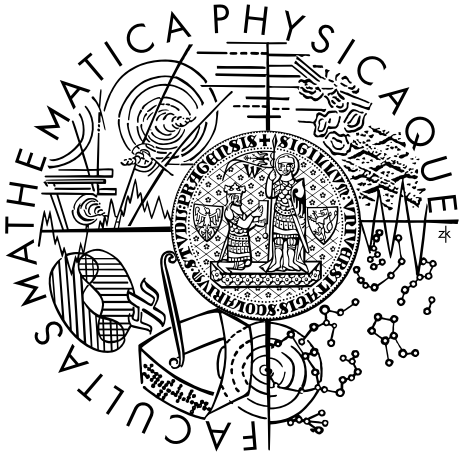
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Jakub Náplava

PerfJavaDoc: extending API documentation with performance information

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Vojtěch Horký
Study programme: Computer Science
Study branch: IOI

Prague 2015

I would like to express my deep gratitude to my supervisor, Mgr. Vojtěch Horký, for the patient guidance, numerous pieces of advice and corrections, without which the thesis would not be possible. I have been lucky to have supervisor like him.

I would also like to thank my family for their never ending support and understanding.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: PerfJavaDoc: rozšíření API dokumentace informací o výkonnosti

Autor: Jakub Náplava

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Vojtěch Horký, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Javadoc je dokumentační nástroj určený pro automatické generování API dokumentace přímo ze zdrojových kódů programu. Takto vygenerovaná dokumentace může u některých metod obsahovat slovní popis použitého algoritmu a jeho asymptotické složitosti, ten je však nepoužitelný v situaci, kdy potřebujeme znát přesnou dobu běhu metody vzhledem k některým charakteristikám. V této práci jsme se rozhodli rozšířit Javadoc o automatické generování výkonnostní části, která vývojářům dovolí změřit výkonnost metody vůči některým předem definovaným vlastnostem. Tyto vlastnosti se specifikují v takzvaném generátoru zátěže, což je metoda, jejímž úkolem je připravit argumenty měřené metody společně s instancí třídy, na které je měřená metoda zavolána. Separace měřené metody od generátoru zátěže pak vývojářům umožňuje jednoduché a srozumitelné psaní a snadné sdílení generátorů. Samotné měření výkonnosti metody pak probíhá na měřícím serveru, který může běžet na určeném referenčním stroji, a byl naprogramován tak, aby poskytoval co nejpřesnější výsledky s ohledem na platformu Java.

Klíčová slova: Java, dokumentace, výkonnost, generování zátěže

Title: PerfJavaDoc: extending API documentation with performance information

Author: Jakub Náplava

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Vojtěch Horký, Department of Distributed and Dependable Systems

Abstract: Javadoc is a documentation tool used for generating API documentation from Java source code. For some methods the generated documentation can contain description of the algorithm and its asymptotic complexity. However, such information is futile when the exact execution time of the method is needed with respect to certain critical characteristics. In this work we decided to enhance the Javadoc tool with a performance extension which permits to measure the performance of the method against the predefined characteristics. These characteristics are specified in a workload generator, which is a method used to prepare the actual arguments and instance for the measured method. The separation of the measured code and the preparatory code allows developers to implement both parts clearly and easily, and also share the generator amongst multiple methods. The measuring itself is performed by a measuring server which may run on a dedicated reference machine, and was programmed to provide accurate results with respect to the Java platform.

Keywords: Java, documentation, performance, workload generation

Contents

1. Introduction

A decision concerning which library is the most suitable one for a particular situation and, therefore, is to be used in a particular project is a routine task of any programmer. There are, however, many factors which need to be considered including the library public API along with its performance.

In order to avoid the need to inspect library methods line by line and read language specifications (standards), numerous documentation tools (such as Doxygen or Javadoc) have been developed. Such tools not only generate the documentation from a source code, but also offer special comments to add additional information about the specific pieces of code. A typical example is a method description containing information about the usage of the method as well as the information about its parameters.

None of these tools, however, contain a mechanism to measure the performance of a particular method on an ad hoc basis, which would be in this instance rather convenient. It is for this reason that we selected one of these documentation tools, namely Javadoc, and decided to add a performance extension.

1.1 Goals

The primary goal of this thesis is to design and implement an interactive measuring tool that will be integrated into the Javadoc tool. The users will be able to select a workload and the parameters they wish to use to test the method performance. The request will be sent to a measuring server which will measure the method under the selected workload and parameters. The measured results will be then continuously updated from preliminary to more accurate results.

The implementation needs to take into account specific behaviour of the Java platform in order to provide accurate measurement results. An important part of the thesis is designing a reasonable interface for the workload generators, i.e. for the code which prepares the actual workload for the measured method.

Another important feature of the implemented tool is to show a trend of the method on a given interval. The simple example is the situation when a collection contains between 1,000 and 10,000 items and we would like to determine the time needed to sort this collection. The displayed results should show tendencies as well as raw measurement data.

Finally, our tool should be ready for possible extensions, which may for example be adding another performance metrics.

1.2 Thesis structure

In the following chapter the issues with performance measuring in general as well as restricted to execution time are discussed. We introduce the idea of the workload generators, analyse the application architecture and compare the tool functionality to other existing solutions. The third chapter is dedicated to the programming documentation which explains the fundamentals of how the tool was made. The fourth chapter then describes the installation of the tool together

with basic settings and demo examples. In the conclusion, we summarize the thesis, suggest possible future work and describe CD contents of the thesis.

2. Problem analysis

In this chapter, we will briefly describe performance measuring in Java together with its issues. Then, we bring up the idea of workload generators and analyse it. After that, we will discuss the application architecture, in which we separate the frontend and backend part of the application and discuss each of them separately. Finally, we will compare our tool with existing solutions.

2.1 Java performance measuring

This section explains the context of our thesis as for performance measuring in Java. It defines the performance measurement and the performance metrics, explains the basics of Java platform with some examples that can make measuring more difficult, defines micro benchmarking with its problems and describes Java timers.

2.1.1 Performance measuring

"Performance measurement is the process of collecting, analyzing and/or reporting information regarding the performance of an individual, group, organization, system or component." [9] We are interested only in a computer system performance, where a performance metric is a measure of some of its properties. The computer system performing task can be generally evaluated with three performance metrics classes. The speed class is for the case, when the system completes task successfully and provides correct results. The reliability class measures situation, when the system completes task, but the results may be incorrect. The availability class then contains metrics for the situation, when the system can be down and thus not performing tasks.

Our intent is to measure methods, which always end regardless of theirs results; therefore we are further interested only in the speed class. The typical examples of this class are metrics like responsiveness metrics (how long does it take to finish the task?), utilization metrics (how much is the system loaded when working on a task?) or productivity metrics (how many tasks per time unit can the system perform?). It would make sense to choose any of these metrics to evaluate the method performance, but we decided to concentrate on the execution time, which is in our opinion one of the most important method properties.

In the next chapters, we will use the term benchmarking, which is the process of measuring particular performance metric, which can then be compared to others.

2.1.2 Java technology overview

Java is a high-level programming language designed to work across multiple software platforms ("Write once, run anywhere"), meaning that compiled Java code can run on all platforms for which JVM (Java virtual machine) exists without the need for recompilation. Java applications are typically compiled to bytecode, which is then executed by a Java interpreter or converted directly into machine

code instructions by JIT (just-in-time compiler). This approach enables to generate specific instructions for particular CPU at runtime and stands for the Java code portability.

Whereas Java interpreter is good for the methods that are called rarely, JIT excels at "hot" methods (methods that are called often). The great advantage of JIT is that it knows the entire environment and with the observations about the actual patterns of execution of specific parts of code, can make better decision about what code optimization to use than any static compiler (such as javac) possibly could. JIT can also deoptimize the already compiled code when its preconditions do not hold anymore. Therefore, JIT makes most of the optimizations, whereas javac can make only the basic ones. These optimizations make the final machine code more effective and faster, however with other Java features such as lazy class-loading they also make benchmarking more difficult. The typical example of such optimization that makes benchmarking more difficult is the dead-code elimination, which will be deeply explained later.

2.1.3 Micro benchmarking in Java

Benchmarking can be generally classified as macro and micro benchmarking. While macro benchmarks measure the performance of a whole computer system, micro benchmarks concentrate only on a specific aspect of it. Because our application measures the method performance in isolation, it is classified as a micro benchmarking harness, so from now on, the thesis deals only with micro benchmarks.

The smaller the method which performance we measure, the stronger impact of:

- benchmark infrastructure
  - Overhead: measurement can incur very big overhead (in terms of execution time, memory usage, disk space, etc.)
  - Perturbation: measurement often needs to add additional code, which can change behavior of the measured system (caches, more instructions, etc.)
- JVM internal processes: our major concern is the garbage collector, that may run any time during our measurement
- operation system internals
- etc.

All these issues distort the measurement and we may actually measure something else than we want. The basic advice to avoid these problems is to keep the benchmarking code as small as possible while performing larger amount of measurements, which must be then statistically processed. These issues with our solutions will be further discussed in the next sections.

2.1.4 Timers

Because we decided to choose the execution time as the performance metric, we will mention the common ways, how to measure time in Java.

- java.lang.System.currentTimeMillis() method : returns value of system time since Epoch (1.1.1970) in milliseconds
- java.lang.System.nanoTime() method : returns value of highest resolution system clock in nanoseconds
- java.lang.management.ThreadMXBean class : provides methods to get CPU time of current threads, however, may be unsupported by VM
- using Java JNI (Java native interface)

Since the *ThreadMXBean* may be unsupported, the precision of *System.currentTimeMillis()* is simply inadequate and using Java JNI has cross platform compatibility issues, we decided to measure time intervals in our application using *System.nanoTime()* method.

According to the documentation, the resolution[1] of all methods is undefined (depends on OS). To illustrate the typical values we can expect, we performed several measurements using the *System.nanoTime()* method on Windows and Linux machines:

JVM	OS	CPU	Resolution
HotSpot 1.8.0_25	Windows 8.1	i5 (x86-64)	430ns
HotSpot 1.7.0_80	Linux 3.18 (Gentoo)	Intel Core 2 (x86-64)	28ns

The measured results correspond to those measured by Alexey Shipilev [11] and imply that on the most of Windows machines, it is not possible to precisely measure methods whose execution time is less than 430ns.

2.2 Workload generators

We want to measure the method performance; therefore we need to generate the parameters, with which the measured method will be called as well as the instance, on which the measured method will be called. We decided to separate measured method and code to prepare the parameters with the instance, which we call workload generator, mainly because the measuring mechanism should add as little of additional code as possible (see section 2.1.3).

2.2.1 Binding generator to the methods

When defining generator as another method, we have to figure out, how to connect a measured method and its generator.

