnitPerformance [12] and JUnitBenchmarks [11] also are extensions for JUnit. Both tools are benchmarking frameworks and do not meet our requirements for a testing framework. Furthermore, the developers of JUnitBenchmarks Carrot Search Lab [11] marked their tool as deprecated

[11] in favor of the mircobenchmarking framework *Java Microbenchmarking Harness* by *Java Microbenchmarking Harness*.

Since various factors affect the performance of Java, it is not trivial to benchmark. The given Java application and its input affect the performance, so do the virtual machine and the garbage collector, and many other factors [23] [8]. Georges, Buytaert, and Eeckhout [18] evaluate up to 50 different papers with regards to the methodologies of Java performance test. As a result, Georges, Buytaert, and Eeckhout [18] point out the importance of statistically rigorous data analysis for dealing with non-determinism. Some problems of the surveyed methodologies are the way to report the results, the way to iterate the benchmarks, and the reproducibility of the experiments. It makes a difference whether the average or the maximum runtime is presented. Further, some iterate the benchmark multiple times within a single VM invocation, while others consider multiple VM invocations and iterate a single benchmark execution. There are many more aspects that can lead to incorrect conclusion when interpret the results of a performance test. As a consequence, we focus especially on the methodologies we use in our framework.

At the University of Kiel different tools we developed to measure performance. One of these tools is Monitoring Overhead Benchmark (MooBench) by Waller [54]. MooBench has been developed to measure and quantify the overhead of monitoring frameworks, such as Kieker [22]. Furthermore, MooBench is applicable in a continuous integration environment [55]. However, this tool measures performance benchmarks, yet no performance tests [55]. Another tool to measure performance is the tool Regression Benchmarking Execution Environment (RBEE) developed by Moebius and Hasselbring [43]. This tool claims to have benchmarking capabilities for detecting performance regressions of Java-based software and is applicable in a continuous integration environment [43]. However, we were not able to install and use it, since it is still under development. Furthermore, this tool does mesaure performance, yet does not test performance. A third tool that was developed to measure performance is RadarGun [20] by Henning, Wulf, and Hasselbring [21]. RadarGun was developed in in a student's research project by [19]. RadarGun utilizes JMH to execute and measure micro-benchmarks. The results are compared against predefined assertion, which are defined for specific hardware. If the result is within the bounds of the given assertion's, RadarGun reports that the performance test finished successfully. Otherwise, RadarGun reports that the test has failed or contains no results. Thus, this tool meets our requirements for performance tests. We enhanced this framework, as described in Chapter 4, to report a progress monitoring in real time and to report data in a continuous integration environment.

Chapter 10

Conclusions and Future Work

We enhanced the performance testing framework RadarGun [21] in Chapter 4. Additionally, we developed two plugins to execute and visualize the performance tests executed by RadarGun. One plugin was developed for the CI environment Jenkins [33], which is described in Chapter 5. The second plugin was developed for the IDE Eclipse [16] and is described in Chapter 6. The performance testing framework RadarGun utilized the P&F framework TeeTime. Hence, its whole architecture is built upon pipes and filters. Though, RadarGun was not fully exploiting the potential of a P&F architecture. The `Benchmark Runner` stage (see Figure 3.1) blocked the process until all benchmarks were executed. If and only if all benchmark were finished, the data was reported. Hence, no progress monitoring was possible. Furthermore, the assertions for each benchmark were looked up and compared in the same stage, after all benchmarks finished. This is contrary to the idea of separating a performance test's configuration from the benchmark's configuration in Java. Thus, we split up the stages *Benchmark Runner* and *Result Comparator* by abstracting their super tasks into stages that solely conduct one single task. The improved architecture is shown in Figure 4.1. As a consequence, we now receive a progress monitor during execution. Additionally, developers configure the run mode, the measured timeunit, and the confidence level via the performance test configuration file. Thus, the benchmarks do not have to be recompiled to execute them in another run mode or measured time unit. Supplementary to the CSV exporting, developers can chose further export formats, such as JSON or XML. Therefor, we introduced a model using the annotations provided by Jackson, a JSON processor framework for Java. As a consequence, RadarGun exports (1) the performance test name, (2) the corresponding assertion, (3) the score, (4) the corresponding confidence interval, (5) the average, mininum, and maximum values, (6) and whether the performance test finished successful, did fail or has no result. Frameworks that include RadarGun are able to import the data and utilize RadarGun's model objects to report and visualize the performance test results.

