

PerfCake 5.x

Developers' Guide

Martin Večeřa

PerfCake 5.x: Developers' Guide

by Martin Večeřa

Table of Contents

Acknowledgements	vii
1. Introduction	1
I. General Contributors' and Developers' Guide	2
2. Developing with PerfCake	5
2.1. Using PerfCake from a Maven project	5
2.2. Maven Project	5
2.2.1. Project Structure	5
2.2.2. Maven Profiles	5
2.2.3. Standard Maven Goals	5
2.2.4. Documentation	6
2.2.5. License	6
2.2.6. JaCoCo Code Coverage	6
2.2.7. FindBugs Report	6
2.2.8. Digital Signatures	7
2.2.9. Distribution	7
2.2.10. Transformation of Scenarios	7
2.3. Coding Standards	7
2.3.1. Source file basics	8
File name	8
File encoding: UTF-8	8
Special characters	8
2.3.2. Source file structure	9
License or copyright information, if present	9
Package statement	10
Import statements	10
Class declaration	10
2.3.3. Formatting	11
Braces	11
Block indentation: +3 spaces	12
One statement per line	12
Column limit: 400	12
Line-wrapping	12
Whitespace	13
Grouping parentheses: recommended	14
Specific constructs	14
2.3.4. Naming	17
Rules common to all identifiers	17
Rules by identifier type	17
Camel case: defined	19
2.3.5. Programming practices	20
@Override: always used	20
Caught exceptions: not ignored	20
Static members: qualified using class	20
Finalizers: not used	20
2.3.6. JavaDoc	21
Formatting	21
The summary fragment	22
Where JavaDoc is used	22
2.4. Test Development	23
2.5. Writing Guides	23
3. PerfCake Architecture	24

3.1. PerfCake Architecture Overview	24
3.2. Run Info	25
3.3. Generators	25
3.3.1. Generators Architecture	25
3.3.2. Writing a New Generator	25
3.4. Developing Senders	26
3.5. Reporting	26
3.5.1. Reporting Architecture	26
3.5.2. Reporters	26
3.5.3. Destinations	26
3.5.4. Accumulators	26
3.6. Validators	27
3.6.1. Validators Architecture	27
3.6.2. Writing a New Validator	27
II. Core Developers' Guide	28
4. Release Procedure	30
5. Continuous Integration	31

List of Figures

3.1. Architecture Overview	24
3.2. Generators Architecture	25
3.3. Reporting Architecture	26
3.4. Validators Architecture	27
4.1. Prepare a release branch	30
4.2. Merge the changes to the devel branch	30
4.3. Merge the changes to the master branch as well	30
4.4. Merge the changes to the master branch as well	30

List of Tables

2.1. XML plugin properties	7
2.2. Special characters examples	9
2.3. Camel case examples	19

Acknowledgements

We would like to thank to everybody who helped PerfCake to be born and to grow up by any means.
Special thanks goes to our contributors who we love!

Chapter 1. Introduction

PerfCake is a lightweight performance testing framework and a load generator. It has a component design which means that you can build up your own performance measurements from some basic building blocks. This allows a high flexibility in configuration and the components can be reused in different tests.

PerfCake has a pluggable architecture with many supported interfaces/protocols out of the box (HTTP, REST, JMS, JDBC, SOAP, socket, file etc.). If you need support for another interface ask our community or provide the implementation. It is really easy as you will find later in this guide.

Except for interfaces/protocols, PerfCake also supports various means of generating the load. It can send a preset count of messages, it can send as many messages as the target system is able to consume for a given period of time, or it can carefully aim for maximum allowable throughput.

What would be a measurement with proper reports?! PerfCake supports various means of reporting the measured values including average throughput (possibly over a time window), memory consumption of the target JVM with linear regression analysis to see if there is a memory leak and others. There are various output formats supported. Except for console or a log file, we support CSV that can be imported as a spreadsheet. In the future we plan to add more output formats as the requests arise.

What if the target system crashed and just quickly returns error messages? There are validators that can validate the response to see if everything is OK. With validators you can also use PerfCake to write an end-to-end test of your system. Like in case of JDBC - when you provide a list of SQL commands and expected results.

To run PerfCake, one just needs to supply a so called "scenario". This is an XML file that describes the building blocks that should be used together. Then you just run the scenario using Maven or a prepared shell script if you downloaded just the binaries. We understand the need for an IDE plugin and this is one of our short term goals. You can also use PerfCake in your application using its API. In future we plan integration with Arquillian and testing frameworks including TestNG and JUnit.

We hope to make your performance testing a real piece of cake!

This book is further divided into two main parts. This first part describes general information useful for all developers and contributors. The second part is intended for core developers with major control over the project. It describes release procedure, automatic site generation etc.

Part I. General Contributors' and Developers' Guide

This part contains general information suitable for all project contributors and developers. It is a great starting point for anybody with the aspiration to contribute to PerfCake, which we really love.

Table of Contents

2. Developing with PerfCake	5
2.1. Using PerfCake from a Maven project	5
2.2. Maven Project	5
2.2.1. Project Structure	5
2.2.2. Maven Profiles	5
2.2.3. Standard Maven Goals	5
2.2.4. Documentation	6
2.2.5. License	6
2.2.6. JaCoCo Code Coverage	6
2.2.7. FindBugs Report	6
2.2.8. Digital Signatures	7
2.2.9. Distribution	7
2.2.10. Transformation of Scenarios	7
2.3. Coding Standards	7
2.3.1. Source file basics	8
File name	8
File encoding: UTF-8	8
Special characters	8
2.3.2. Source file structure	9
License or copyright information, if present	9
Package statement	10
Import statements	10
Class declaration	10
2.3.3. Formatting	11
Braces	11
Block indentation: +3 spaces	12
One statement per line	12
Column limit: 400	12
Line-wrapping	12
Whitespace	13
Grouping parentheses: recommended	14
Specific constructs	14
2.3.4. Naming	17
Rules common to all identifiers	17
Rules by identifier type	17
Camel case: defined	19
2.3.5. Programming practices	20
@Override: always used	20
Caught exceptions: not ignored	20
Static members: qualified using class	20
Finalizers: not used	20
2.3.6. JavaDoc	21
Formatting	21
The summary fragment	22
Where JavaDoc is used	22
2.4. Test Development	23
2.5. Writing Guides	23
3. PerfCake Architecture	24
3.1. PerfCake Architecture Overview	24
3.2. Run Info	25
3.3. Generators	25