There are generally three options we found usable:

[1] resolution = smallest interval the particular timer can measure

The first one was to make a measuring tool based on so called naming conventions (e.g. a generator name would be derived from the measured method name), which is a concept used for example by JavaBeans. As easy as this approach may seem to be, in our case it hides many problems. The main one is definitely the impossibility to assign multiple generators to one method (as well as assigning same generator to multiple methods), which could be for example when creating workload for collections convenient.

Another option was to have a configuration file, where all relations between a method and its generator would be saved. This approach would allow assigning multiple generators to one method as well as using custom package structure. However, we force programmers to use additional file, which must be then also distributed.

Finally, since version 1.5. of the JDK we may use annotations. The principle is easy. Programmer annotates measured method with appropriate annotation, in which he specifies the location of one or more generators. Then he adds another annotation to generator, which allows us to specify other options like generator description, and the bind is done. This approach does not seem to have any disadvantages mentioned in previous approaches, but because Java lacks methods pointers, we must define generator as a String annotation member, therefore we do not have dynamic compiler check on whether the requested generator exists. Another disadvantage is that we need to have annotations always present, when using the methods, and we also cannot add annotations to the binary distributions.

When deciding which way to go, we first realized, we want to "pollute" measured class code with another measurement code as least as possible. Because of this and already mentioned impossibility of assigning multiple generators to one method (and vice versa), we decided not to use naming convention option.

With the configuration file we do not have to pollute a code with the binding information which also means that methods do not require the configuration file for their run. Our intent is, however, to provide the possibility to measure the method for all method's users, therefore we do not mind having annotations present all the time. We also want to make it easy to see that the method enables measuring of its performance, which is the reason, why we decided to use annotations.

2.2.2 Generator structure

The main task of a generator is to prepare arguments for a measured method together with an instance of a class on which the measured method will be called (let us name these arguments and instance a workload). When measuring performance of methods dealing with external resources like files or databases, we also want generators to clean up the environment after measurement.

Let us now discuss what generator structure would be the best.

The first two options suppose that the benchmarking harness calls them only for preparing workload and cleaning generated content.

The first option we considered was that the generator could return workload as an iterable collection of array of objects, where each object array would contain an instance of a measured class and measured method arguments (see Listing 1).

```
@Generator(...)
public Iterable<Object[]> prepare(...) {
  List<Object[]> data = ...;
  ...
  data.add(new Object[] {objectOnWhichToInvoke, methodArguments});

  return data;
}

//clean-up method
public void cleanUp() {
  ...
}
```

Listing 1: Generator structure returning Iterable <Object[]>

```
@Generator(...)
public Iterable<Object[]> prepare(Workload w) {
  ...
  w.addCall(new Object[] {objectOnWhichToInvoke, methodArguments});

  //clean-up method is in the same class
  w.setHooks(this);
}

@CleanUp
public void cleanUp() {
  ...
}
```

Listing 2: Generator structure passing workloads using argument

The code to return system to its pre-measurement state would be saved in other methods.

The second option was to pass an instance of class, which would support adding workloads to it, as one of generator's arguments. This instance would also support adding methods to return system to pre-measurement state (see Listing 2).

The last option we considered is a little different from the others. Instead of preparing workload for the measured method and leaving the task of calling measured method for the benchmarking tool, we would call the measured method directly in generator. The call of the measured method would start and end with some specific instructions so that it can be found in a bytecode by the benchmarking harness (see Listing 3).

```
@Generator(...)
public void prepare(...) {
  ...
  start();
  measuredMethod(...);
  stop();

  //cleanup directly here
}
```

Listing 3: Generator structure that calls the measured method directly

After writing some simple generators, we realized that we are supposed to write a little bit more code in the first option than in the second option. Otherwise are the first two options quite similar, which is the reason, why we rejected the first approach.

When we are about to compare the second and the third option we must note that the third option seems more straightforward as for the code written in generator, because it does not need to define other methods for removing generated content. The third option would, however, require complex bytecode manipulation (imagine the situation, when the generator writer decides to prepare multiple workloads according to some conditions), which was the reason why we decided to use the second option – annotations.

2.2.3 Parameters

So far we decided how a generator should prepare workload for calling the measured method. Now we need to think of what are the values, from which the generator should prepare appropriate workload as well as what should be the values, in which can user select to perform measurement.

Because a generator is just an annotated method, it can declare parameters. These parameters seem as an easiest and most straightforward way to meet these two requirements at once. The process is simple. The generator writer declares parameters for which he is able to prepare corresponding workload. Then these parameters are shown to a user, who chooses their values and selects to perform measurement with them.

Let us now discuss what type of parameters to allow programmers to use, because, evidently, not all of them can be easily shown to user (e.g. objects). The types that should not be missing are definitely numeric types (integer, float and double), because they allow users to specify all kind of sizes. These types are for our tool also important in specifying range, against which the measurement will be performed. This was one of the thesis goals and is met by these parameters from which exactly one must be set as a range when user creates new measurement request. There are two more types that would, in our opinion, be sufficient to make the tool able to measure all kind of methods, and therefore are by our tool supported. One of these types is Enum that serves for specifying the type (e.g. encryption type) and the second one is then String that can be used to specify all kind of names.

Parameter properties, such as the minimal and the maximal value or the description of the parameter, are then stored as annotations. The main reasons to choose annotations were that we can create them statically, they are close to the code and also the fact that we already use annotations to mark measured methods and generators. We also considered other possibilities, such as the configuration file containing the properties, however, none of them seem as good as annotations.

2.3 Application architecture

The frontend of our application is obvious as we decided to add the performance part to the Javadoc. However, there are generally two possibilities, where a measuring mechanism may be placed. The first option is to integrate the measuring

mechanism directly into the Javadoc (e.g. as an applet). The second option is to separate the measuring mechanism from the Javadoc and thus create a client-server architecture. We decided to choose the client-server approach, mainly because it allows to separate measuring server and thus provide minimal measuring perturbation. The measuring server can then run on a dedicated reference machine.

We will discuss the client (frontend) ant the server (backend) part separately in the following sections.

2.4 Frontend

The goal of the frontend is to enable user to select values in which he wants to measure method performance, send these values to the measuring server and display measured results.

2.4.1 Alternative frontend GUIs

As we already mentioned in the introduction of the thesis, we decided to add the performance part to the Javadoc documentation. The main reason for this was to store performance information about methods on the place, where the programmers look for basic method information. However, when thinking of what the frontend of our application should be, we also considered other possibilities.

The first intuitive idea was making Java application in Swing or JavaFX. These are both Java GUI (graphical user interface) libraries, which would have an advantage while communicating with the measuring server, because both sides would be written in Java. However, we would force the programmers to use another application, which would be for many of them unpleasant. We would also need to make some parser to parse java sources, which is when extending doclet already done by Doclet API.

Another idea was to integrate our tool with some Java IDE (integrated development environment) as for example Eclipse. Because Eclipse shows HTML output and Javadoc is practically just a set of HTML pages, it should not be that difficult to add such performance part to Eclipse after writing our application, which is the reason, why we decided to extend Javadoc instead.

2.4.2 Javadoc modifying

"Javadoc is a tool for generating API documentation in HTML format from doc comments in source code." [7] Javadoc as a tool defines a concept of doclets, that are programs written in Java using the Doclet API to specify the output of the Javadoc tool. HTML documentation is generated by the standard doclet. This is a doclet that we need to redefine in order to add a performance part to it.

2.4.3 Performance part

As already mentioned in section 2.4.2, the standard doclet generates HTML documentation. This documentation contains, among other things, information about

each method. As we can see in Figure 2.1, the method part starts with the name of the method, continues with its signature and description, then reports various things such as method parameters or exceptions. Because we are about to measure method performance, we thought, it would be best to place performance part just after these information while preserving all conventions (e.g. structure).

Figure 2.1: Method *get* of List<T> in Javadoc

When we decided, where to place method performance part, we need to decide, what will be its structure. The performance part is logically divided into two parts. The first part should contain controls that would enable user to choose values, in which the measurement will be performed. The second part should then show measured results in an easy to interpret form.

The controls in the first part are generated from parameters and their annotations from the particular generator as discussed in section 2.2.3. The representation of the controls is as for String and Enum type intuitive, for the numeric types we chose JQuery slider.

We must consider two approaches for the results reporting in the second part. The first one is in the situation, when the user only needs to know the exact measurement results. This can be easily satisfied by showing results in a table form. The second approach is the need to know the behavior of the measured method on a particular interval determined by the selected range. This need is easier presented with a graph, which we assume is more illustrative in presenting tendencies. The selection of appropriate graph library will be discussed in section 2.4.6.

2.4.4 Dynamic web pages

We can think about Javadoc as a collection of static web pages. With a performance part, we would, however, need to make some pages dynamic in order to be able to select values chosen by user, send them to a measuring server and then display results. To make these pages dynamic, we must use some client side scripting language, which is in this case JavaScript with AJAX (asynchronous JavaScript and XML). To simplify JavaScript coding, we decided to use its popular library jQuery [6].

2.4.5 Request priority

One of the goals of the thesis was that the measured results should be continuously updated from the preliminary and possibly imprecise results to more accurate ones. To achieve this functionality, the client side must after receiving measured results decide, whether it wants more accurate results, and if yes then send another request. The decision may be manually made by a user after every response; however, we do not want to bother him with the alerts, whether he finds the results accurate enough. Instead of that we define request priority, which is a number between one and four indicating, how accurate the measured results should be. The priority is sent with every request starting from one in the initial one and with every further request is incremented by one until reaching four, which is the final request. The semantic value of each priority is left on the server and will be thus discussed later.

The concept of priorities was not the only option we considered. There was also the opportunity to specify the number of requests together with their preciseness directly in the Javadoc, or send requests until the server indicates that the results cannot be better. The first option would in our opinion lead to not so clear frontend and the second one would force the server to count the number of requests, which was not the way, we wanted to go.

2.4.6 Graph

The graph can be either generated from the measured results in the Javadoc, or can be sent together with the results from the measuring server. To keep communication protocol simple, we decided to create graph on the client.

Our requirements for the graph library are:

- Open source JavaScript graph library
- We only need 2D line graphs (multiple lines needed) with support for error bars
- Simple API as well as proper documentation is important
- Easily updatable
- Code size should not be too big

There is a lot of JavaScript libraries [4], from which we chose three candidates that meet our requirements: *Chart.js*, *Dygraphs* and *D3.js*. All these libraries have proper documentation full of examples, in which *Dygraphs* charts seem to perform the best mainly because of their zooming ability together with the support for additional error bars. On the other hand, *D3.js* provides box plot charts convenient for showing statistical characteristics and the library *Chart.js* excels in responsiveness. It would make sense to use any of these libraries in the project, nevertheless, we decided to use *Dygraphs* mainly for its zooming abilities, code simplicity and the ability to process large amount of data.