As described in Chapter 5, we developed a plugin for automatic testing in the continuous integration environment Jenkins. This plugin provides a post build step to report the performance test results for each build. Additionally, a build history visualizes the results for each performance test in each build. Thereby, we are able to compare different performance tests in the same chart. This post build steps imports the data that was exported by RadarGun in a previous build step. Imported performance tests are part of Jenkins' object model and thus, define an URL subspace. The data is rendered by the framework *Stapler*

[36]. The labels in this renderings are internationalized, e.g., in English and German. To plot the data, we included the visualization framework CanvasPlot, based on Javascript and the framework D3 [10]. This framework allows us to plot the performance test results dynamically and to zoom in or out in each plot.

In Chapter 6, we developed a plugin to manually execute performance test in the integrated development environment Eclipse [16]. We developed this plugin using the Eclipse 4.*x* API. The Eclipse 4.*x* API provides features, such as dependency injection and an event broker. The event broker allows to use the publish-subscribe pattern in Eclipse components. Since the Eclipse IDE is running in the compatibility mode, we still had to use the Eclipse 3.*x* API at some occasions. In Eclipse we defined a launch configurations and a shortcut to launch RadarGun and visualize the test results. The shortcut enables us to start RadarGun via the *"Run As"* right-click menu in Eclipse. Using this shortcut, a default group launch configuration for a given project is built automatically, if no default configuration was found. A group launch configuration consists at least of one Maven configuration to build the performance tests and of a RadarGun launch configuration to launch the RadarGun framework. All components utilize the built-in event broker to subscribe to specific events. A launch listener notifies the subscribers that RadarGun was launched. Afterwards, a client, which connects with RadarGun via TCP, starts. This connection is an unilateral connection between RadarGun and the Eclipse plugin. Each performance tests is sent via TCP to Eclipse and visualized in the subscribed GUI. The GUI shows the progress of the running performance tests and the results. Furthermore, we are able to jump into the the code of a benchmark, by double-clicking on the performance test's name in the GUI.

We evaluated our tools in a feasibility evaluation and presented the result in Chapter 8. To increase the external validity, we conducted the evaluation on two different system. One system was a MacBook Pro using the operation system MacOS High Sierra and was named MBP in our evaluation. The second system is named PC and runs the operation system Windows 10 Education. The specifications are shown in Table 8.1. On both systems we executed performance tests for the P&F framework TeeTime [57] and the monitoring frameworking Kieker [22]. The performance tests in TeeTime were already used to evaluation RadarGun's prototype by Henning, Wulf, and Hasselbring [21]. In Chapter 7 we showed by an example how to write performance tests for Kieker. Both projects were used to evaluated RadarGun, the Jenkins plugin and the Eclipse plugin. Thereby, we focused on the machine identification, the newly implemented progress monitoring. In Jenkins we evaluated the visualization of a build history for performance tests. To evaluate these functions, we simulated several test scenarios. We present a brief overview of the evaluation's setup in Figure 8.1.

Since the machine identification was not evaluated by Henning, Wulf, and Hasselbring [21], we evaluated the machine identification provided by RadarGun's prototype, first. The identifiers `MacAddressIdentifer` and `WindowsComputernameIdentifier` contained bugs, which we fixed and evaluated again. To identify the Jenkins instances, we used the `MacAddressIdentifer`. In Eclipse our MBP used the `NetworkAddressIdentifier`. Our PC

used the `WindowsComputernameIdentifier` in Eclipse. All systems were correctly identified. Thus, the machine identification was successfully evaluated. The progress monitoring by RadarGun reported as well for TeeTime (see Listing 8.1) as for Kieker (see Listing 8.2) all results on both system correctly. Thus, we showed that RadarGun's progress monitoring is working.

We successfully showed the progress monitoring in Eclipse, too. Nevertheless, when starting a performance tests while another one is still executing, Eclipse throws exceptions we did not handle properly. Furthermore, a view, which visualized the finished performance tests, does no close when a new performance test run session starts. Hence, two or more views visualize the results. The colors, which are used to mark the performance test according to their results, are to bright. Moreover, a currently running performance tests can not be properly distinguished from a failed performance tests, since both are colored in yellow.