3.3.1. Generators Architecture	25
3.3.2. Writing a New Generator	25
3.4. Developing Senders	26
3.5. Reporting	26
3.5.1. Reporting Architecture	26
3.5.2. Reporters	26
3.5.3. Destinations	26
3.5.4. Accumulators	26
3.6. Validators	27
3.6.1. Validators Architecture	27
3.6.2. Writing a New Validator	27

Chapter 2. Developing with PerfCake

This chapter describes various useful information for using PerfCake as a dependency in your own project. It further provides guidance for possible project contributors.

2.1. Using PerfCake from a Maven project

In this section, you can find information on using PerfCake in your own Maven project in case you do not want to or you cannot contribute directly to PerfCake repository, or you just want to its API.

For inclusion into your project, you can search for PerfCake artifacts at <http://search.maven.org>. In the results click at the version number and you will see snippets for various build systems. For Maven and version 1.0 you just include the following XML piece into your project dependencies.

```
1 <dependency>
2   <groupId>org.perfcake</groupId>
3   <artifactId>perfcake</artifactId>
4   <version>4.1</version>
5 </dependency>
```

2.2. Maven Project

The following Maven plugins and their goals can be used with PerfCake project.

2.2.1. Project Structure

There is a parent project composed of two child Maven modules. This is because PerfCake Agent installs into the application under inspection and needs to be compiled with an older Java version because of compatibility reasons.

Because of the split into multiple child modules, there is a need to run *mvn install -DskipTests* to install the parent module into your local repository when working on the *devel* Git branch.

- *perfcake* Core PerfCake module, contains all the code needed to build the product.
- *perfcake-agent* JVM Agent to be used when running target application (application under inspection) with memory monitoring. This module is compiled with an older Java version (1.5).

2.2.2. Maven Profiles

Use the following Maven profiles to achieve desired results.

- *production* Generates output classes without debugging information. This produces smaller results and is intended for binary distributions. This must be used during the release procedure.
- *sign* Automatically adds digital signatures of build artifacts during the *install* phase. Needs properly configured GPG agent.

Profiles for controlling test groups can be found in section Section 2.4, “Test Development” .

2.2.3. Standard Maven Goals

We try to minimize the necessity for running standard Maven goals in a different manner or in a way they are not originally intended to. So far we use the following goals.

- *clean* Cleans the compiled artifacts and reports. Basically deletes the `target` directory.
- *compile* Compiles the whole project including tests.
- *test* Executes all the tests. See section Profiles for controlling test groups can be found in section Section 2.4, “Test Development” for Maven profiles that control test groups.
- *install* Install project artifacts into the local Maven repository. This is needed when working in the *devel* Git branch with snapshot versions.
- *package* Packages the distribution
- *exec:exec* Works only in the *perfcake* module. Runs PerfCake, parameters are passed using `-Dparam=value`. Never use *exec:java* directly as Maven will comply about missing configuration (the configuration is passed in from *exec:exec*).
- *site* Works only in the *perfcake* module. Generates the Maven HTML site. Consumes output of other plugins like JaCoCo. If the other consumables should be part of the generated site, they must be ready before calling this goal.

2.2.4. Documentation

It is possible to generate JavaDoc for individual Maven modules as well as a single aggregated package for all of them.

- *javadoc:javadoc* Generates documentation for the current Maven module.
- *javadoc:aggregate* Generates aggregated documentation package when executed in the parent project.

2.2.5. License

PerfCake is licensed under Apache License 2.0 and the license text can be found in `LICENSE.txt` file. `license-maven-plugin` is used to check that each file has the appropriate header.

- *license:check-file-header* Checks the current status of the license header in project files. This goal claims to modify the files but nothing is done in fact. The output is a bit misleading.
- *license:update-file-header* Performs the actual update of file headers. Note that we recognize the true license by two hard space characters at the end of some blank lines. Carefully configure your IDE to preserve these.

2.2.6. JaCoCo Code Coverage

JaCoCo is configured to use a Java Agent so no class instrumentation is needed. The agent is automatically switched on for running tests. The only goal needed is report generation (`jacoco:report`) which must be done together or after test execution.

The resulting report can be found in the `target/site/jacoco` directory.

2.2.7. FindBugs Report

We use FindBugs to warn us about bad code practices. To obtain the report execute the *findbugs:findbugs* goal. FindBugs works with the compiled classes, so for a fresh report, the `compile` target must be invoked.

To inspect the code on the fly, run the *findbugs:gui* tool. Please note that it only shows the data from the previous analysis run.

2.2.8. Digital Signatures

To sign the artifacts created by the *package* goal, one need to run the *gpg:sign* goal. The resulting signatures accompany the files in the `target` directory.

2.2.9. Distribution

To create the archived distributions we offer for download at our website, simply run the *assembly:single* goal. Its output can be found in the `target` directory as usually.

2.2.10. Transformation of Scenarios

It is possible to automatically transform scenarios from previous versions of PerfCake into the newest one. There are XSTL stylesheets included and they can be used by invoking the *xml:transform* goal.

There are properties that can be used to control the transformation. They are described in the following table.