2.5 Backend

The server part of our application is a component, which must be able to accept request from a client, measure the execution time of a method obtained from the parsed request and send a response containing measured results back to the client. Because our application measures Java methods, it makes sense to write the server as a Java application.

The main goal of the server is to measure the execution time of the method as most precisely as possible. As already described in section 2.1, this may be sometimes difficult and we discuss this problematic in the next sections.

The server should be multi-threaded, so that multiple users can measure methods performance at the same time. However, we do not want to allow one user to measure multiple methods at the same time especially in the case, when he runs the measuring server on the same machine, because this may easily lead to the server overload. Therefore every user's request contains also identifier of the user and when the server receives request with identifier for which the measurement is actually being performed, the new request is deferred until the actually performing measurement is done.

The server does not need any GUI, but must be configurable by command line arguments as well as with a configuration file.

2.5.1 Measuring

When the server receives a measuring request from the client, it needs to measure the particular method, which must be present on the server. This section describes the possibilities, how can the server measure the method.

The first intuitive idea of how could the server measure method execution time is using the reflection. The reflection is a tool to inspect classes and allows us to call methods. This call is programmatically easy to perform, but we must count with performance overhead causing our method to run slower than if we called it directly [8]. Due to this performance overhead is the reflection sufficient only for the preliminary results, whose intent is to inform user about approximate execution time.

Another option is to generate a code that performs measurement and calls measured method directly. We need to use reflection again to generate this code, but the method call does not incur any performance overhead when compared to the reflection call. Nevertheless, as we need to inspect the measured method and its generator, create some source files, compile them, run the compiled classes and finally collect results, this whole process lasts longer than a simple reflection call, therefore is this process not suitable for the preliminary results as they need to be measured quickly, but for the most accurate ones.

The last option we considered usable was to measure the performance with another benchmarking tool. The tool we found for this most suitable was JMH (see section 2.7.1) because of its extremely precise results. However, the process of preparing data for the JMH tool together with running it and collecting results, which would need to be transformed a little, was in our opinion too difficult. Therefore, instead of implementing this, we decided to create interface for all classes attempting to measure the method performance. This interface is then one

of the points, where one can extend our application by writing other measurers.

2.5.2 Basic measuring issues

While measuring time in Java, the first measurement iterations have higher execution time than the rest of them. The main reasons for this are usually the Java lazy class loading mechanism (loading binary), possible initialization of JVM, filling cache or paging. Another important factor causing the first iterations being slower is JIT, which until realizing that the method is called often, may compile the method using weak optimizations (or even let the method being interpreted - see section 2.1.2). To illustrate this problem, we performed several measurements with DaCapo (*9.12-bach*) benchmark suite [1], where we measured thousand times its *fop* benchmark. In order to enforce stable GC behavior, following command was used:

```
java -XX:ParallelGCThreads=1 -Xmx640m -Xms640m -XX:NewSize=192m
    -XX:MaxNewSize=192m -XX:-UsePSAdaptiveSurvivorSizePolicy -XX:SurvivorRatio=1
    -XX:MaxTenuringThreshold=4 -XX:InitialTenuringThreshold=4
    -jar dacapo-9.12-bach.jar fop -n 1000
```

The measurement generated 1000 measurements, out of which we plotted (in Figure 2.2) first 100 measurements to show the slower performance at the first measurement indices. We can see that the first measurement has about four times higher execution time than the rest of the measurements.

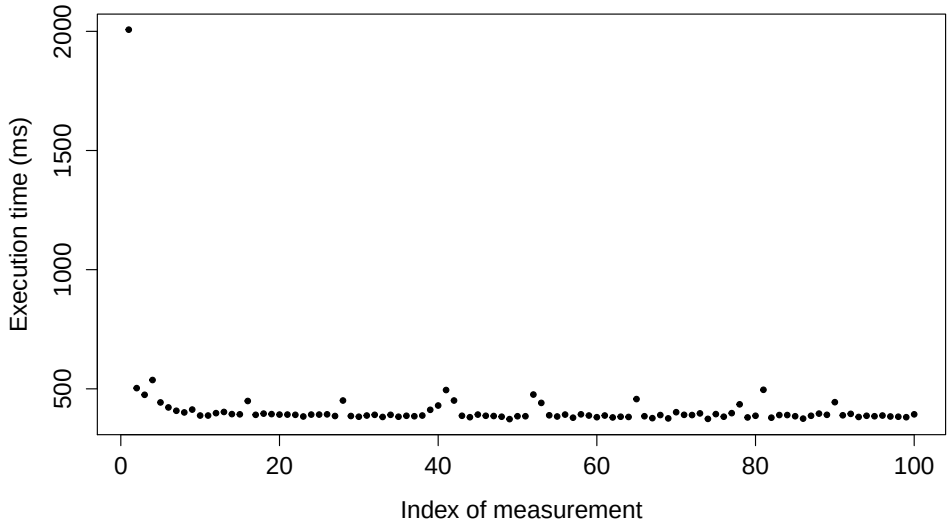


Figure 2.2: Performance overhead at first measurement iterations

The Figure 2.2 also shows that the method seems to be stable after twenty measurements, after which the execution time seems to be quite the same (stable). When we plotted all thousand measurements into a graph, it showed that the estimate about first twenty measurements holds, therefore we can be reasonably sure that the method execution time will not change anymore.

Because the startup performance is not usually critical (or typical), we decided to discard first values (so called warmup) of all measurements. The amount of non-measured warmup measurements can be determined in many different ways, from which we chose the number of warmup measurements (user specifies, how many measurements should be considered warmup), and the time to spent with warmup measurements (user specifies, what time should the program spend at warmup).

Another benchmarking issue is the dead code elimination (see section 2.1.2). This JIT optimization may eliminate out benchmarking code (code of the measured method) having no side effects (i.e. the code, whose result is never used). Therefore, the programmers writing methods to be measured must write them to avoid this optimization (mostly compute a result from problematic parts and return it), while the benchmarking tool must make some computations with the returned result, so that whole call is not optimized out.

We already know that the first measurement iterations have higher execution time, however, practically any measurement iteration may get interrupted either by JVM process (e.g. by garbage collector) or OS process, making the execution time of current measurement iteration higher. The remedy for this is not easy. We may suggest JVM to run garbage collection before every measurement to decrease the chances of running garbage collection in it, as well as yield the current thread before every measurement in order to increase a chance of method not being interrupted, but these methods do neither ensure that the garbage collection will not really run nor that there will be no OS process distorting our measurement. Both these methods have also disadvantages in the possible flush of caches, which makes the next measurement slower (or even higher) than expected. Therefore, even when running these methods, we must count that there will be distorted measurements, which must be somehow handled.

We would like to remove distorted measurements. When doing so, we must also notice that the method execution time may vary from run to run due to different workloads prepared by the workload generator, because the generator programmer may for example decide to prepare random values (e.g. when preparing collection to be sorted). With these assumptions, it is already extremely difficult to locate and remove distorted measurements. Therefore, we will use statistical methods to remove at least major outliers, which would affect the measurement results the most. We will also make measurement results available for user, so that he can process them himself if needed.

2.5.3 Statistical results processing

The measuring server performs multiple measurements for each point of measurement. These measurements are then processed to remove outliers and now we need to compute its statistical characteristics that can be shown in a graph as well as in a table. The ideal situation would be, when the measurements belong to some well-known distribution as for example to the normal one. However, because every measurement may take another time due to another workload prepared by the workload generator, we cannot suppose any of them. Therefore, we must compute statistical characteristics independent on the underlying distribution. These characteristics should give user an overview about what is the estimated

method execution time. We can generally report two things: single numbers and intervals.

The most common single number characteristics are measures of central tendency: mean, median and mode. While the mode may be useful when working with categorical data, it loses information when working with continuous data. The arithmetic mean then reports the expected value of the measurement; therefore we chose it for our measurement characteristic. Because not all measurement outliers may be detected and the arithmetic mean is highly susceptible to them, we decided to report also median, which may report the expected value better in measurements influenced by outliers. Because both mean and median do not reflect how much the measurements are dispersed, we need to report standard deviation, which quantifies the amount of dispersion in the set of measurements, too.

We often need to determine intervals in which the measurements lie with the high probability. We decided to use the first and the third quartile to determine the boundaries of an interval, which means that at least 50 percent of all measurements lie within this interval.

2.5.4 Measurement quality

As already described in section 2.4.5, the client sends with every request the priority saying, how accurate results he wants. We also decided to set the semantic value of the priority on the server side. Let us now discuss, what defines the accuracy of the measurement.

From the section 2.5.1 we know that we may measure either using reflection or using direct call. The reflection is good for approximate results, which are results requested by lower priority. The higher priority request must be then processed using direct call, which provides much more precise results.

We also discussed (see section 2.5.2) that every measurement should have a warmup phase. The longer the warmup phase, the greater probability we have that the measurement already reached the stable state. Quite similar quality indicator is the number of performed measurements, because naturally the more measurements we have, the more sure we can be that the results are correct. It is also easier to locate and remove outliers in a huge data set than in a small one.

Finally, we must remind that we are measuring intervals, which means that we have to choose points from it, in which the measurements will be performed. The number of selected points is the last indicator of measurement quality used in the project.

2.5.5 Result caching

The server needs to cache the measured results, so that the same measurements will not be performed multiple times, and also to enable user to browse through them. The cache should be persistent as well as consistent, which leads to the use of database.

Although we were thinking about using an ordinary database such as MySQL or PostgreSQL, we decided to use embedded Derby database. The embedded database engine runs in the same JVM as the measuring server, which makes

the database another part of the application just like any other library used by the application. The main advantage of this approach is the full program control of the database together with no need to run new process and manage the communication. The database may, however, be little slower, because the maintaining processes must be done only when user calls database functions , while the ordinary database could have for this purposes their own process. The performance issue occurs especially when processing large amount of data. Nevertheless, we decided to select just a sufficiently small representative subset of all measurements, which will be stored in the database.

Derby has full support of SQL92 and SQL99 standards, has a small footprint and is easy to use, which was a reason, why we decided to use this database. The database engine can be, due to JDBC (Java database connectivity technology) interface, easily changed in the future by simple rewriting database connection strings and the definition of auto increment keys, which is in the Derby database implemented in a non-standard way.

2.6 Communication protocol

The client needs to send to the server information about the requested measurement, which is the measured method together with its generator and generator's arguments, the priority number (see section 2.4.5) and the identifier of the user (see section 2.5). The server needs to send to the client the measured results, their priority number (it may happen that the server found in the cache more accurate results than required), units, in which the results are sent, and if any error occurred during the measurement, then also an error message.