In Jenkins we used 20 builds to evaluate the reports for a single build and for a build history. Therefor, we used the methodology used by "RadarGun: Toward a Performance Testing Framework". Build 11 was decelerated to exceed the corresponding assertion's upper bounds. To undercut the lower bounds, we accelerated Build 18. For TeeTime we showed results similar to the results for RadarGun's prototype on both systems [21]. However, we executed three forks, five warm-up iterations and 30 measurements for Kieker benchmarks. Thus, we executed more measurements than for TeeTime and the results differ significantly. On the MBP the performance tests deviated from run to run by up to 800 ns/op. Consequently, most of the performance tests have failed, although the score was within the corresponding assertion's bounds. The reason was the confidence interval, which exceeded or undercut the bounds. We guess that the MBP's hardware is not fast enough to execute the performance tests before the scheduler interrupts the process. On the better hardware of our PC we did not measure this large deviation between builds. Nevertheless, we proofed that each build reports the performance test results. Furthermore, we showed that the build history visualizes aggregated performance tests correctly. Performance tests that are equal in name, yet differ in the run mode or timeunit, are aggregated separately. We demonstrated that two or more performance tests are comparable, if and only if the units are equal. If the units differ, a pop-up message indicates the difference. Furthermore, we illustrated that the build history for each build plots the data correctly. However, we found some bugs in the visualization, e.g., an faulty if-statement that returns all performance tests, instead of all test that have failed. This bug causes that a build's performance tests are all displayed in the table for failed performance tests. We fixed that bug after the evaluation was finished. Similar to our Eclipse plugin, the colors that are used to mark whether a performance tests finished successfully, has failed, or had no results, are to bold. The visualization of the reports should be beautified in general. All in all, the feasibility evaluation was in accordance with our expectations.

The feasibility evaluation emphasizes some features that would improve our tools. In a

future work, the progress monitoring by RadarGun should report the system that was identified on start up. Due to the enhanced architecture of RadarGun, stages solely conduct single tasks. This stages can be evaluated by unit tests. TeeTime provides an internal DSL to write unit tests for the stages. This way RadarGun can be continuously developed and automatically build and tested using Jenkins or another CI system. Additionally, the test result model can be replaced by Enum values, since the methods hasFailed() and `isInBounds()` are redundant. We solely require the type of the class to mark a performance test as was successful, has failed or contains no results.

Especially the Eclipse plugin requires some major improvements. A proper exception handling could prevent the run from crashing, when several performance test run session are started in parallel. If an exception is still thrown, the user receives a quick feedback where and why the exception was thrown. Hence, a proper exception handling could improve the user experience with our tool. Since the colors are to bright, the view should be beautified. Furthermore, a failed performance tests can be colored in yellow, although the score is within the corresponding assertion's bounds. This performance test then failed, since the confidence interval was not within the bounds. However, the confidence interval is not presented. Thus, the performance test results should be presented in detail by clicking on a performance test. This way a user can comprehend the results. The coloring of a performance test, which is currently executed, should be removed. To prevent the user from defining faulty performance test configurations, a configuration GUI for performance tests should be implemented. This GUI could contain a drop down menu to select an identifier. Corresponding to the selected identifier the possible parameters could be presented in a selected dropdown menu, too. Afterwards, a menu presents all benchmarks that are found in the project the configuration is defined for. For each benchmark the user sets the values in fields for the lower bound, upper bound, run mode/timeunit (or vice versa), and the confidence level. As a consequence, we would prevent users from defining faulty performance tests. We mentioned before that we still use the Eclipse 3.*x* API at some occasions, e.g., the panels to select the project and the assertions in the launch configuration. The panels could easily be ported to the Eclipse 4.*x* API, by writing custom panels using the SWT API. Thereby, the launch configurations remain the last parts that use the Eclipse 3.*x* API. If the developers of Eclipse Foundation ever port the launch configuration to the Eclipse 4.*x* API, then the whole plugin uses the latest API.

The Jenkins plugin primarily has to be beatified. The colors and table styles do not appeal to us. The plot library *CanvasPlot - A JavaScript Plotting Library Based on D3.js for Visualizing Large Data Sets* [29] needs improvements. Whether the score, the average, maximum, or minimal values are shown should be selectable by the user. Furthermore, the confidence intervals should be visualized as background areas to the plot. Thereby, users easily see why a performance test may has failed, although the score is within the bounds.

To be accessible for other users, our tools should be published on different sites. Our Eclipse plugin could be published via an Eclipse update site. For our Jenkins plugin we use the Jenkins plugin platform. However, we received the invitation to the Jenkins repository

after we conducted our feasibility evaluation and thus, were unable evaluate our plugin on the Jenkins server of the Software Engineering Group at the University of Kiel. Nevertheless, our Jenkins plugin can be downloaded from the Jenkins repository on GitHub [5].