Property name	Description	Required	Default value
<code>transform.scenarios.dir</code>	An input directory where the XML plugin will look for the scenarios to transform.	Yes	-
<code>transform.scenarios.outputDir</code>	An output directory where the transformed scenarios will be placed.	No	<code>\${project.build.directory}/scenarios</code>

Table 2.1. XML plugin properties

2.3. Coding Standards

This section describes coding standards used for the PerfCake project. The detailed guidelines are based on Google Java Coding Standard ¹ with some changes applied. For a brief overview, please make sure to:

- Develop tests for your features/fixes (keep the high level of test coverage),
- Use the appropriate code style and formatting (Eclipse configuration file is present in the repo),
- Avoid introducing new FindBugs warnings (see Developers' Guide to find out how to generate the site report),
- Keep backward compatibility unless previously discussed with the developers,
- Write/update the JavaDoc comments,

¹ <http://google-styleguide.googlecode.com/svn/trunk/javaguide.html>

- Avoid bad practices ²,
- You do not keep unused imports, FIXME comments, unused variables, cryptic code or anything ugly,
- Submit only a masterpiece code that you could be proud of.

Following is the complete definition of PerfCake's coding standards for source code in the Java™ Programming Language. A Java source file is described as being in PerfCake Style if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the hard-and-fast rules that we follow universally, and avoids giving advice that isn't clearly enforceable (whether by human or tool).

In this document, unless otherwise clarified:

1. The term *class* is used inclusively to mean an "ordinary" class, enum class, interface or annotation type (`@interface`),
2. The term *comment* always refers to implementation comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

Example code in this document is non-normative. That is, while the examples are in PerfCake Style, they may not illustrate the only stylish way to represent the code. Optional formatting choices made in examples should not be enforced as rules.

Previous four paragraphs are copied from the Google Java Coding Standard ¹ as well as the rest of this section.

2.3.1. Source file basics

File name

The source file name consists of the case-sensitive name of the top-level class it contains, plus the `.java` extension.

File encoding: UTF-8

Source files are encoded in UTF-8.

Special characters

Whitespace characters

Aside from the line terminator sequence, the **ASCII horizontal space character (0x20)** is the only whitespace character that appears anywhere in a source file. This implies that:

1. All other whitespace characters in string and character literals are escaped,
2. Tab characters are not used for indentation.

² <http://blog.codinghorror.com/new-programming-jargon/>

Special escape sequences

For any character that has a special escape sequence (`\b`, `\t`, `\n`, `\f`, `\r`, `\"`, `\'` and `\\`), that sequence is used rather than the corresponding octal (e.g. `\012`) or Unicode (e.g. `\u000a`) escape.

Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character (e.g. ∞) or the equivalent Unicode escape (e.g. `\u221e`) is used, depending only on which makes the code **easier to read and understand**.

Tip

In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

Example	Discussion
<code>String unitAbbrev = "\u03bc";</code>	Best: perfectly clear even without a comment.
<code>// "\u03bc" String unitAbbrev = "\u03bcs";</code>	Allowed, but there's no reason to do this.
<code>// Greek letter mu, "s" String unitAbbrev = "\u03bcs";</code>	Allowed, but awkward and prone to mistakes.
<code>String unitAbbrev = "\u03bcs";</code>	Poor: the reader has no idea what this is.
<code>// byte order mark return '\uffff' + content;</code>	Good: use escapes for non-printable characters, and comment if necessary.

Table 2.2. Special characters examples

Tip

Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are **broken** and they must be **fixed**.

2.3.2. Source file structure

A source file consists of, **in order**:

1. License or copyright information, if present,
2. Package statement,
3. Import statements,
4. Exactly one top-level class.

Exactly one blank line separates each section that is present.

License or copyright information, if present

If license or copyright information belongs in a file, it belongs here.

Package statement

The package statement is **not line-wrapped**. The column limit (the section called “Column limit: 400”) does not apply to package statements.

Import statements

No wildcard imports

Wildcard imports, static or otherwise, **are not used**.

No line-wrapping

Import statements are **not line-wrapped**. The column limit (the section called “Column limit: 400”) does not apply to import statements.

Ordering and spacing

Import statements are divided into the following groups, in this order, with each group separated by a single blank line:

1. All static imports in a single group
2. `org.perfcake` imports (only if this source file is in the `org.perfcake` package space)
3. Third-party imports, one group per top-level package, in ASCII sort order
 - for example: `com`, `org`, `junit`, `sun`
4. `java` imports
5. `javax` imports

Within a group there are no blank lines, and the imported names appear in ASCII sort order. (**Note:** this is not the same as the import *statements* being in ASCII sort order; the presence of semicolons warps the result.)

Class declaration

Exactly one top-level class declaration

Each top-level class resides in a source file of its own.

Class member ordering

The ordering of the members of a class can have a great effect on learnability, but there is no single correct recipe for how to do it. Different classes may order their members differently.

What is important is that each class order its members in **some logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.

Overloads: never split

When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no intervening members.

2.3.3. Formating

Note

Terminology Note: *block-like construct* refers to the body of a class, method or constructor. Note that, by the section called “Arrays”, any array initializer *may* optionally be treated as if it were a block-like construct.

Braces

Braces are used where optional

Braces are used with `if`, `else`, `for`, `do` and `while` statements, even when the body is empty or contains only a single statement.

Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for *nonempty* blocks and block-like constructs:

- No line break before the opening brace,
- Line break after the opening brace,
- Line break before the closing brace
- Line break after the closing brace *if* that brace terminates a statement or the body of a method, constructor or *named* class. For example, there is *no* line break after the brace if it is followed by `else` or a comma.