The client uses AJAX to send the request to the server. Therefore, it would make sense to use XML as the communication protocol language. However, because we are sending just simple data that do not benefit from XML complexity (e.g. do not need any validation, namespaces or attributes), this language may seem to be too heavyweight as it needs a lot of tags, which make it also harder to process. This was the reason, why we chose JSON (JavaScript Object Notation), which is a light-weight interchange format easy to use with JavaScript. Its main advantage over XML is its simplicity, which makes it easier to parse and also to read (see Listing 4 and Listing 5).

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<request>
  <method>...</method>
  <generator>...</generator>
  <genenArgs>...</genenArgs>
  <priority>...</priority>
  <id>...</id>
</request>
```

JSON

```
{
  "method": "...",
  "generator": "...",
  "generatorArgs": "...",
  "priority": ...,
  "id": "..."
}
```

Listing 4: Comparison of XML and JSON complexity at the request

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <results>
    <result>...</result>
    ...
    <result>...</result>
  </results>
  <units>...</units>
  <priority>...</priority>
</response>
```

JSON

```
{
  "results":[...],
  "units":"...",
  "priority":...
}
```

Listing 5: Comparison of XML and JSON complexity at the response

2.7 Comparison with similar products

We did not find any project that would add a performance extension directly into Javadoc, which makes our tool in this approach unique. However, there are micro-benchmarking tools, which have an IDE support for the performance measuring as well as tools, which perform measurement with simple console interface. We chose three tools that in our opinion everyone, who wants to write any benchmark, should get familiar with.

2.7.1 JMH

JMH [3] is a micro-benchmarking harness written by the Oracle's Hotspot developers. Among other things, these developers were responsible for JIT compiler, therefore, they were able to avoid all benchmarking pitfalls making the tool highly accurate [10], which is definitely one of its main advantages over other projects. To measure the method performance with this tool, you just need to setup a maven project using a command obtained from the JMH official sites and annotate the methods, which performance you wish to measure, with an appropriate annotation. The annotations are in the tool highly utilized. We can use them for example to set the methods to prepare and reset the state objects during the lifetime of benchmark (so called fixtures), to set the measuring mode (e.g. measuring throughput or average time), or the number of warmup and measurement cycles. The suggested approach of how to run benchmarks is using the command line, however, there are also multiple IDE supported ("Running benchmarks from the IDE is generally not recommended due to generally uncontrolled environment in which the benchmarks run [3].")

In terms of our project, we have to mention its *Blackhole* class, which one can use to consume the values, which may be otherwise optimized out due to the dead code elimination. Because of the elegant class implementation (the consume method must last the shortest time possible) and the similar code license, we decided to use this class also in our project.

The last thing that needs to be mentioned are the amazing hints [5], how to write the code to be measured so that we really measure, what we intended. These hints contain many examples illustrating the common flaws that would we otherwise consider being correct measurement code.

2.7.2 JunitPerf

"JUnitPerf is a collection of JUnit test decorators used to measure the performance and scalability of functionality contained within existing JUnit tests [2]." The tool allows us to add performance tests to an existing JUnit test without changing the functionality of the JUnit test. The tool provides two test decorators (TimedTest and LoadTest), whose intent is to measure the execution time of the particular method, which will be in the case of LoadTest done under several concurrent users and iterations. The typical use case of this tool is in the situation, when we use JUnit framework to validate the methods and we need to measure, whether these methods execute within some time frame. This approach shares with JUnit tests the huge advantage in the possibility to rerun tests after refactoring the methods.

2.7.3 SPL tools

"Stochastic Performance Logic (SPL) serves for capturing performance assumptions [12]." The tool uses annotations to mark tested methods. The annotation contains formula for assumption about the method, which may for example be the assumption that the current method runs at least three times quicker than array sorting. These assumptions are then evaluated as the part of the build process. The tool uses well-defined logic, statistical sound testing, enables automatic evaluation and has also the support for different method versions, which means that we can evaluate, whether the current method version is faster than the previous one. The code design of this tool brings also the separation of workload generators, which is quite similar to ours (see section 2.2).

2.7.4 Summary

All of these tools are suitable for another use. The main advantage of our project is the integration with Javadoc, which does not force programmers to use other environment than the documentation, which they usually use when studying new methods. Our tool also enables an easy measurement of interval things with an easy to read output, which is in the other tools a little bit more complicated.

3. Programming documentation

In this chapter, we will introduce the structure of the program. After that, we will describe the communication protocol between frontend and backend part of the project, which will be then described, too. In the end of this chapter, we will talk about project support for possible extensions.

3.1 Project structure

The project structure can be best demonstrated with the folders storing main parts of the project.

config/ Stores configuration files for the measuring server

lib/ Contains external libraries used in the project

out/ All generated files (e.g. compiled classes, generated documentation or JARs) will be placed here

src/

- **java-demo/** Demo examples of the framework usage
- **java-doclet/** Source files of the doclet
- **java-server/** Source files of the measuring server
- **java-shared/** Source files shared between doclet and server (e.g. annotations)

build.xml Build file of the application

3.2 Communication protocol

We already decided to use JSON as the format for the communication protocol 2.6. This means that the communication will be text-based and because we do not need to protect transmitted data, also unencrypted. We will now describe the attributes and their values, at first in the measuring request and then in the response.

The measuring request contains these attributes with their values:

testedMethod Encodes the measured method in format: *package#class#method#@param1@param2*, where this sequence may end with another *#number*, which is just a facilitation of the doclet work with no real meaning.

generator Encodes workload generator and its values have the same format as those of testedMethod attribute.

rangeValues Tells, against which of the generator's arguments the measurement will be performed.

priority Describes the quality of the requested results (see section 2.4.5).

id Identifier of the user to help server distinguish clients, so that the server performs only one measurement per user (see section 2.5).

data JSON array containing selected values saved as ordinary Strings with the range value in format *"lowerBound to upperBound"*.

```
{
  "testedMethod": "example001#MyArrayListMoreOps#doMultiple#@cz.cuni.mff.d3s.
      tools.perfdoc.blackhole.Blackhole@int@int@int@int#0",
  "generator": "example001#MyAListGenerator#prepareDataMultiple#@cz.cuni.mff.d3s
      .tools.perfdoc.workloads.Workload@cz.cuni.mff.d3s.tools.perfdoc.workloads.
      ServiceWorkload@int@int@int#0",
  "rangeValue": 0,
  "priority": 1,
  "id": "353xqgs0pb9",
  "data": [
    "10716 to 74753",
    "409",
    "149"
  ]
}
```

method: doMultiple(Blackhole, int, int, int, int) in example001.MyArrayListMoreOps class
generator: prepareDataMultiple(Workload,int,int,int) in example001.MyAListGenerator

Figure 3.1: The sample measure request

The response then consists of these attributes and their values:

data Contains measured results as the array of arrays, where each of the inner arrays contain the value in which the measurement was performed together with two arrays in format *[mean - standardDeviation, mean, mean + standardDeviation]* and *[firstQuartile, median, thirdQuartile]*. This format was chosen to facilitate the parsing process of the doclet, because the chosen value format may now be unchanged passed to the graph as new data.

units Tells the units of measured data.

priority Informs client about the priority of sent results.

```
{
  "data":[[103,[47,102,157],[85,85,86]],[247,[-139,459,1057],[340,342,384]]],
  "units":"ms",
  "priority":2
}
```

Response contains two points, in which the measurement was performed: 103 and 247. The mean in the first point was 102ms, the standard deviation (102-47)=55ms, the first quartile and median 85ms and the third quartile 86ms. The priority of results is 2, which means that the next request will continue with priority 3.

Figure 3.2: The sample measure response

3.3 Doclet

In the frontend analysis (see section 2.4), we already discussed the basic structure of the performance part, decided to use JavaScript for dynamic page changes and also chose the graph library. This section should then describe, how the generation of performance part works.

3.3.1 Doclet rewriting

The Javadoc standard doclet is not extensible by plugins; therefore, we need to rewrite it in order to add the performance part to it. The standard doclet uses a XML file to define the structure of the generated documentation. This file contains elements for packages, classes, annotations or methods, where for each of these elements, a class containing methods, whose names are derived from the appropriate child element names, is defined. When the doclet is then generating documentation for example for a method, it finds the method element of the XML file, goes through its child elements and for each of them calls (using reflection) the method from the *MethodDocumentation* class, whose name corresponds to the element name. The intent of these methods is to generate the particular part of the documentation into the content passed as one of the arguments. So that the called method knows the structure of the documented method, it can obtain an instance of *MethodDoc* class, which is a class that resembles the classic reflection *Method* representation.

3.3.2 Program flow

As we realized, how to make the doclet engine call our method, we simply added a new element *PerformanceInfo* in the end of the method documentation of the XML file as well as a new method to the *MethodBuilder* class. This method is an entry point to the performance part, whose high level program flow can be seen in Figure 3.3

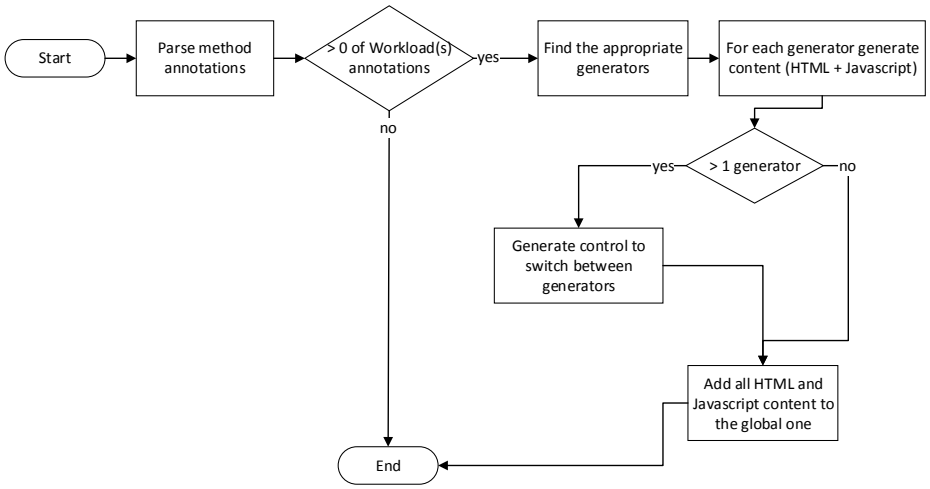


Figure 3.3: Doclet program flow chart

The first thing the program makes is the parsing of the method annotations. If the method does not contain any *Workload* or *Workloads* annotation, the programmer did not intend to add any performance measurement extension to this method; therefore we may leave the method unchanged. Otherwise, we search for the generator classes and for each of them create performance part, which is in case of the first generator visible and for the rest of them hidden. After we generated content for the generators, we may need to create control to switch between them in case that the method has more than one available generator. The last step is then adding of locally created content to the global one.

This was just a high level description of the doclet. We will describe each part of the program flow in the classes, which are responsible for the concrete tasks, in the following sections.