RadarGun's prototype[1] is already published on Sonatype[2], which is a default repository for Apache Maven and other build systems. Maven uses a `groupid` and an `artifactid` to identify dependencies. We changed the `groupid` from our enhanced RadarGun version from `de.soeren-henning` to `de.cau.se`. The `artifactid` represents the name of a tool. In Table 10.1 we present our used `groupids` and `artifactids`. Only the last part is the `artifactid` of the tools. The previous part the `groupid`. Thereby, the tools can be published and maintained by

Table 10.1. The Maven `groupids` and `artifactids` of our tools

Tool	*Artifact ID*
RadarGun	de.cau.se.radargun
RadarGun Eclipse Plugin	de.cau.se.eclipse.radargun
RadarGun Jenkins Plugin	de.cau.se.jenkins.radargun

members of the Software Engineering Group of the University of Kiel. If a developer aims to add a feature or plugin, this can be easily published under that `groupid`. Furthermore, this would help to maintain our plugins for Eclipse and Jenkins. Both plugins still include RadarGun as library and hence, they always require the latest jar-file. If RadarGun gets an update that solely effects RadarGun and not the visualization in Jenkins or Eclipse, these plugins still have to be updated. If RadarGun is published with such an `groupid` on Sonytype or another public Maven repository, our plugins could include RadarGun as Maven dependency.

Finally, we recommend to integrate RadarGun as plugin in *IntelliJ IDEA* [28] and the *NetBeans IDE* [49]. Both IDEs are widely used by developers.

[1] `https://oss.sonatype.org/content/repositories/snapshots/de/soeren-henning/radargun/`

[2] `http://central.sonatype.org`

Bibliography

[1] Apache Software Foundation. *Commons Math: The Apache Commons Mathematics Library*. URL: http://commons.apache.org/proper/commons-math/ (visited on 02/01/2018) (cited on page 20).

[2] Apache Software Foundation. *Jelly*. URL: http://commons.apache.org/proper/commons-jelly/ (visited on 01/20/2018) (cited on pages 44, 45).

[3] Apache Software Foundation. *JMeter*. 1998. URL: http://jmeter.apache.org (visited on 06/13/2017) (cited on page 89).

[4] Atlassian. *Bamboo - Continuous Integration, Deployment & Release Management*. 2007. URL: https://de.atlassian.com/software/bamboo (visited on 06/13/2017) (cited on page 89).

[5] A. Barbie. *RadarGun Reporting - Reporting Performance Tests in a Continuous Integration Environment*. URL: https://github.com/jenkinsci/radargun-reporting-plugin (visited on 02/07/2018) (cited on page 95).

[6] O. Ben-Kiki and C. Evans. *Ingy. YAML Ain't Markup Language (YAML) version 1.2*. 2009. URL: URl:%20http://www.yaml.org/spec/1.2/spec.html (visited on 01/07/2018) (cited on pages 22, 31).

[7] V. Bergmann. *ContiPerf - a Java Library for Measuring Performance*. 2010. URL: http://databene.org/contiperf.html (visited on 08/02/2017) (cited on page 89).

[8] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *Proceedings of the OOPSLA*. 2004 (cited on page 90).

[9] G. Booch. *Object solutions: managing the object-oriented project*. Addison Wesley Longman Publishing Co., Inc., 1995 (cited on page 1).

[10] M. Bostock et al. *D3 - Data-Driven Documents*. 2011. URL: https://d3js.org (visited on 12/29/2017) (cited on pages 24, 89, 92).

[11] Carrot Search Lab. *Junitbenchmarks*. 2013. URL: http://labs.carrotsearch.com/junit-benchmarks-integration.html (visited on 06/13/2017) (cited on pages 89, 90).

[12] M. Clark. *JUnitPerf - Performance Testing for JUnit*. 2008. URL: https://github.com/clarkware/junitperf (visited on 06/13/2017) (cited on page 89).

[13] N. Daley and E. Nielsen. *Plot Plugin*. 2008. URL: https://plugins.jenkins.io/plot (visited on 06/12/2017) (cited on pages 22, 41, 89).

[14] Eclipse Foundation. *SWT: The Standard Widget Toolkit*. URL: https://www.eclipse.org/swt/ (visited on 01/25/2018) (cited on page 56).

[15] FasterXML. *Jackson*. URL: http://wiki.fasterxml.com/JacksonHome (visited on 12/29/2017) (cited on page 32).

[16] E. Foundation. *Eclipse IDE*. URL: https://www.eclipse.org/ (visited on 01/04/2017) (cited on pages 19, 55, 91, 92).

[17] M. Fowler and M. Foemmel. *Continuous Integration*. 2006. URL: https://www.thoughtworks.com/continuous-integration (visited on 06/18/2017) (cited on pages 1, 23, 89).

[18] A. Georges, D. Buytaert, and L. Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the OOPSLA*. 2007 (cited on pages 2, 3, 15, 16, 18, 90).

[19] S. Henning. "Performance Testing Support in a Continuous Integration Infrastructure". Student research project. Institut für Informatik, 2017. URL: http://eprints.uni-kiel.de/38837/ (cited on pages 21, 22, 81, 90).