Example:

```
return new MyClass() {  
    @Override public void method() {  
        if (condition()) {  
            try {  
                something();  
            } catch (ProblemException e) {  
                recover();  
            }  
        }  
    }  
};
```

A few exceptions for enum classes are given in the section called “Enum classes”.

Empty blocks: may be concise

An empty block or block-like construct *may* be closed immediately after it is opened, with no characters or line break in between (`{}`), **unless** it is part of a *multi-block statement* (one that directly contains multiple blocks: `if/else-if/else` or `try/catch/finally`).

Example:

```
void doNothing() {}
```

Block indentation: +3 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in the section called “Nonempty blocks: K & R style”.)

One statement per line

Each statement is followed by a line-break.

Column limit: 400

PerfCake uses a column limit of 400 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in the section called “Line-wrapping”.

Line-wrapping

Terminology Note: When code that might otherwise legally occupy a single line is divided into multiple lines, typically to avoid overflowing the column limit, this activity is called *line-wrapping*.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

Tip

Extracting a method or local variable may solve the problem without the need to line-wrap.

Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**. Also:

1. When a line is broken at a *non-assignment* operator the break comes *before* the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)
 - This also applies to the following "operator-like" symbols: the dot separator (`.`), the ampersand in type bounds (`<T extends Foo & Bar>`), and the pipe in catch blocks (`catch (FooException | BarException e)`).
2. When a line is broken at an *assignment* operator the break typically comes *after* the symbol, but either way is acceptable.
 - This also applies to the "assignment-operator-like" colon in an enhanced `for` ("foreach") statement.
3. A method or constructor name stays attached to the open parenthesis (`(`) that follows it.
4. A comma (`,`) stays attached to the token that precedes it.

Indent continuation lines at least +6 spaces

When line-wrapping, each line after the first (each *continuation line*) is indented at least +6 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +6 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

the section called “Horizontal alignment: never required” addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

Whitespace

Vertical whitespace

A single blank line appears:

1. *Between* consecutive members (or initializers) of a class: fields, constructors, methods, nested classes, static initializers, instance initializers.
 - **Exception:** A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create *logical groupings* of fields.
2. Within method bodies, as needed to create *logical groupings* of statements.
3. *Optionally* before the first member or after the last member of the class (neither encouraged nor discouraged).
4. As required by other sections of this document (such as the section called “Import statements”).

Multiple consecutive blank lines are permitted, but never required (or encouraged).

Horizontal whitespace

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places^{**only**}.

1. Separating any reserved word, such as `if`, `for` or `catch`, from an open parenthesis (`()`) that follows it on that line.
2. Separating any reserved word, such as `else` or `catch`, from a closing curly brace (`}`) that precedes it on that line.
3. Before any open curly brace (`{`), with two exceptions:
 - `@SomeAnnotation({a, b})` (no space is used)
 - `String[][] x = {{ "foo" }};` (no space is required between `{`, by item 8 below)
4. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:
 - the ampersand in a conjunctive type bound: `<T extends Foo & Bar>`
 - the pipe for a catch block that handles multiple exceptions: `catch (FooException | BarException e)`
 - the colon (`:`) in an enhanced `for` ("foreach") statement
5. After `,` `:` `;` or the closing parenthesis (`)`) of a cast
6. On both sides of the double slash (`//`) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required
7. Between the type and variable of a declaration: `List<String> list`

8. *Optional* just inside both braces of an array initializer

- `new int[] {5, 6}` and `new int[] { 5, 6 }` are both valid

Note

This rule never requires or forbids additional space at the start or end of a line, only interior space.

Horizontal alignment: never required

Terminology Note: *Horizontal alignment* is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is **never required** by PerfCake Style. It is not even required to *maintain* horizontal alignment in places where it was already used. Here is an example without alignment, then using alignment:

```
private int x; // this is fine
private Color color; // this too

private int    x;      // permitted, but future edits
private Color color; // may leave it unaligned
```

Tip

Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is **allowed**. More often it prompts the coder (perhaps you) to adjust whitespace on nearby lines as well, possibly triggering a cascading series of reformatting. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.

Grouping parentheses: recommended

Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is not reasonable to assume that every reader has the entire Java operator precedence table memorized.

Specific constructs

Enum classes

After each comma that follows an enum constant, a line-break is optional.

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see the section called “Arrays”).

```
private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

Since enum classes *are classes*, all other rules for formatting classes apply.

Variable declarations

- **One variable per declaration**

Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.

- **Declared when needed, initialized as soon as possible**

Local variables are **not** habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

Arrays

- **Array initializers: can be "block-like"**

Any array initializer may optionally be formatted as if it were a "block-like construct." For example, the following are all valid (not an exhaustive list):

```
new int[] {          new int[] {
    0, 1, 2, 3
}                    0,
                    1,
                    2,
                    3,
new int[] {          }
    0, 1,
    2, 3
}                    new int[]
                    {0, 1, 2, 3}
```

- **No C-style array declarations**

The square brackets form a part of the type, not the variable: `String[] args, notString args[]`.

Switch statements

Terminology Note: Inside the braces of a *switch block* are one or more *statement groups*. Each statement group consists of one or more *switch labels* (either `case FOO:` or `default:`), followed by one or more statements.

- **Indentation**

As with any other block, the contents of a switch block are indented +3.

After a switch label, a newline appears, and the indentation level is increased +3, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

- **Fall-through: commented**

Within a switch block, each statement group either terminates abruptly (with `break`, `continue`, `return` or thrown exception), or is marked with a comment to indicate that execution will or *might* continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically `// fall through`). This special comment is not required in the last statement group of the switch block. Example:

```
switch (input) {
    case 1:
    case 2:
```

```
        prepareOneOrTwo();  
        // fall through  
    case 3:  
        handleOneTwoOrThree();  
        break;  
    default:  
        handleLargeNumber(input);  
}
```

- **The default case is present**

Each switch statement includes a `default` statement group, even if it contains no code.

Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (the section called “Line-wrapping”), so the indentation level is not increased. Example:

```
@Override  
@Nullable  
public String getNameIfPresent() { ... }
```

Exception: A single parameterless annotation may instead appear together with the first line of the signature, for example:

```
@Override public int hashCode() { ... }
```

Annotations applying to a field also appear immediately after the documentation block, but in this case, *multiple* annotations (possibly parameterized) may be listed on the same line; for example:

```
@Partial @Mock DataLoader loader;
```

There are no specific rules for formatting parameter and local variable annotations.

Comments

- **Block comment style**

Block comments are indented at the same level as the surrounding code. They may be in `/* ... */` style or `// ...` style. For multi-line `/* ... */` comments, subsequent lines must start with `*` aligned with the `*` on the previous line.