3.3.3 HtmlDocletWriter

We followed the doclet convention, which in this case was that the *MethodBuilder* class, which contains the method starting the performance part, is just delegating the job to another class, which then delegates it on the *HTMLDocletWriter*. This is the first class that actually does something in the context of the performance part and at the same time the last class of the standard doclet that need to be modified (all other performance classes had to be created). The method we are interested in the class is *addPerformanceInfo*, which parses the *Workload*, respectively *Workloads*, annotation of the method, and if finds any of them, then calls *PerformanceWriter* to create performance part with these generators.

3.3.4 PerformanceWriter

This class dynamically loads the generator classes using the *ClassParser* class. The path, where these classes are searched is passed to the doclet using *workloadPath* argument, and can be then obtained from *DocletArguments* class. Every generator is then passed to *PerformanceBodyWriter*, which generates the content specific for each generator, where each generator's content is wrapped in the div tag having unique id. These contents are then saved in the code sequentially, while the first of them is made visible and the rest of them are hidden. If there are multiple generators available for the current method, this class then prepares select control to switch between generators. The switch simply hides currently showed generator's div, while making the newly selected one visible.

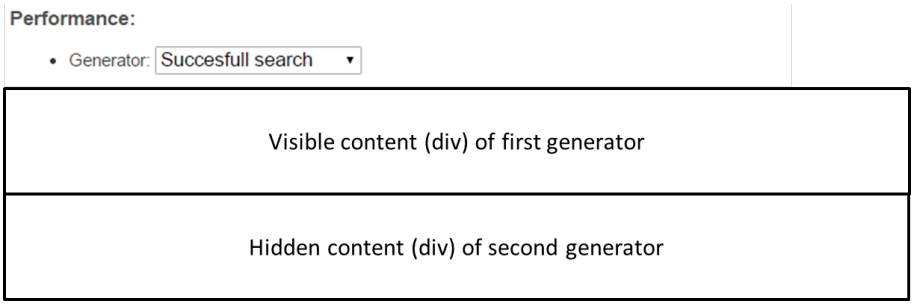


Figure 3.4: Content generated by PerformanceWriter for method having two generators

3.3.5 PerformanceBodyWriter

PerformanceBodyWriter is responsible for generating content of the generator. As we already discussed in section 2.4.3, this content can be logically divided into two parts, one for generator description together with controls to set his arguments and submit button to send the request, and another one for the graph and table to present measured results.

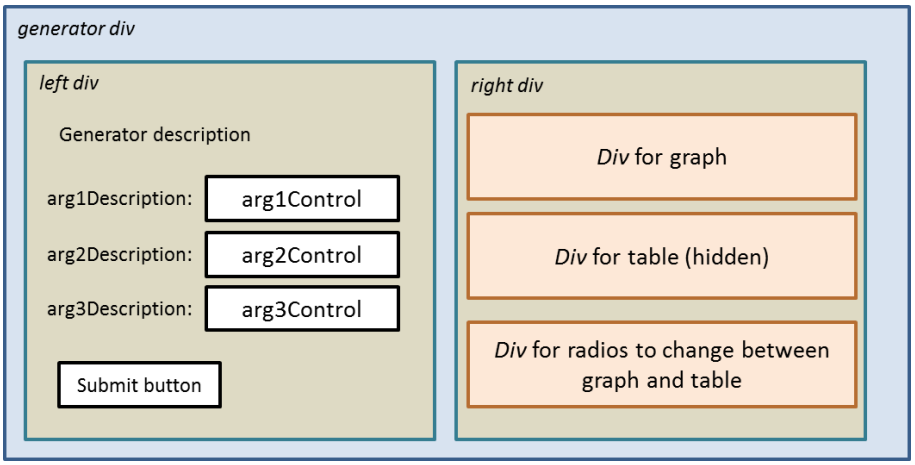


Figure 3.5: Schema of the generator HTML structure

As we can see in Figure 3.5, whole generator content is wrapped in a div element. This element has a unique identifier, from which all identifiers (e.g. identifier of submit button) used in the generator's content are derived. The generator's identifier is generated using *PerformanceNamingUtils* class, which generates unique identifier that tries to describe the generator in a human readable form instead of randomly generated sequences. The main div element then contains two other div elements each for one logical part as mentioned above. Let us first discuss the left part, which contains the generator specific settings.

The first item in the left div element is a generator description, which is obtained from the generator annotation. Then are controls to set generator arguments. These controls are generated from generator's parameters and theirs annotations. The simplest parameter type is String, for which the simple text input is generated. Then we have an enumeration type, which is represented by a select control with all the enumeration values. Finally for the numerical types (e.g. int, float, double), we decided to use JQuery UI slider, which enables users to easily select required values as well as us to restrict the values according to the parameter annotation. The last item in the left part is then a submit button, whose purpose is to send request to the measuring server.

The right div element then contains three other div elements. The first of them is dedicated to the graph, the second of them to the table and the third one contains radio buttons to select, whether we want to see the graph or tabular output.

So far, we discussed only HTML part of the code, but we have to naturally generate JavaScript code too. The code is generated for each slider (*JSSliderWriter*), to validate selected values before sending them to the server (mainly whether we chose one interval – *JSControlWriter*), to send the request to the server (*JSA-*

jaxHandler) and to create graph and table from the results (*successfunction.js*). Except for the creation of graph and table, which is handled by the global method, the code to make the job is created for each new element (generator) and saved into *JavascriptCodeBox*, which is a container for all generated JavaScript code.

3.3.6 Error handling

Programmer may naturally make mistakes when writing methods to be measured or their generators. The common examples may be that we forget to annotate generator or any of its parameters, forget to add *Workload* parameter, or try to use unsupported type of generator parameter. All these mistakes are reflected either in no performance part added to the Javadoc (in case that we found no generator for the method) or in the absence of some of the method generators. We decided to create own exceptions for these cases, so that we may easily recognize and report what went wrong.

3.4 Server

In this chapter we will describe the basic server routine, mention most important classes together with their relations and some other important features.

The measuring server runs on a simple HTTP server (*HttpServer* class), which is in the Java included since version 1.6. This class allows us to use multiple threads stored in a cached thread pool and to create multiple contexts. We define two contexts: one for measurement requests, which are handled by *MeasureRequestHandler* and run on */measurement* context path, and one for cache requests, which are handled by *CacheRequestHandler* and run on */cache* context path. The measurement context is more important as all clients send their measurement request to it; therefore, we will explain it first. The cache context then enables users to browse through cache using web pages and will be discusses after the measurement context.

3.4.1 Measurement request flow

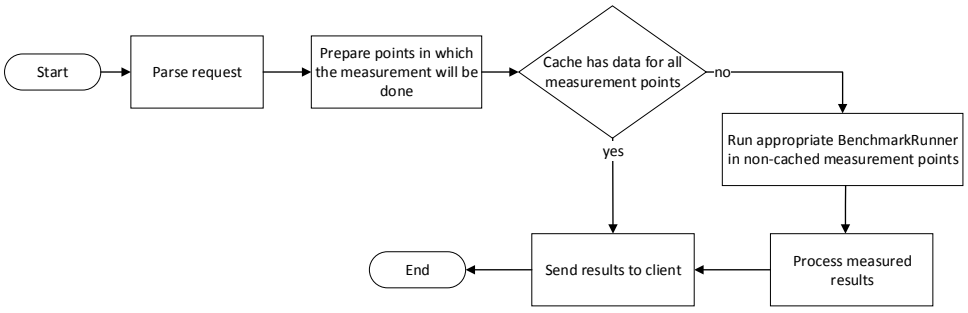


Figure 3.6: Measurement server program flow chart

The measurement flow starts with a parsing of a client request into *MeasureRequest*, which is a data structure that contains all information about the re-

quested measurement. Since we are measuring against a range, the next step is a selection of points from the interval, in which the measurement will be performed. Before doing any measurement, the program checks, whether the cache contains results for all measurement points and if does, the results are immediately sent to the client without any measuring. If there were not found results for all measurement points, we must perform measurements, which are performed by the *BenchmarkRunner* interface implementations selected according to the priority number. The measured results must be then statistically processed in order to remove outliers and compute appropriate characteristics, from which the JSON Object is created and sent to the client.

3.4.2 Data structures

The intent of this section is to provide an overview of the most important data structures used in the server part of the project.

MethodInfo
: This class represents the measured method, respectively generator method, along whole project. It provides methods to get the basic information about represented method as well as to get its format used in a database.

MethodReflectionInfo
: Extension of MethodInfo containing also instance of reflection *Method<?>* of represented method.

MeasurementQuality
: Contains information about requested measurement quality (e.g. how many warmup measurement should be performed).

MeasureRequest
: The request from the client is parsed into this structure, which contains measured method and generator represented by *MethodReflectionInfo* class, index of generator argument against which the measurement will be performed, requested measurement quality and the identification of the client.

BenchmarkSetting
: Binds together all information describing single point of the measurement. This involves the measured method, its generator, the generator parameters and the measurement quality.

3.4.3 HttpMeasureServer class

This class contains main method, which starts the measuring server with settings referenced by mandatory argument *configuration*. This argument points to the directory that contains several configuration files (*Class_classPath.txt*, *measure.properties* and *server.properties*), whose format will be described in the user documentation. The server runs on a specified port with two contexts, which are handled by *MeasureRequestHandler* (*/measure* context) and *ClassSiteHandler* (*/cache* context) classes. This class is also responsible for starting the database that contains cached results as well as for creating *LockHashMapBase* class, which

restricts the number of measurements being performed at the same time for the browser to one.

3.4.4 MeasureRequestHandler class

When the new measurement request comes to the server, a thread from the thread pool is obtained and runs the *handle* method of *MeasureRequestHandler*. This class is responsible for reading the incoming request, which is parsed into *MeasureRequest* and passed to *MethodMeasurer*, which returns JSON Object containing measured results. These results are then sent by this class to the client. Finally, when we were measuring results with the highest priority, we save the results into the database.

3.4.5 MethodMeasurer class

This class prepares points in which the measurement will be performed using *MeasuringUtils*, which tries to prepare points to cover the whole interval equally (by binary dividing). When the points are prepared, cache is searched, whether it contains results for all points, and if does, we may return these results immediately. When not all points have cache records, we must perform measurement in these points. For this purpose we defined interface *BenchmarkRunner*, whose goal is to measure one point of the measurement. We have two *BenchmarkRunner* implementation in the project: *DirectRunner* and *MethodReflectionRunner*, where the first one is used for measurements with the highest priority, while the second one for the rest of them. After we have results for all points of measurement, we can process them (e.g. remove outliers or choose other units) and finally create JSON Object from them, which is then returned.