[20] S. Henning. *The Performance Testing Framework RadarGun*. 2017. URL: https://github.com/SoerenHenning/RadarGun (visited on 08/01/2017) (cited on pages 8, 21, 27, 33, 38, 58, 90).

[21] S. Henning, C. Wulf, and W. Hasselbring. "RadarGun: Toward a Performance Testing Framework". In: *Symposium on Software Performance 2017 (SSP '17)*. 2017. URL: http://eprints.uni-kiel.de/40364/ (cited on pages 2, 3, 7, 22, 27, 31, 41, 61, 67, 68, 70, 81, 82, 90–93).

[22] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, 2012, pages 247–248. URL: http://eprints.uni-kiel.de/14418/ (cited on pages 21, 90, 92).

[23] V. Horký et al. "DOs and DON'Ts of Conducting Performance Measurements in Java". In: *Proceedings of the ICPE*. 2015 (cited on pages 1, 14, 90).

[24] IBM Knowledge Center. *JIT Compiler Overview*. 2017. URL: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.zos.70.doc/diag/understanding/jit_overview.html (visited on 07/03/2017) (cited on page 12).

[25] International Organization for Standardization. *Iso/iec 25010*. 2014. URL: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (visited on 06/18/2017) (cited on page 2).

[26] International Organization for Standardization. "Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE". In: *ISO/IEC 25000:2014* (2014) (cited on page 1).

[27] A. Irle. *Wahrscheinlichkeitstheorie und Statistik: Grundlagen-Resultate-Anwendungen*. Springer-Verlag, 2005 (cited on page 16).

[28] JetBrains. *IntelliJ IDEA*. URL: http://www.jetbrains.com/idea (visited on 12/29/2017) (cited on page 95).

[29] A. Johanson. *CanvasPlot - A JavaScript Plotting Library Based on D3.js for Visualizing Large Data Sets*. 2016. URL: https://a-johanson.github.io/canvas-plot/ (visited on 06/13/2017) (cited on pages 24, 25, 46, 89, 94).

[30] A. Johanson et al. "OceanTEA: Exploring Ocean-Derived Climate Data Using Microservices". In: *Proceedings of the Sixth International Workshop on Climate Informatics (CI 2016)*. 2016, pages 25–28. URL: http://eprints.uni-kiel.de/34758/ (cited on page 89).

[31] *JUnit: Unit Testing Framework*. URL: http://junit.org/junit5/ (visited on 02/01/2018) (cited on page 2).

[32] K. Kawaguchi. *Evaluation of URL: Reference*. URL: http://stapler.kohsuke.org/reference.html (visited on 01/25/2018) (cited on page 43).

[33] K. Kawaguchi. *Jenkins*. 2011. URL: https://jenkins.io (visited on 06/12/2017) (cited on pages 41, 89, 91).

[34] K. Kawaguchi. *Jenkins Wiki - Action and its Family of Subtypes*. 2017. URL: https://wiki.jenkins.io/display/JENKINS/Action+and+its+family+of+subtypes (visited on 01/20/2018) (cited on page 44).

[35] K. Kawaguchi. *Jenkins Wiki - Architecture*. 2016. URL: https://wiki.jenkins.io/display/JENKINS/Architecture (visited on 01/20/2018) (cited on page 42).

[36] K. Kawaguchi. *Stapler*. 2006. URL: http://stapler.kohsuke.org (visited on 01/20/2018) (cited on pages 42–44, 91, 92).

[37] G. Kiczales et al. "Aspect-oriented Programming". In: *ECOOP'97—Object-oriented programming* (1997), pages 220–242 (cited on page 21).

[38] Lars Vogel. *Eclipse 4 Migration Guide*. URL: http://www.vogella.com/tutorials/Eclipse4MigrationGuide/article.html (visited on 01/24/2018) (cited on pages 55, 56).

[39] Lars Vogel. *Eclipse Modeling Framework*. URL: http://www.vogella.com/tutorials/EclipseEMFPersistence/article.html (visited on 01/24/2018) (cited on page 55).

[40] Lars Vogel. *Eclipse - Rich Client Platform*. URL: https://wiki.eclipse.org/Rich_Client_Platform (visited on 01/24/2018) (cited on page 55).

[41] D. Lion et al. "Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems." In: *OSDI*. 2016, pages 383–400 (cited on page 12).

[42] J. McAffer, P. VanderLei, and S. Archer. *Equinox and osgi: the power behind eclipse*. Addison-Wesley Professional, 2009 (cited on page 19).