```
/*  
 * This is           // And so           /* Or you can  
 * okay.             // is this.         * even do this. */  
*/
```

Comments are not enclosed in boxes drawn with asterisks or other characters.

Tip

When writing multi-line comments, use the `/* ... */` style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in `// ...` style comment blocks.

Modifiers

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

**public protected private abstract static final transient
volatile synchronized native strictfp**

Numeric literals

long-valued integer literals use an uppercase `L` suffix, never lowercase (to avoid confusion with the digit 1). For example, `3000000000L` rather than `3000000000l`.

It is advised to use the numeric literal underscore separator (`_`) as it comes with Java 7. This is even better for readability: `3_000_000_000L`.

2.3.4. Naming

Rules common to all identifiers

Identifiers use only ASCII letters and digits, and in two cases noted below, underscores. Thus each valid identifier name is matched by the regular expression `\w+`.

In PerfCake Style special prefixes or suffixes, like those seen in the examples `name_`, `mName`, `s_name` and `kName`, are **not** used.

Rules by identifier type

Package names

Package names are all lowercase, singular, with consecutive words simply concatenated together (no underscores). For example, `com.example.deepspace`, not `com.example.deepSpace` or `com.example.deep_space`.

Class names

Class names are written in UpperCamelCase (the section called “Camel case: defined”).

Class names are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

There are no specific rules or even well-established conventions for naming annotation types.

Test classes are named starting with the name of the class they are testing, and ending with `Test`. For example, `HashTest` or `HashIntegrationTest`.

Method names

Method names are written in lowerCamelCase (the section called “Camel case: defined”).

Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop`.

Underscores may appear in JUnit test method names to separate logical components of the name. One typical pattern is `test<MethodUnderTest>_<state>`, for example `testPop_emptyStack`. There is no **one correct way** to name test methods.

Constant names

Constant names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores. But what *is* a constant, exactly?

Every constant is a static final field, but not all static final fields are constants. Before choosing constant case, consider whether the field *really feels* like a constant. For example, if any of that instance's observable state can change, it is almost certainly not a constant. Merely *intending* to never mutate the object is generally not enough.

Examples:

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES =
    ImmutableList.of("Ed", "Ann");

// because Joiner is immutable
static final Joiner COMMA_JOINER = Joiner.on(', ');
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }

// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements =
    ImmutableSet.of(mutable);
static final Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
```

These names are typically nouns or noun phrases.

Non-constant field names

Non-constant field names (static or otherwise) are written in `lowerCamelCase` (the section called “Camel case: defined”).

These names are typically nouns or noun phrases. For example, `computedValues` or `index`.

Parameter names

Parameter names are written in `lowerCamelCase` (the section called “Camel case: defined”).

One-character parameter names should be avoided.

Local variable names

Local variable names are written in `lowerCamelCase` (the section called “Camel case: defined”), and can be abbreviated more liberally than other types of names.

However, one-character names should be avoided, except for temporary and looping variables.

Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

Type variable names

Each type variable is named in one of two styles:

- A single capital letter, optionally followed by a single numeral (such as E,T,X,T2).
- A name in the form used for classes (see Class names above), followed by the capital letter T (examples:RequestT,FooBarT).

Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, Google Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
 - *Recommended*: if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case *per se*; it defies *any* convention, so this recommendation does not apply.
3. Now lowercase *everything* (including acronyms), then uppercase only the first character of:
 - ... each word, to yield upper camel case, or
 - ... each word except the first, to yield lower camel case.
4. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded. Examples:

Prose form	Correct	Incorrect
"XML HTTP request"	XmlHttpRequest	XMLHTTPRequest
"new customer ID"	newCustomerId	newCustomerID
"inner stopwatch"	innerStopwatch	innerStopWatch
"supports IPv6 on iOS?"	supportsIpv6OnIos	supportsIPv6OnIOS
"YouTube importer"	YouTubeImporter, YoutubeIm- porter*	

Table 2.3. Camel case examples

*Acceptable, but not recommended.

Note

Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names `checkNonempty` and `checkNonEmpty` are likewise both correct.

2.3.5. Programming practices

@Override: always used

A method is marked with the `@Override` annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method, and an interface method respecifying a superinterface method.

Exception: `@Override` may be omitted when the parent method is `@Deprecated`.

Caught exceptions: not ignored

Except as noted below, it is very rarely correct to do nothing in response to a caught exception. Typical responses are to log it, or if it is considered "impossible", rethrow it as an `IllegalStateException`.

When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

```
try {
    int i = Integer.parseInt(response);
    return handleNumericResponse(i);
} catch (NumberFormatException ok) {
    // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

Exception: In tests, a caught exception may be ignored without comment if it is named expected. The following is a very common idiom for ensuring that the method under test does throw an exception of the expected type, so a comment is unnecessary here.

```
try {
    emptyStack.pop();
    fail();
} catch (NoSuchElementException expected) {
}
```

Static members: qualified using class

When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type.