3.4.6 Measuring

As we mentioned in the previous section, to measure one point of measurement, we defined BenchmarkRunner interface.

```
interface BenchmarkRunner {
  MeasurementStatistics measure(BenchmarkSetting);
}
```

This interface defines one method, which takes *BenchmarkSetting* (see section 3.4.2) as the only argument and should produce *MeasurementStatistics*, which is an implementation of Statistics interface that defines basic statistics method (e.g. mean, median or standard deviation). The *MeasurementStatistics* contains all measured results, from which the characteristics is counted, and allows us to detect and remove outliers too.

While creating methods to be measured, users often need some mechanism to avoid dead code elimination for code with no side effects that cannot be effectively returned. Therefore, we decided to copy JMH (see section 2.7.1) approach, where users may declare the first method parameter *Blackhole*, which is a class having methods that accept all primitive types as well as Object. User than utilize these methods to consume the problematic parts of code. This class is then

not prepared by the workload generator, but by the tool, which uses *BlackholeFactory* to get the singleton instance of *Blackhole*. We decided to create just one instance of *Blackhole* to make the minimal perturbation of the system. Because the programmers of JMH tool made their *Blackhole* fast (as for execution time of consuming methods) as well as efficient (in terms of dead code elimination), we decided to use their implementation in our tool.

3.4.7 MethodReflectionRunner

The basic implementation of *BenchmarkRunner* interface that uses reflection to run the measured method. The class uses *WorkloadImpl* class to get workloads from the generator and *ServiceWorkloadImpl* to provide basic information (for now only the priority of the measurement) to the generator. The warmup and measurement cycle are contained in one method, which just according to one of its arguments decides, whether current run is a warmup, therefore the measured results should be discarded, or already measurement run, which means that the results are saved into *MeasurementStatistics*, which is in the end returned.

3.4.8 DirectRunner

DirectRunner is the second implementation of the *BenchmarkRunner* interface. This class provides results comparable with the results one would normally get when running the method in the program. To reach this, the program generates Java source files that measure the method performance, compiles these classes, runs the code in a separate JVM, collects and returns the measured results.

The first step of the measurement process is the code generation, for which we use templates that are filled in with the data obtained from the methods via classic reflection and then transformed into Java source files using the Apache Velocity tool. We use three template files managed by *CodeGenerator* class:

TGenerator.vm Contains code that calls the generator with given *Workload* and *ServiceWorkload* instances

TMethod.vm Performs direct call of the measured method. This means that the call is done with arguments of correct type and if the method returns something (is not void), the result is saved into public static volatile variable, which prevents JIT compiler from optimizing out the method call.

TMeasurement.vm Contains main method that performs the measurement of the method using two previous classes to call generator, respectively measured method. The code to perform measuring itself is quite similar to the reflection one except for the method call self, where we now use *TMethod* (*TGenerator*) to call method directly. The measured results are then saved into a file.

Generated source files are placed into the folder, whose name is derived from the measured method, generator and its arguments. This folder is then placed to the directory set by *generatedCodeDir* property of the *server.properties* file.

To compile these sources, we use *Compiler* class, which obtains system Java compiler (one of the reasons, why it is not sufficient to use JRE instead of JDK) and compiles classes to the same folder, where the sources are located.

The new JVM is then started to run the *TMeasurement main* method, which measures the performance of the requested method and saves the results into the selected file. These results are then collected by *DirectRunner* class, which stores them in *MeasurementStatistics*. In the end of the process, all generated content is deleted and results returned.

3.4.9 Database

Measured results are due to performance reasons stored in the database, which for this purpose contains three tables:

measurement_information Stores information about a measurement of one point, for which stores its ID, identification of measured method and generator, arguments of generator, aggregate measurement statistics (e.g. mean, median) and ID of measurement quality (foreign key into *measurement_quality* table)

measurement_detailed Every row of this table contains one measured result that corresponds to some measurement stored in *measurement_information* table

measurement_quality Contains information about measurement quality (e.g. warmup_time) of performed measurements

The measuring server classes do not work with the database directly, but utilize interface *ResultCache* (implemented by *ResultDatabaseCache*), which defines two basic database operation:

```
interface ResultCache {
  BenchmarkResult getResult(BenchmarkSetting setting);

  boolean insertResult(BenchmarkResult benResult);
}
```

- *getResult* method gets *BenchmarkSetting* and finds appropriate *BenchmarkResult*, which is a class containing two items: *Statistics* and *BenchmarkSetting*. The database is always searched for at least as precise results as requested, which means that all items of returned *MeasurementQuality* (contained in *BenchmarkSetting*) must be equal or bigger than those of requested *MeasurementQuality*

- *insertResult* method inserts measured results into database, which includes deleting all results having same measured method, generator and generator arguments, but having worse *MeasurementQuality*; inserting one record into *measurement_information* table, inserting at most one record into *measurement_quality* table (depends on, whether measurement with such *MeasurementQuality* was already saved) and finally inserting as many records

as measured results into *measurement_detailed* table. There may be a lot of insertions into *measurement_detailed* table, therefore we use batch insert, which basically performs multiple operations at once.

3.4.10 Cache context

So far we were describing the */measure* context, which is responsible for correct measuring. However, our tool provides also */cache* context, with web pages allowing users to browse the cached results. This context is handled by *ClassSiteHandler*, which according to the URL decides, what *SiteHandler* to use to process the page.

```
interface SiteHandler {
  void handle(HttpExchange exchange, ResultCacheForWeb res);
}
```

The first of the *handle* method parameters is the *HttpExchange*, which is mainly used for investigating URL address that is utilized for passing information between web pages. The second parameter is then another interface defining operations with database, this time more oriented to retrieving database data.

Each web page is maintained by one class implementing *SiteHandler* interface. Every class has its own template, which is using Apache Velocity tool transformed into requested web page. When the template is transformed, *HttpExchangeUtils* is used to send the requested web page to the client and close the exchange.

3.4.11 Logging

The measuring server uses standard Java logging mechanism to log events on the server. Every class obtains an instance of *Logger* using the name of its class and use this *Logger* to log important events. We decided to use these three logging levels to distinguish the relevance of the events:

- SEVERE: When there is any event causing the current measurement to fail (e.g. when the requested class could not be found on the server)
- INFO: To log things, the server administrator (e.g. the one looking on the standard program output) wants to see to have the basic idea, what is currently happening on the server (e.g. new measurement request was accepted)
- CONFIG: To log more detailed things that the programmer wants to see when looking in the program log (e.g. insert of a new result in the database).

The logging levels were chosen with the respect to the logging levels of standard Java libraries (e.g. used *HttpServer* class) and external libraries.

3.5 Extending

The tool is written in a way that does not prevent programmers from adding other extensions. Our original aim was to add the performance extension only to the Javadoc; and we ultimately designed the communication protocol with respect to the possible client replacement.

As for the measuring server, the information regarding the specific source of the request is rendered unimportant, and therefore multiple distinct clients implementing the described communication protocol may be active simultaneously. However, since our aim was to maintain a relative simplicity of the Javadoc code, no direct extensions of the Javadoc are supported. As a consequence, any changes, such as the graph library replacement, will inevitably lead to slight changes of the code.

Due to the *BenchmarkRunner* interface, the measuring mechanism of the server is extensible in two ways. The first consists of creating other classes used to measure the execution time of the method. This has been previously discussed in section 2.5.1 which includes the issue of integrating the JMH tool into our application. The second consists of changing the performance metric from execution time to, for example, memory consumption. The basic version of this particular metric may be rather easy to implement when using the current *BenchmarkRunner* interface, where a memory state can be measured instead of measuring time.

Finally, when the users need to change the cache storing the results, they can do so either by creating new implementation of *ResultCache* interface or, in case they only want to change the database, by changing the database connection string and replacing the database binary files.

4. User documentation

This chapter describes, how to download and install the tool, demonstrates its functionality on demo examples, shows the way to run the modified doclet, the measuring server and configure the measuring server for different quality of results and finally shows a simple method together with its workload generator.

4.1 Requirements

To compile and run the framework, the following software is required:

- Java SDK >= 1.7.0
- Apache Ant
- Web browser supporting JQuery >= 2.0

4.2 Download and installation

The tool can be found on the enclosed CD (see Appendix A) or be downloaded from `https://github.com/arahusky/performance_javadoc` as a zip archive. This archive contains one folder, whose structure was already described in section 3.1. The project can be built using Apache Ant, therefore, the folder contains *build.xml* file with some predefined tasks, from whose these are important:

main compiles all sources (into *out/classes folder*), creates project documentation (in *out/javadoc*) and two JAR files (*out/jars*), which will be described later

run-server runs the measuring server

run-demo runs our doclet on the demo examples (the generated documentation is located in *out/demo*)

clean removes all content generated by the previous tasks (and possible server run)

4.3 Demo examples

The functionality of the tool can be best demonstrated on examples. For this purpose, we created several classes that contain methods to be measured as well as their generators. These classes are saved in *src/java-demo* folder and their packages start at *example001* and end with *example005*. To get the basic idea, of what is the tool capable of, we suggest you to start at *example001* package, which shows the tool functionality used on collections (e.g. how long does it take to sort collection containing certain amount of items). To create documentation from these classes using our doclet, you can run *ant run-demo*, which generates the Javadoc documentation into *out/demo* folder. The doclet basically adds

a performance part to methods marked with Workload annotation, which are in this case all methods. After opening *index.html* file and selecting documentation of *MyArrayList* class one can see method *contains* as in Figure 4.1.

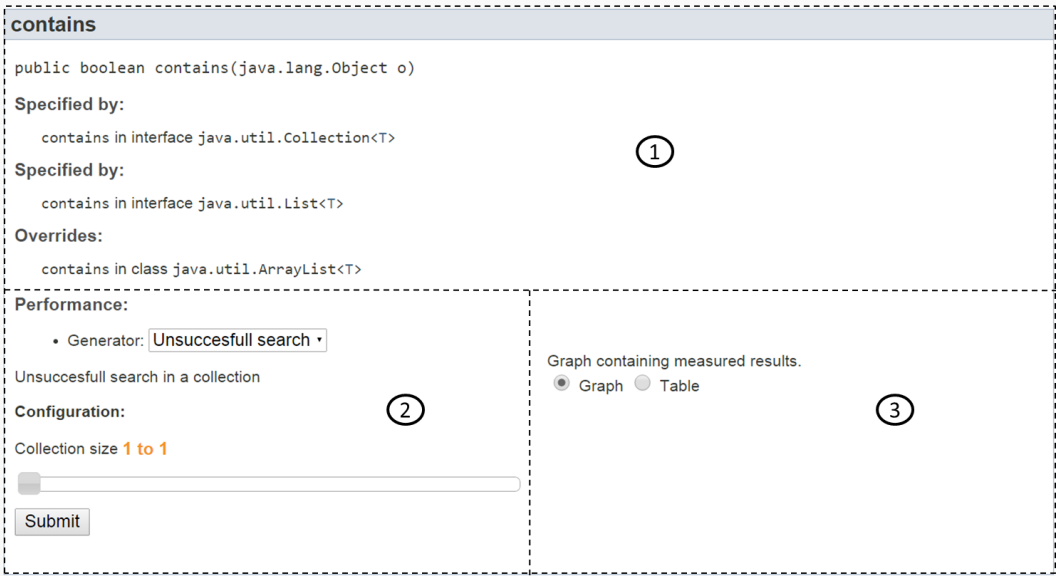


Figure 4.1: Detail of ArrayList<T>.contains method with performance extension

The method documentation now consists of three sections. The first one is the unchanged part of the standard doclet. The second and the third part are then the performance extension, where the second part serves for setting arguments to the particular generator (selected by the select control) of the method and the third part is responsible for showing the measured results.

