Otto von Guericke University Magdeburg

Faculty of Computer Science
Department of Technical and Business Information Systems

Master Thesis

The Impact of Refactorings on Non–Functional Software Properties

Author:
Stefan Moschinski

January 29, 2011

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Martin Kuhlemann
Dipl.-Inform. Norbert Siegmund

Otto von Guericke University Magdeburg
Faculty of Computer Science
P.O. Box 4120, 39016 Magdeburg, Germany

Moschinski, Stefan:
The Impact of Refactorings on Non–Functional Software Properties
Master Thesis, Otto von Guericke University Magdeburg, 2011.

Acknowledgment

First of all, I want to thank my parents for all their support.

Moreover, I would like to thank my advisors Martin and Norbert for their helpful comments regarding my work. Particularly, their extremely fast feedbacks in the final stage of the thesis were very beneficial.

In addition, I thank Franz and Marc for reading and commenting my work.

Last but not least, I want to thank Frank, Eva, and Sebbo for their moral support during the time when this thesis was written.

Contents

Contents iii

List of Figures vii

List of Tables ix

List of Abbreviations xiii

1 Introduction 1

1.1 Goals . . . 2

1.2 Structure . . . 2

2 Background 5

2.1 Refactoring . . . 5

2.1.1 Definition . . . 5

2.1.2 Performance Optimization . . . 6

2.1.3 Analyzed Refactorings . . . 7

2.2 Background Java . . . 8

2.2.1 JIT compiler . . . 9

2.3 Benchmarks and Experimental Setup . . . 10

2.3.1 Micro—Benchmarks . . . 10

2.3.2 Optimization breakpoint . . . 10

2.3.3 Used Java Virtual Machines and Hardware . . . 11

2.4 Statistical Analysis . . . 13

2.4.1 Descriptive Statistic . . . 13

2.4.2 Inferential Statistic . . . 14

2.5 Memory Footprint . . . 16

2.5.1 Instance Memory Footprint 17
2.5.2 Class Memory Footprint 18
2.5.3 Hypotheses 19

3 Performance **21**
3.1 Replace Inheritance with Delegation 21
3.1.1 Methodology 21
3.1.2 Hypotheses 22
3.1.3 Results 23
3.1.4 Influence of the JIT compiler 28
3.1.5 Conclusion 30
3.2 Inline Method 30
3.2.1 Methodology 31
3.2.2 Hypotheses 31
3.2.3 Results 31
3.2.4 Conclusion 36
3.3 Inline Class 37
3.3.1 Methodology 38
3.3.2 Hypotheses 38
3.3.3 Results 38
3.4 Summary 44

4 Memory Footprint **47**
4.1 Replace Inheritance with Delegation 47
4.1.1 Methodology 47
4.1.2 Results 48
4.1.3 Conclusion 55
4.2 Inline Method 56
4.2.1 Methodology 57
4.2.2 Results 57
4.2.3 Conclusion 62
4.3 Inline Class 63
4.3.1 Methodology 63

4.3.2 Results 63
4.3.3 Conclusion 71
4.4 Summary 71

5 Evaluation **75**
5.1 Benchmarking 75
5.2 Inline Method 76
5.2.1 Hypotheses 77
5.2.2 Results 78
5.2.3 Summary 81
5.3 Inline Class 81
5.3.1 Hypothesis 82
5.3.2 Results 82
5.3.3 Summary 85
5.4 Replace Inheritance with Composition 85

6 Related Work **87**
6.1 Performance 87
6.2 Memory Footprint 89

7 Conclusion **91**
7.1 Future Work 92

Bibliography **95**

List of Figures

2.1 The `Moped` class is refactored by *Replace Inheritance with Delegation* to the wrapper `Moped` class. 7
2.2 An example for method inlining. 8
2.3 A class inlining example. 9

3.1 A comparison of the mean object creation times for a (refactored) wrapper object and objects with a varying inheritance depth on a client–mode HotSpot VM 6. 23
3.2 A comparison of the mean object creation times for a wrapper object and objects with a varying inheritance depth on different virtual machines. . 24
3.3 A comparison of the mean times for object creation for a wrapper object and objects with an inheritance depth of two. 24
3.4 A comparison of the mean mehtod invocation times on a composed object and objects with a varying inheritance depth on a client–mode HotSpot 6. 26
3.5 A comparison of the mean method invocation times for a composed (i.e., refactored) object and objects with a varying inheritance depth on different virtual machines. 27
3.6 The mean performance benefit due to inlining a varying number of methods in the 1000 iterations benchmark on a HotSpot 6 VMCM. 32
3.7 The mean performance benefit due to *Inline Method* on a JRockit VM for a varying number of method inlinings and benchmark iterations. 35
3.8 The cumulated performance benefit due to the inlining of nine methods after 5000 and 20,000 iterations. 36
3.9 The cumulated decrease in runtime due to class inlining on an interpreted–mode HotSpot 6 VM. 40
3.10 The cumulated decrease in runtime due to class inlining on an client–mode HotSpot 6 VM. 41

4.1 The increase of the CMF per level of inheritance for an empty class. . . . 50
4.2 Comparison between the CMF of an inherited class and the CMF of a composed class. The inheritance depth of the inherited classes equals the number their overridden methods plus one. For example, the inheritance–based class with four overridden methods has an inheritance depth of five. 54
4.3 The increase of memory footprint of an empty class by adding empty methods. 58
4.4 A comparison of the memory footprints of parameters with different types. 59
4.5 A comparison of the memory footprint of a method invocation within the class and outside of it. 60
4.6 The estimated decrease in memory consumption of a Java program through method inlining. 62
4.7 The decrease in IMF if three classes are inlined. 70