```
Foo aFoo = ...;
Foo.aStaticMethod(); // good
aFoo.aStaticMethod(); // bad
somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

Finalizers: not used

It is **extremely rare** to override `Object.finalize`.

Tip

Don't do it. If you absolutely must, first read and understand *Effective Java* [<http://books.google.com/books?isbn=8131726592>] Item 7, "Avoid Finalizers," very carefully, and then don't do it.

2.3.6. Javadoc

Formatting

General form

The *basic* formatting of Javadoc blocks is as seen in this example:

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

... or in this single-line example:

```
/** An especially short bit of Javadoc. */
```

The basic form is always acceptable. The single-line form may be substituted when there are no at-clauses present, and the entirety of the Javadoc block (including comment markers) can fit on a single line.

Paragraphs

One blank line -- that is, a line containing only the aligned leading asterisk (*) -- appears between paragraphs, and before the group of "at-clauses" if present. Each paragraph but the first has `<p>` immediately before the first word, with no space after.

At-clauses

Any of the standard "at-clauses" that are used appear in the order `@param`, `@return`, `@throws`, `@deprecated`, and these four types never appear with an empty description. When an at-clause doesn't fit on a single line, continuation lines are indented **six** (or more) spaces from the position of the `@`.

Language

Use the third person in describing class purpose, method actions, parameters, return types, exceptions and all others. Usually, present tense should be used. All sentences including those describing parameters descriptions, thrown exceptions, and return types start with an uppercase letter and end with a full stop.

Avoid using reference to the item being described like *this class*, *the purpose of this method*, *it stores*, etc. Also avoid obvious verbs including *used to*, *returns*, etc. All these references are very obvious in the resulting Javadoc. Simply describe the immediate actions, purpose and all tohers. The same applies to exceptions being thrown.

Methods intended for fluent API returning `this` should describe the return value as *Instance of this to support fluent API*.

Following is an example of Javadoc with the proper language:

```
/**
 * Gets a value of an accumulated result.
 *
 * @param key
```

```
*      Key in the results hash map.
* @return The value associated with the given key.
*/
protected Object getAccumulatedResult(final String key) {
    ...
}
```

The summary fragment

The Javadoc for each class and member begins with a brief **summary fragment**. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment -- a noun phrase or verb phrase, not a complete sentence. It does not begin with `A {code Foo} is a...`, or `This method returns...`, nor does it form a complete imperative sentence like `Save the record..` However, the fragment is capitalized and punctuated as if it were a complete sentence.

Tip

A common mistake is to write simple Javadoc in the form `/** @return the customer ID */`. This is incorrect, and should be changed to

```
/**
 * Gets the customer ID.
 * @return The customer ID.
 */
```

All of the sentences should be properly ended by a period including the sentences following "at-clauses".

Where Javadoc is used

At the *minimum*, Javadoc is present for every `public` class, and every `public` or `protected` member of such a class, with a few exceptions noted below.

Other classes and members still have Javadoc *as needed*. Whenever an implementation comment would be used to define the overall purpose or behavior of a class, method or field, that comment is written as Javadoc instead (it's more uniform, and more tool-friendly.)

Exception: self-explanatory methods

Javadoc is optional for "simple, obvious" methods like `getFoo`, in cases where there *really and truly* is nothing else worthwhile to say but "Returns the foo".

Important

It is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a method named `getCanonicalName`, don't omit its documentation (with the rationale that it would say only `/** Returns the canonical name. */`) if a typical reader may have no idea what the term "canonical name" means!

Exception: overrides

Javadoc is not always present on a method that overrides a supertype method.

2.4. Test Development

All classes in PerfCake are supposed to have a good test coverage. This might not currently be the situation for all of them but it definitely is our goal. When developing new code, please make sure the tests are developed as well.

There are the following test groups defined. Each of the test methods or classes should be categorized at least into one of them.

- *unit* - Test of a single component (class).
- *integration* - Tests of multiple components working well in the complete context of PerfCake and in relation to other components.
- *performance* - Tests of a performance of a component or multiple components to make sure they are not a bottle-neck.
- *stress* - Tests to verify that a component or multiple components work correctly under a heavy load.
- *ueber* - Advanced integration/performance/stress tests that verify components at larger scale than previous groups and take significant amount of time. Tests in this group are not executed by default, unless a Maven profile *allTests* is used.

To find out what is the current code coverage of tests, refer to the Section 2.2.6, “JaCoCo Code Coverage” .

2.5. Writing Guides

This section describes guidelines to writing PerfCake guides. It contains information on styles used, language, and good practices.

TODO: This chapter will be completed based on the experiences with completing this guide.

Chapter 3. PerfCake Architecture

In this chapter we describe the inner architecture of PerfCake. First we inspect the overall design and then we delve deeper into the individual parts.

There is an easy concept that is good to bear in mind while working with PerfCake -- a single communication unit called a Message . PerfCake usually sends the message to the target system and measures the response time needed by the system being measured to provide a valid response. It is also the basic unit of load that can be generated for the target system.

Let's investigate the architecture in more details.

3.1. PerfCake Architecture Overview

Let's start with a Figure that is worth a thousand words.