The Javadoc sends their request to a measuring server; therefore, we must start it before sending any request (otherwise an alert informing us about an error shows). To do so, just run ant task *run-server*, which makes the job for you. If nothing goes wrong, you should have an output informing you about the successful server start as can be seen in Figure 4.2.

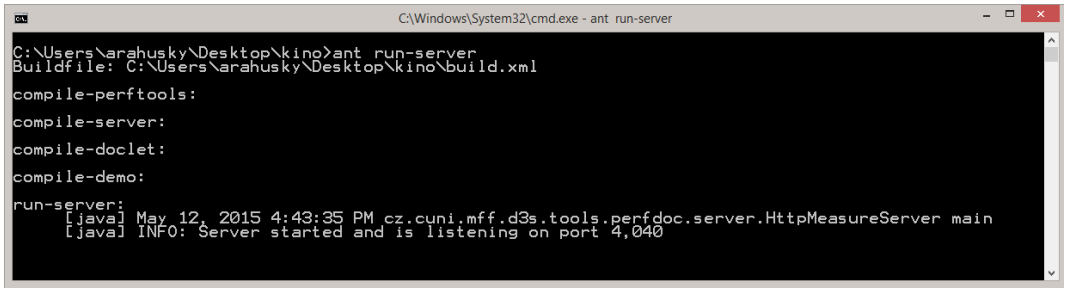


Figure 4.2: Successful start of the server

As the server runs, we can try to measure the performance of the method. This process generally consists of three steps:

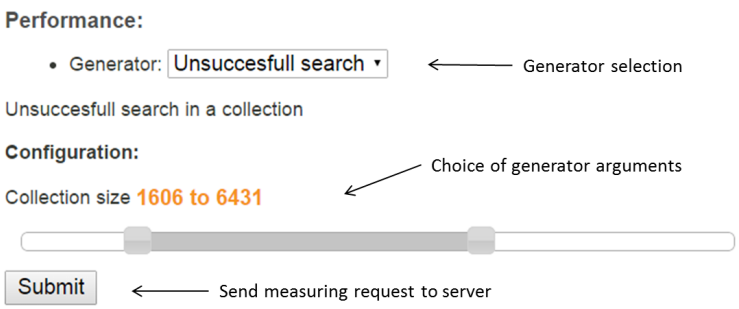


Figure 4.3: The part of the performance extension to set the generator and its arguments

1. Generator selection: if the method has more than one generator available, we can choose, which of them to use. In this case, we may use either *Successful search* or *Unsuccessful search*, where the first generator prepares an element, which is always found by the *contains* method called on the prepared collection, and the second generator then an element, which is not a member of the collection, therefore never found.

2. Arguments selection: we choose the arguments, with which the generator prepares corresponding workload for the measured method. In this case, we may choose the size of the collection, on which the method contains will be called. When choosing the arguments, exactly one of them must be set as an interval argument (a range must be showed on a slider). The measurement will be then performed against this argument.

3. Request send: after we set all arguments with exactly one range, we can send the request to the server. The message informing us about the successful sending of the request is then displayed in the third part.

When the first results are received, the graph is shown in the third part. The lines in the graph are dashed and thin to show that the results may be imprecise. From the moment, when the first results are shown, three more results improvement will be done (every improvement makes the line thicker). The last results, which should be the most precise, are shown with the thick and more importantly solid line.

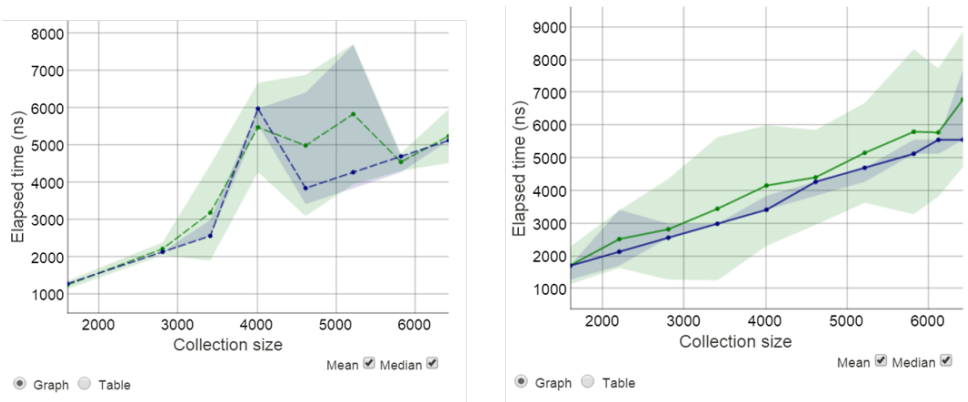


Figure 4.4: Comparison of the third and the fourth (final) results

The graph now shows the relation between the collection size and the time it takes to find out, that the collection does not contain some element. There are two lines in the graph connecting the green and blue points. The green points stand for the mean of the method execution times in the particular point (there were multiple measurements for each point) and the blue points then for the median. There is also green area around the green line, which shows the standard deviation of measurements in the particular point, and the blue area, which shows the interval between the first and the third quartile in the particular point. Both of these lines may be disabled using the checkbox under the graph.

The graph also provides interaction features. One of them is zooming, which can be easily done using the common click and drag mechanism. To restore default zoom level, double click wherever in the graph. If you want to move the displayed (zoomed) area, use the shift-click-drag combination.

Apart from the graph results also table results are generated. To see these results, you must switch the radio button in the bottom of the third part to table (see Figure 4.5). As the graph is great to show tendencies, table enables user to read results for specific points more easily.

Collection size	Elapsed time mean (ns)	Std. deviation	Q1, Q2, Q3 (ns)
1606	1718	571	1283,1710,1711
2209	2524	878	1711,2138,3421
2812	2825	1538	2566,2566,2993
3415	3451	2187	2993,2994,2994
4018	4161	1841	3421,3421,3849
4621	4411	1450	3849,4276,4277
5224	5160	1522	4277,4704,4705
5827	5805	2518	5132,5132,5560
6129	5785	1957	5132,5559,5560
6431	6793	2063	5559,5560,7697

Graph Table

Results contain information about size of the measured collection, mean time, standard deviation, first, second (median) and third quartile of measured times

Figure 4.5: Measured results in the table form

4.4 Doclet

As we already mentioned, the Javadoc documentation with the performance extension is generated by our doclet. You can use ant *main* task (or *create-doclet-jar*) task to create JAR file (*out/jars/doclet.jar*) that will contain all classes the doclet needs to the proper run.

The doclet adds two new arguments to the javadoc command, which are the *serveraddress* and *workloadpath*. The first of them tells the generated documentation, what is the measuring server location, and must be set in the case that the measuring server is not located on the same computer, or if is listening on the port different from 4040. The second of them then tells the doclet, where the classes containing workload generators (separated by appropriate path separator) are located.

The sample run of javadoc command using the doclet JAR file may look like this:

```
javadoc -doclet cz.cuni.mff.d3s.tools.perfdoc.doclets.standard.Standard
        -docletpath path/to/doclet.jar
        -serveraddress http://localhost:4040
        -workloadpath path/to/first/generator;path/to/second/generator
        documentatedClass.java
```

4.5 Server

There are two recommended ways to run the measuring server. The simplest way was already described in section 4.3 and the second way is using the *server.jar*. This file can be generated either using ant *main* task or *create-server-jar* task. The generated JAR file is then located in *out/jars* folder.

The run the server, we must set its mandatory argument *configuration*, which specifies the folder, where the configuration files are. There are three configuration files that must be in the right format before running the server:

- Class_ classPath.txt: every line of this file contains one path to the directory or JAR, where the measured methods, respectively their generators, are
- measure.properties: serves for customizing measurement quality and will be deeply described in the next section (default settings work reasonably in most situations)
- server.properties: contains properties to set basic server configuration (e.g. running port)

The server supports two optional arguments: *empty* and *port*. The first of them is used to tell the server to empty all caches and the second one then specifies the port number, on which the server listens (this argument has higher priority than one set in the *server.properties* file). The sample run of the server using the *server.jar* looks like this:

```
java -jar path/to/server.jar -configuration path/to/config [-empty] [-port 4040]
```

4.6 Measurement quality

As we already described in section 4.3, the Javadoc sends four measurement requests to the server. All requests contain same information except for the priority number, which starts at 1 and is with every other request incremented by 1. Its intent is to tell the server, how quickly should the server perform measurement. The semantic value of the priority number is, however, left on the server and may be set in *measure.properties* file (default location in *config* folder). This file contains several things that can be changed.

First of them is a boolean flag, whether to use code generation for the measurement with highest priority number. The code generation tends to achieve much more precise results than the reflection, which is used for lower priorities; however, it requires creating and deleting files on the disk as well as running

a new JVM. If any of these things may cause a problem, set this property to false, otherwise left on true.

Second of them (*priorityXXXNumberOfPoints*) are four properties determining, how many points will be chosen from the selected interval to be measured for the given priority.

Finally, there are 16 properties restricting time spent while measuring one point. The first 8 of them restricts the maximal number of measurements (*priorityXXXNumberOfMeasurementsWarmup*), respectively maximal time (*priorityXXXElapsedTimeWarmup*), spent with warm-up measurements of one point. The second 8 of them restricts the same thing, but for the measurement self (*priorityXXXNumberOfMeasurements* and *priorityXXXElapsedTimeMeasurement*).

4.7 Viewing cache

The server caches all measurements, so that it does not have to perform same measurements again. These caches are accessible through web interface, which can be accessed at the same address and port as the measuring server is running, on the context */cache*. The address will then look like: *http://localhost:4040/cache*, when the server is running on the same machine as we are accessing its caches.

The first page of the cache shows all classes, whose methods were measured. When clicking on any of them, we may see all methods of selected class that have any record in the cache. Because every method may have been measured using several generators, next page shows all generators, with which the selected method was measured. Finally, next page shows concrete measurement results. The table with results has in its last column button, which when clicked shows another window containing concrete values, from which the mean, median and another statistical characteristics were calculated in the form of graph as well as in the raw textual form in case that we wanted to process these results ourselves.