[43] A. Moebius and W. Hasselbring. "Employing Open Source Software for RBEE Regression Benchmarking Execution Environment". In: *2. Kieler Open Source Business Konferenz*. Volume TR_1609. Technical Reports of the Department of Computer Science at Kiel University, 2016. URL: http://eprints.uni-kiel.de/35408/ (cited on page 90).

[44] S. Oaks. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*. " O'Reilly Media, Inc.", 2014 (cited on pages 11, 12).

[45] OpenJDK. *Java Microbenchmarking Harness*. 2017. URL: http://openjdk.java.net/projects/code-tools/jmh (visited on 06/12/2017) (cited on pages 19, 21, 90).

[46] Oracle. *Hudson - Extensible Continuous Integration Server*. 2008. URL: http://hudson-ci.org (visited on 06/13/2017) (cited on pages 23, 89).

[47] Oracle. *Java Garbage Collection Basics*. 2017. URL: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html (visited on 07/03/2017) (cited on page 12).

[48] Oracle. *JMH - Source code of AbstractStatistics.java*. URL: http://hg.openjdk.java.net/code-tools/jmh/file/fbe1b55eadf8/jmh-core/src/main/java/org/openjdk/jmh/util/AbstractStatistics.java (visited on 01/04/2017) (cited on page 18).

[49] Oracle. *NetBeans IDE*. URL: http://netbeans.org (visited on 12/29/2017) (cited on page 95).

[50] L. Papula. "Mathematische Formelsammlung für Ingenieure und Naturwissenschaftler". In: *Vieweg+ Teubner* 10 (2009) (cited on pages 17–19).

[51] J. Ponge. *Avoiding Benchmarking Pitfalls on the JVM*. Technical report. Java Magazine, 2014 (cited on page 21).

[52] D. Simpson. *Extending Jenkins*. Packt Publisher Ltd, 2015 (cited on page 41).

[53] J. F. Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. " O'Reilly Media, Inc.", 2011 (cited on pages 23, 24, 41).

[54] J. Waller. "Performance Benchmarking of Application Monitoring Frameworks". Doctoral thesis/PhD. Faculty of Engineering, Kiel University, 2014. URL: http://eprints.uni-kiel.de/26979/ (cited on pages 8, 21, 90).

[55] J. Waller, N. C. Ehmke, and W. Hasselbring. "Including Performance Benchmarks into Continuous Integration to Enable DevOps". In: *ACM SIGSOFT Software Engineering Notes* 40.2 (2015), pages 1–4. URL: http://eprints.uni-kiel.de/28433/ (cited on page 90).

[56] C. Wulf, W. Hasselbring, and J. Ohlemacher. "Parallel and Generic Pipe-and-Filter Architectures with TeeTime". In: *International Conference on Software Architecture (ICSA) 2017*. 2017 (cited on pages 21, 29).

[57] C. Wulf and N. Tavares de Sousa. *TeeTime - The Next-generation Pipe-and-Filter Framework for Java*. Last visited: 13.06.2017." 2015. URL: http://teetime.sf.net/ (cited on pages 21, 92).

Appendix

Feasibility Evaluation Results

Progress Monitoring Results in RadarGun

The progress monitoring of RadarGun on the MBP and the PC.

```
1  12:54:06.777 [main] DEBUG
       radargun.lib.teetime.stage.basic.distributor.Distributor:Distributor-0
       - numOpenedInputPorts (inc): 1
2  [...]
3  12:54:08.116 [START UP] Found 3 benchmarks.
4  12:54:08.119 [STARTING] teetime.benchmark.Port2PortBenchmark is running
       now
5  [...]
6  12:54:29.762 [FINISHED] teetime.benchmark.Port2PortBenchmark [FAILED]
       Score: 19.525826346909735 CL: 0.95 CI: [19.158227918217456,
       19.893424775602014] (Bounds: [35.0, 45.0]) ns/op
7  12:54:29.763 [STARTING]
       teetime.benchmark.Port2PortWithTermInstanceofCheckBenchmark is
       running now
8  [...]
9  12:54:50.471 [FINISHED]
       teetime.benchmark.Port2PortWithTermInstanceofCheckBenchmark [FAILED]
       Score: 34.82793388739904 CL: 0.95 CI: [34.26032241267795,
       35.39554536212013] (Bounds: [18.0, 26.0]) ns/op
10 12:54:50.472 [STARTING]
       teetime.benchmark.Port2PortWithTermReferenceCheckBenchmark is running
       now
11 [...]
12 12:55:11.223 [FINISHED]
       teetime.benchmark.Port2PortWithTermReferenceCheckBenchmark
       [SUCCESSFULL] Score: 17.255597568401072 CL: 0.95 CI:
       [16.74379099137456, 17.767404145427584] (Bounds: [15.0, 20.0]) ns/op
13 12:55:11.224 [SHUTDOWN] Finished all performance tests
```