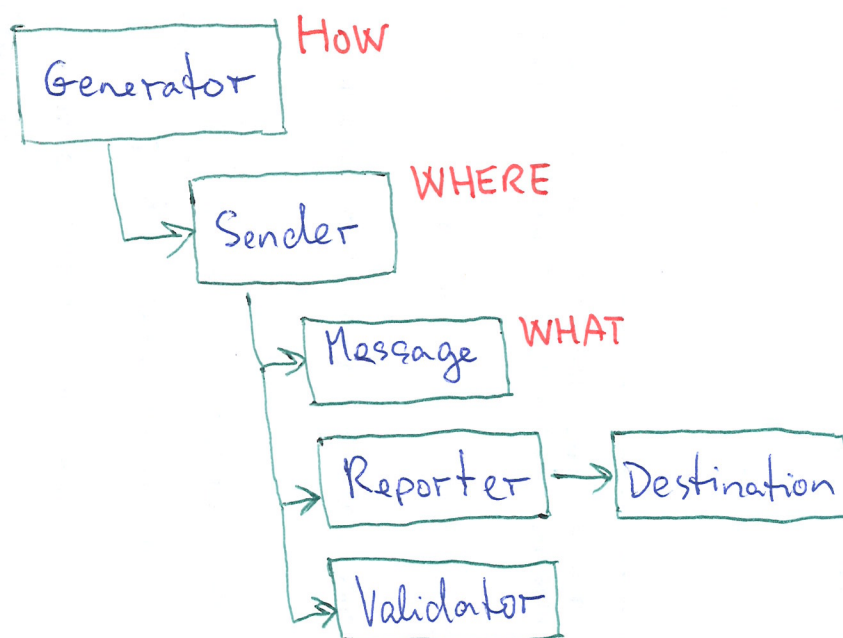


Figure 3.1. Architecture Overview

In the Figure 3.1, “Architecture Overview”, we can see the high level overview of PerfCake's architecture. There is always a single Generator. The main purpose of the Generator is to specify how the messages are generated. The easiest case would be to send a message to the target system, wait for the response and measure the response time. However, this would not tell us anything about performance of the target system. What is more interesting is a load generated in many parallel threads as fast as possible.

The default generator implementation (DefaultMessageGenerator) has a queue of messages ready to be send by a Sender to the target system. More details about Generator architecture can be found in the next section. For now, just remember that Generator specifies HOW the messages are sent.

A Sender specifies the protocol that is used to send the message. This can be HTTP, REST, JMS, JDBC, SOAP, socket, file etc. The Sender is configured with a specific address for the given protocol. So the Sender tells WHERE TO send the messages.

A Message is the content of the smallest unit of load that is sent to the target system. So basically it says WHAT is sent.

The procedure of sending the messages is carefully monitored by the reporting facility of PerfCake. This allows us to write results to various outputs.

Finally, there are validators that prove the responses to be valid. Some system can start responding very fast under a heavy load, but just with a blunt overload warning.

More details about individual parts of the application can be found in later sections.

3.2. Run Info

TODO: What is RunInfo, how does it control the run.

3.3. Generators

In this section we will reveal as much details about message generators as possible. First we describe the architecture and then we discuss an approach to develop a new generator.

3.3.1. Generators Architecture

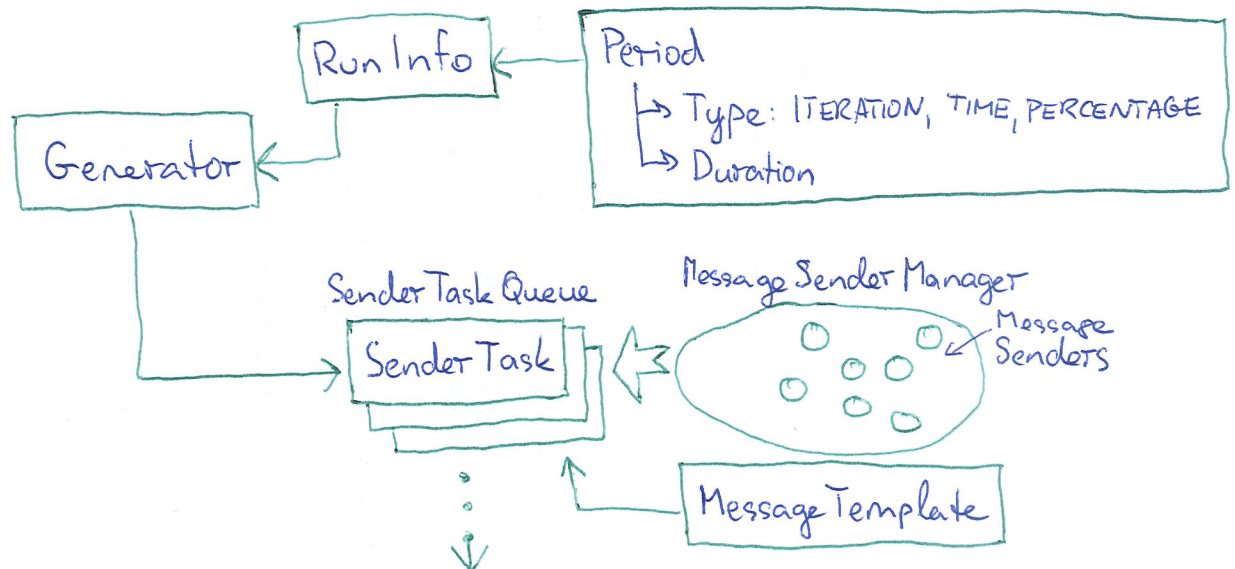


Figure 3.2. Generators Architecture

TODO: Describe the architecture.

3.3.2. Writing a New Generator

TODO: Introduce AbstractMessageGenerator in detail, how to inherit, what methods to override, RunInfo reference, SenderTask.

3.4. Developing Senders

TODO: How to develop a new sender, how to use inheritance from existing senders, how are senders used in a thread pool, are they thread safe?

To allow fluent API usage with senders, we strongly encourage you to make sure all your senders' setters return `this`.

3.5. Reporting

In this section we will reveal as much details about reporting facilities as possible. First we describe the architecture and then we discuss an approach to develop new reporters, destinations and accumulators.