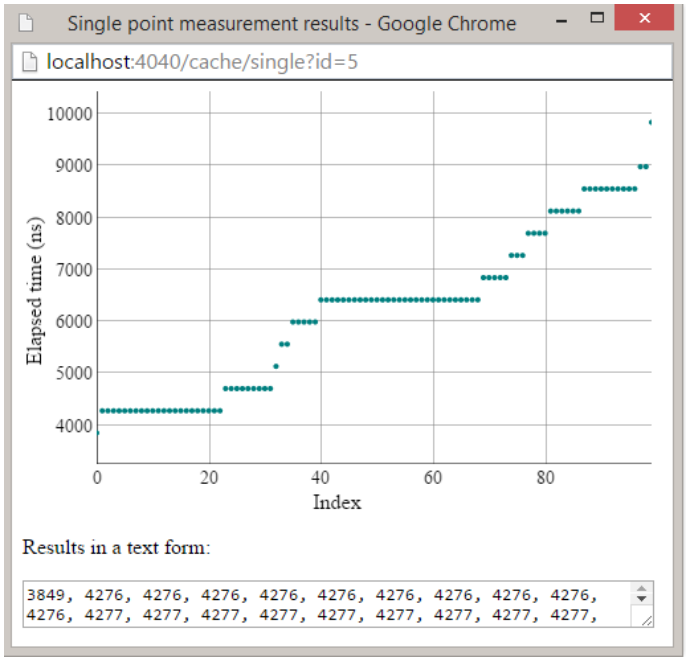


Figure 4.6: All measured results for a single point of a measurement

4.8 Writing own methods and generators

There are two things programmer must do, when he decides to enable users to measure his method using the Javadoc performance extension. The first thing is annotating his method with *Workload* annotation, in which he specifies the generator in format *package.class#methodName*. All annotations are located in *doclet.jar* file, which must be therefore on the class path, when compiling the code. In case that the programmer wants to write multiple generators, he should annotate the method with *Workloads* annotation, which is just an array of *Workload* annotation. The second thing to do is then the writing of the generator.

4.8.1 Generator

Generator is a method, whose purpose is to prepare instance and arguments for the measured method. Generator must also ensure the cleaning of environment after the measurement. This means that if the generator or measured method creates anything (e.g. files) on the disk, these should be by the methods set in the generator removed after the measurement.

To distinguish generator method from the ordinary method, developer must annotate it with *Generator* annotation, which contains the description of the generator and its name, where both of them will be displayed in the Javadoc.

Another important thing about generator is its parameters, from which the user controls, allowing user to select values for the generator, will be created. This rule does not hold for the first two parameters, which are mandatory and have a special meaning:

- The first parameter is interface *Workload*. This interface defines method *addCall(Object o, Object... args)* that prepares one call of the measured method, which could be using reflection performed like *method.invoke(o, args)*. The second defined method is then *setHooks(Object obj)*, which serves for saving class containing methods, which remove generated content, and will be described later.

- The second parameter is interface *ServiceWorkload*, whose purpose is to pass additional information from the measuring server to the generator. This interface currently contains one method *getPriority()* that the generator may use to obtain the priority with which the measurement is being performed.

The rest of parameters must be of following types: numerical (int, float and double), String and any Enum class. This limitation is due to HTML representation and is also associated with parameters' annotations. Every such defined non-numerical parameter must be annotated with *ParamDesc* annotation, which contains the description of the parameter, the numerical parameters are then annotated with *ParamNum* annotation that can be seen in Listing 6.

As we already know, how to mark the generator and declare its argument, we may write the body of the generator, in which we prepare the workloads for the measuring server. As we already denoted, we may set methods to clean the generated content using *setHooks* method. This method takes an instance

```
public @interface ParamNum {
 //description of the parameter displayed in Javadoc
  String description();
  //minimal value this parameter can have
  double min;
  //maximal value
  double max();
  //minimal distance between two possible parameter values
  double step() default 1;
  //whether it makes sense to use this parameter as an interval against which
      the measurement will be performed
  boolean axis() default true;
}
```

Listing 6: ParamNum annotation

of the class, which contains the cleaning methods. There are two kinds of these cleaning methods, where the class may contain at most one of each. The first one is method annotated with *AfterMeasurement* annotation and has the same parameters as the measured method. This method is then called after every call of measured method with the same arguments as the measured method. The second of them is annotated with *AfterBenchmark* annotation, must be parameter-less and is called after all workloads have been called.

4.8.2 Example usage

To show the process of preparing the measured method with the generator, we will show you a model example. Let's say, we have a method that takes an array of bytes, which should be written into a file using the file stream, and we would like to know its performance with the growing size of the array. The first step is annotating the method with the textual representation of generator, where the generator is now contained in method *prepareStream* of *example.FileGenerator* class as can be seen in Listing 7.

```
@Workload("example.FileGenerator#prepareStream")
public static void writeAll(FileOutputStream fs, byte[] array) throws
    IOException {
  fs.write(array); //writes array into given stream
  fs.flush(); //flushes the stream
}
```

Listing 7: Preparing the method to be measured

The generator's objective is to allow user to select the size of an array to be written and according to the size, prepare appropriate array together with an empty file and the stream, using which the array will be written in the file (see Listing 8). After the method is called, stream should be closed. In the end we must delete the file, too (see Listing 9).

```
@Generator(description = "Prepares file stream and array of given size to be
    written into it.", name = "WriteGen")
public void prepareStreamToWrite(
  Workload workload, ServiceWorkload service, //mandatory parameters
  //we allow users to select size from 1 to 10000 with the step of 1
  @ParamNum(description = "Size of array to be written", min = 1, max = 100000,
      step = 1) int arraySize
) throws IOException {
  //the file into which the stream will be pointed
  file.createNewFile();
  //creating array of given size filled with random bytes
  Random r = new Random();
  byte[] array = new byte[arraySize];
  r.nextBytes(array);
  //adding new workload (null because of static method)
  workload.addCall(null, new Object[]{new FileOutputStream(file), array});
  //the hooks are located in MyHooks class
  workload.setHooks(new MyHooks(file));
}
```

Listing 8: Generator for the measured method

```
public class MyHooks {
  //file to be deleted
  private final File file;
  public MyHooks(File file) {
    this.file = file;
  }

  @AfterMeasurement
  public void destroy(Object instance, Object[] objs) {
  try {
    //the first argument is the stream
    (Closeable) objs[0]).close();
  } catch (IOException ex) {
      //there is not much we can do
    System.err.println("Unable to close file stream.");
  }}

  @AfterBenchmark
  public void destroy() {
    file.delete();
  }
}
```

Listing 9: Class containing methods to return system to its pre-measurement state

5. Conclusion

The implemented tool meets all the original requirements set out in the project specification. The main requirement was to integrate an interactive performance extension into the Javadoc tool. This was done by rewriting the Javadoc standard doclet in such a way that now it also generates the performance extension for all the methods which have appropriate annotation. The measuring itself is maintained by a measuring server which contains two performance measurers, each providing different quality of results; that is mainly dependent on the time intended for the warm-up and measurement. The separation of the measuring server allows us to measure results against a dedicated reference machine. The communication protocol between Javadoc and the measuring server is designed so that it is easy to add other clients.

Another requirement of the thesis was to design and implement a reasonable interface for the code which prepares the actual workload for the measured method. This goal was achieved by the workload generators, which are methods used to prepare arguments for the measured method as well as an instance, on which the measured method will be called. The structure of these generators was carefully selected to enable an easy and clear implementation of the generators.

We are not aware of any project that would add a performance extension into Javadoc, which makes our project together with the concept of workload generators innovative.

Future work on this tool may for example include measuring other metrics (e.g. memory consumption) and integration with JMH, which is the highly precise micro-benchmarking tool that may in some situations reach more accurate results than our application. The tool is ready for all these possible extensions and provides them with a reasonable interface.

Bibliography

[1] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. [online]. [cit. 2015-05-06]. Document available from `http://dl.acm.org/citation.cfm?doid=1167473.1167488`

[2] Clark, Mike. *JUnitPerf*. [online]. [cit. 2015-04-15]. Document available from `http://www.clarkware.com/software/JUnitPerf.html`

[3] *Code Tools: jmh*. [online]. [cit. 2015-04-15]. Document available from `http://openjdk.java.net/projects/code-tools/jmh/`

[4] *Comparison of JavaScript charting frameworks*. [online]. [cit. 2015-05-09]. Document available from `http://en.wikipedia.org/wiki/Comparison_of_JavaScript_charting_frameworks`

[5] *JMH tips for benchmarking*. [online]. [cit. 2015-04-15]. Document available from `http://hg.openjdk.java.net/code-tools/jmh/file/f2e982b7c51b/jmh-samples/src/main/java/org/openjdk/jmh/samples/`

[6] *jQuery*. [online] [cit. 2015-04-08]. Document available from `https://jquery.com/`

[7] Oracle. *Javadoc tool*. [online]. [cit. 2015-03-31]. Document available from www: `http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html`

[8] Oracle. *Trail: The Reflection API*. [online]. [cit. 2015-04-09]. Document available from `https://docs.oracle.com/javase/tutorial/reflect/`

[9] *Performance measurement*. [online]. [cit. 2015-04-02]. Document available from www: `http://en.wikipedia.org/wiki/Performance_measurement`

[10] Shipilev, Aleksey. *JMH vs Caliper: reference thread*. [online]. [cit. 2015-04-15]. Document available from `https://groups.google.com/forum/#!msg/mechanical-sympathy/m4opvy4xq3U/7lY8x8SvHgwJ`

[11] Shipilev, Aleksey. *Nanotrusting the Nanotime*. [online]. [cit. 2015-05-03]. Document available from www: `http://shipilev.net/blog/2014/nanotrusting-nanotime/`

[12] *Stochastic Performance Logic. (SPL)* [online]. [cit. 2015-04-15]. Document available from `http://d3s.mff.cuni.cz/projects/performance_evaluation/spl/`

A. CD contents

All the appendices can be found on the enclosed CD:

A. Application source code
The source code of the application together with external libraries, configuration files, license and build file can be found in folder *source*. The content of this folder is similar to the GitHub repository of the project (see section 4.2).

B. Javadoc of the measuring server
Javadoc of the measuring server may be found in folder *javadoc*. This folder also contains file *doclet_classes.txt*, which describes modified classes of the standard doclet.

C. JARs
JAR files for the modified standard doclet (see section 4.4) and the measuring server (see section 4.5) are located in folder *jars*.

D. Demo examples
Javadoc generated from demo classes, which show the basic functionality of the tool, is located in folder *demo*. To open an index file, go to this folder and open *index.html* file in the browser.