Listing 1. Reported progress by RadarGun on executing TeeTime performance tests on our PC

```
1 08:23:48.905 [main] DEBUG
    radargun.lib.teetime.stage.basic.distributor.Distributor:Distributor-0
    - numOpenedInputPorts (inc): 1
2 08:23:49.532 [main] DEBUG
    radargun.lib.teetime.stage.basic.distributor.Distributor:Distributor-1
    - numOpenedInputPorts (inc): 1
3 [...]
4 08:23:49.636 [START UP] Found 3 benchmarks.
5 08:23:49.640 [STARTING]
    kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior is running now
6 [...]
7 08:24:02.512 [FINISHED]
    kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior [FAILED]
    Score: 474.9080053378 CL: 0.95 CI: [445.467832656009,
    504.34817801959093] (Bounds: [500.0, 550.0]) ns/op
8 08:24:02.513 [STARTING]
    kieker.dumpwriter.benchmark.CountOnFailedInsertBehavior is running now
9 [...]
10 08:24:23.200 [FINISHED]
    kieker.dumpwriter.benchmark.CountOnFailedInsertBehavior [FAILED]
    Score: 488.7567529431356 CL: 0.95 CI: [471.6159780513123,
    505.8975278349589] (Bounds: [450.0, 500.0]) ns/op
11 08:24:23.200 [STARTING] kieker.dumpwriter.benchmark.DoNotInsertBehavior
    is running now
12 [...]
13 08:24:43.888 [FINISHED] kieker.dumpwriter.benchmark.DoNotInsertBehavior
    [SUCCESSFULL] Score: 61.66413406812639 CL: 0.95 CI:
    [61.199985504547485, 62.12828263170529] (Bounds: [60.0, 70.0]) ns/op
14 08:24:43.889 [SHUTDOWN] Finished all performance tests
```

Listing 2. Reported progress by RadarGun on executing Kieker performance tests on our MBP

Progress Monitoring in Eclipse

The results of the performance tests executed with our Eclipse plugin on the MBP and the PC

runtime-New_configuration - Kieker-Performance-Benchmarks/src/main/java/ki

Package Explorer | RadarGun Result View

RadarGun Test Results

Number of Tests: 3

Failed: 2 67.0%

Tests	Score	LB	UB	Units	CL
kieker.dumpwriter.benchmark.DoNotInsertBehavior	Running	60.0	70.0	ns/op	0.95
kieker.dumpwriter.benchmark.CountOnFailedInsertBehavior	450.84	250.0	320.0	ns/op	0.95
kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior	312.09	300.0	360.0	ns/op	0.95

(a) One benchmark is still running.