3.5.1. Reporting Architecture

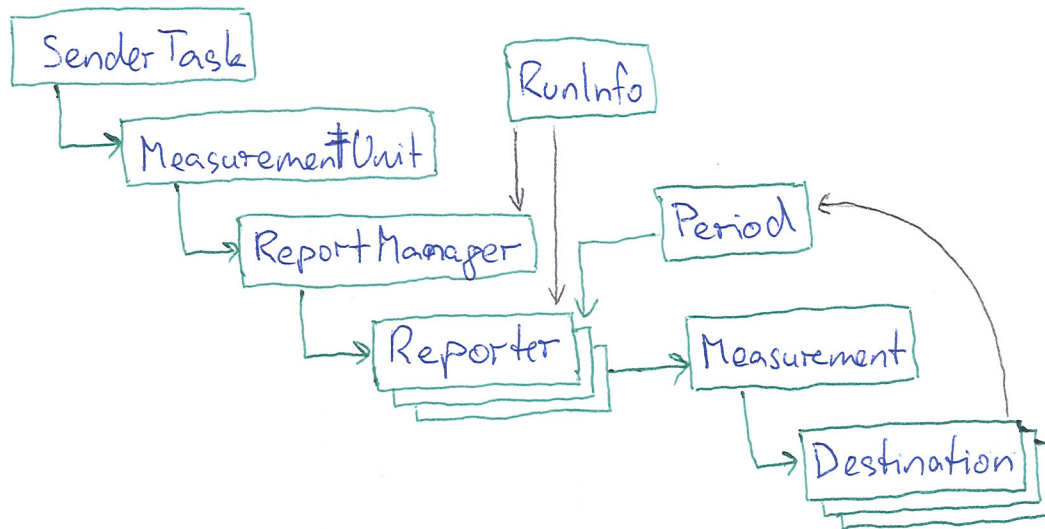


Figure 3.3. Reporting Architecture

TODO: Describe the architecture.

3.5.2. Reporters

TODO: Developing new reporters.

3.5.3. Destinations

TODO: Developing new destinations.

3.5.4. Accumulators

TODO: Developing a new accumulator.

3.6. Validators

In this section we will reveal as much details about validators as possible. First we describe the architecture and then we discuss an approach to develop new validators.

3.6.1. Validators Architecture

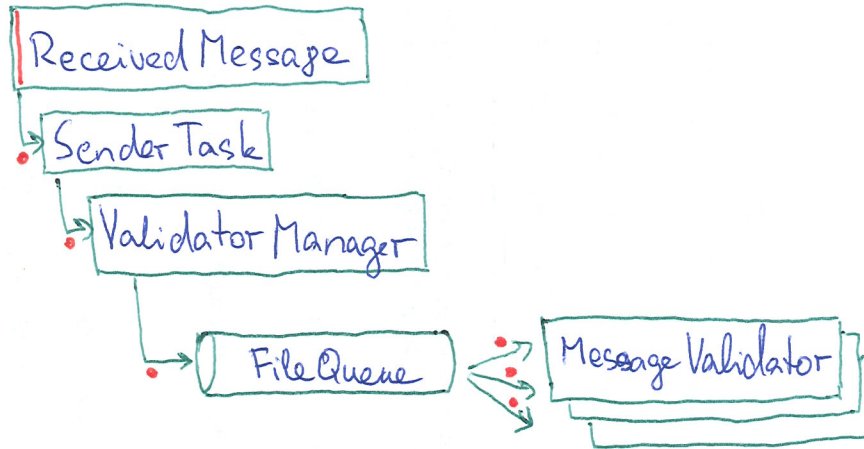


Figure 3.4. Validators Architecture

TODO: Describe the architecture.

3.6.2. Writing a New Validator

TODO: ValidationManager, how are called in parallel thread not to influence measurement, how to write a new validator.

Part II. Core Developers' Guide

This part is intended purely to core developers who control the project releases, configuration of 3rd party tools etc.

Table of Contents

4. Release Procedure	30
5. Continuous Integration	31

Chapter 4. Release Procedure

This chapter describes details about the release procedure using Maven and following the git flow principles. The target audience are core developers with appropriate access rights.

First, start off with a new release branch and let Maven prepare the release for us Figure 4.1, “Prepare a release branch”.

```
$ git checkout -b release/v{VERSION} devel
$ mvn release:prepare -Psign -Pproduction
```

Figure 4.1. Prepare a release branch

At this point, you should get the artifacts from the `target` directory for later publishing Figure 4.2, “Merge the changes to the devel branch”.

Never ever try to perform the release using the Maven release plugin. It cannot be configured to publish the artifacts and breaks the git-flow branching/tagging.

The rest is about getting git into a good shape. This starts by getting the updated `pom.xml` into the devel branch.

```
$ git checkout devel
$ git merge --no-ff release/v{VERSION}
```

Figure 4.2. Merge the changes to the devel branch

We need to get the correct `pom.xml` to the master branch as well. For this, we will use the commit prior to the head in the release branch Figure 4.3, “Merge the changes to the master branch as well”.

```
$ git checkout master
$ git merge --no-ff release/v{VERSION}~1
```

Figure 4.3. Merge the changes to the master branch as well

The last step is to clean everything up and go public Figure 4.4, “Merge the changes to the master branch as well”.

```
$ git branch -D release/v{VERSION}
$ git push --all && git push --tags
```

Figure 4.4. Merge the changes to the master branch as well

Only users the approved account in the Sonatype JIRA ¹ can upload the artifacts. Then follow the rules specified in the Sonatype OSS Maven Repository Usage Guide ².

¹ <https://issues.sonatype.org/browse/OSSRH-7134>

² <http://goo.gl/xBSm4>

Chapter 5. Continuous Integration

This chapter describes details about the continuous integration we use at the PerfCake project. The target audience are core developers with appropriate access rights. The content will be filled in base on the experiences with the release 1.0.

TODO: How to configure jobs, what artifacts they generate, how to verify pull requests...