runtime-New_configuration - Kieker-Performance-Benchmarks/src/main/java/kie

Package Explorer | RadarGun Result View

RadarGun Test Results

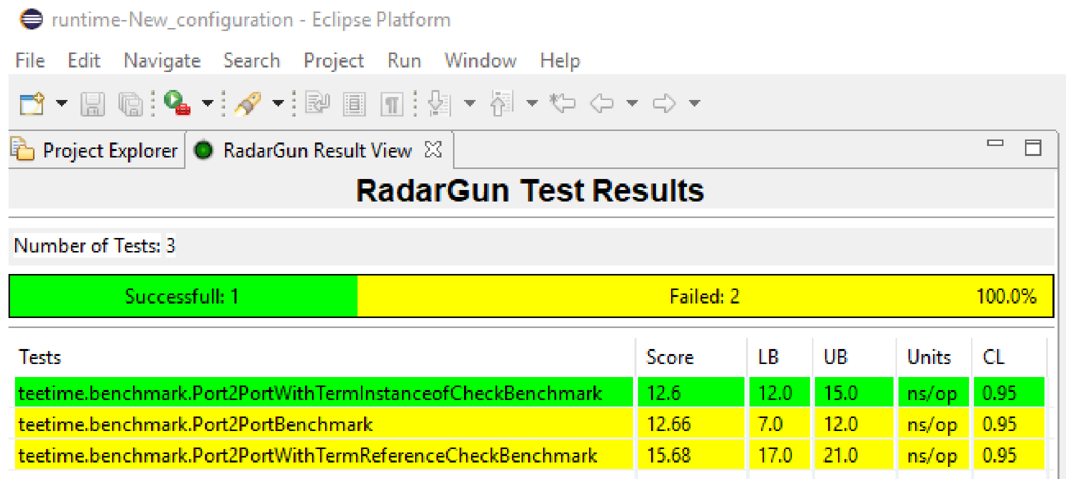
Number of Tests: 3

Successfull: 1 Failed: 2 100.0%

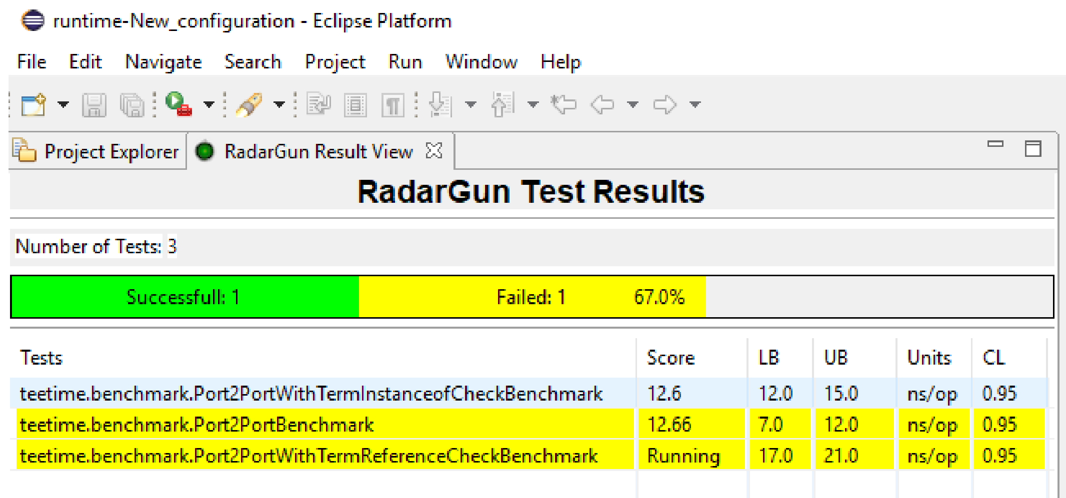
Tests	Score	LB	UB	Units	CL
kieker.dumpwriter.benchmark.DoNotInsertBehavior	61.98	60.0	70.0	ns/op	0.95
kieker.dumpwriter.benchmark.CountOnFailedInsertBehavior	450.84	250.0	320.0	ns/op	0.95
kieker.dumpwriter.benchmark.BlockOnFailedInsertBehavior	312.09	300.0	360.0	ns/op	0.95

(b) All benchmarks finished.

Figure A.1. Execution Kieker performance test using our Eclipse plugin on our MBP.



(a) One benchmark is still running.



(b) All benchmarks finished.

Figure A.2. Execution TeeTime performance test using our Eclipse plugin on our PC.

Performance Test Build Overview in Jenkins

The performance tests results executed and visualized by our Jenkins plugin on the MBP and the PC.

Performance Test Ergebnisse für Build #34

Enthaltene Tests: 3

Fehlerhaft: 0 Fehlgeschlagen: 1 Erfolgreich: 2

Alle fehlgeschlagenen Tests

Performance Test	Ergebnis	Untere Schranke	Obere Schranke	Einheit	Konfidenzlevel	Gesamtdauer
Port2PortWithTermInstanceofCheckBenchmark	59.19	57.0	65.0	ns/op	0.95	00:00:12.447
Port2PortWithTermReferenceCheckBenchmark	58.81	57.0	65.0	ns/op	0.95	00:00:12.415
Port2PortBenchmark	113.44	57.0	65.0	ns/op	0.95	00:00:12.531

Enthaltene Pakete

Paket	Enthaltene Tests	Erfolgreich	Fehlgeschlagen	Fehlerhaft
teetime.benchmark	3	2	1	0

(a) Comparing two performance tests executed on the MBP.

Performance Test Ergebnisse für Build #41

Enthaltene Tests: 3

Fehlerhaft: 0 Fehlgeschlagen: 1 Erfolgreich: 2

Alle fehlgeschlagenen Tests

Performance Test	Ergebnis	Untere Schranke	Obere Schranke	Einheit	Konfidenzlevel	Gesamtdauer
Port2PortBenchmark	6.87	57.0	65.0	ns/op	0.95	00:00:12.569
Port2PortWithTermInstanceofCheckBenchmark	59.2	57.0	65.0	ns/op	0.95	00:00:12.441
Port2PortWithTermReferenceCheckBenchmark	58.35	57.0	65.0	ns/op	0.95	00:00:12.392

Enthaltene Pakete

Paket	Enthaltene Tests	Erfolgreich	Fehlgeschlagen	Fehlerhaft
teetime.benchmark	3	2	1	0

(b) Comparing two performance tests executed on the PC.

Figure A.3. Comparing two performance tests that implement a different strategy.