List of Tables

2.1 Heap memory consumption of a Java field on HotSpot VMs in version 5 and 6. 17
2.2 Non–heap memory consumption of reference fields on HotSpot VM 6 in interpreted mode. 19
2.3 Non–heap memory consumption of int fields on HotSpot VM 6 in interpreted mode. 19

3.1 The instantiation time overhead (in ns) for subclasses with a varying inheritance depth and a wrapper class on a client–mode HotSpot 6. 25
3.2 The confidence intervals for the respective virtual machines. Sample 1 contains the measurements of creation time for an object with the depth of two and sample 2 contains the measurements of creation time for a refactored object. 26
3.3 The confidence intervals for the respective virtual machines. Sample 1 contains the measurements of method invocation on an object with the depth of two and sample 2 contains the measurements of method invocation on a refactored object. 28
3.4 The number of iterations needed to outweigh the overhead of object creation for the respective inheritance depth. 28
3.5 The number of iterations needed to outweigh the overhead of method invocation for the respective inheritance depth. 29
3.6 A comparison between the mean and the standard deviation (in ns) of the 1000 and 30,000 iteration benchmark for creation of an object with the depth of ten. 29
3.7 A comparison between the mean and the standard deviation (in ns) of the 1000 and 30,000 iteration benchmark for method invocation on an object with the depth of ten. 30

3.8 The mean performance benefit (in ns) due to inlining a varying number of methods in the 1000 iterations benchmark. 32

3.9 The *optimization breakpoints* for the HotSpot VMs with a JIT compiler. 33

3.10 The performance gain (in ns) due to *Inline Method* for a varying number of inlinings. 34

3.11 The confidence intervals for a varying number of method inlinings in the 30,000 iterations benchmark. 37

3.12 The creation time (in ns) of deep getter–setter objects and refactored (i.e., flattened) getter–setter objects. 39

3.13 Performance gain (in ns) due to flattening a Wrapper class with five parameters. Negative numbers mean that there is a performance degradation. 39

3.14 Performance improvement (in ns) due to flattening a MultiWrapper class. 41

3.15 The confidence intervals for the sets of measurements. 42

3.16 The *optimization breakpoints* for the HotSpot VMs with a JIT compiler. 42

3.17 The mean time (in ns) for creation of a MultiWrapper object with five parameters on a JRockit VM. 43

4.1 The increase in CMF (in byte) per inheritance level. 52

4.2 A comparison between the IMF (in byte) of the inheritance–based class instances and their composed counterparts. 54

4.3 The memory footprint (in byte) of a class with an increasing number of methods and parameters per method. 59

4.4 The memory footprint (in byte) per additional invocation instruction of the same method. 60

4.5 The memory footprint (in byte) of a method call instruction for a varying number of methods and parameters. 61

4.6 A comparison between the actual and the computed memory footprint decrease (in byte) with a varying number of inlined methods and parameters. 61

4.7 The memory footprint (in byte) of a field assignment on a HotSpot 6 VMIM. 64

4.8 The memory consumption (in byte) of a constructor with an increasing number of `Object` parameters on an interpreted–mode HotSpot VM 6. . . 67

4.9 Memory consumption (in byte) of a constructor with an increasing number of `Object` reference parameters on an interpreted–mode HotSpot VM 6. . 68

4.10 Memory consumption (in byte) of a constructor invocation using the keyword **new**. 69
4.11 A comparison between the actual and the computed memory footprint decrease (in byte) due to class inlining. 70

5.1 The lines of code (LOC) of the tested programs. 76
5.2 The results (in ms) before and after applying Inline Method for the Melbourne and the Bahrain benchmark. 78
5.3 The confidence intervals for the Melbourne benchmark results. 79
5.4 The confidence intervals for the Bahrain benchmark results. 79
5.5 The mean runtimes (in ms) of Eishay benchmark for the normal and refactored version of Kryo. Negative benefit percentages mean that the refactoring causes a performance degradation. 80
5.6 The confidence intervals for the sets of measurements of Kryo. 80
5.7 The PrettyTimeBench results in ms before and after applying method inlining. Negative benefit percentages mean that the refactoring causes a performance degradation. 81
5.8 The confidence intervals for the sets of measurements of PrettyTime. . . 81
5.9 The results for PolePosition (in ms) before and after applying Inline Class. Negative benefit percentages mean that the refactoring causes a performance degradation. 83
5.10 The confidence intervals for the Melbourne benchmark results. 83
5.11 The confidence intervals for the Bahrain benchmark results. 84
5.12 The mean runtimes (in ns) of Eishay benchmark for the normal and refactored version of Kryo. Negative benefit percentages mean that the refactoring causes a performance degradation. 84
5.13 The confidence intervals for the sets of measurements of Kryo. 84

List of Abbreviations

CMF	Class Memory Footprint
CPU	Central Processing Unit
CTI	Class to Inline
IDE	Integrated Development Environment
IMF	Instance Memory Footprint
JVM	Java Virtual Machine
KVM	K Virtual Machine
LOC	Lines of Code
MTI	Method to Inline
RAM	Random Access Memory
VM	Virtual Machine
VMCM	Virtual Machine in Client Mode
VMIM	Virtual Machine in Interpreted Mode
VMSM	Virtual Machine in Server Mode

Chapter 1

Introduction

Nowadays, one of the most common programming languages is Java [BW09]. A reason for its success is the reduction of the complexity of software development [Fla02]. Java offers numerous built–in features and comprehensive application programming interfaces (APIs) that simplify the development of software [HC08].

Therefore, the development of a runnable program with the desirable behavior is significantly easier using Java than in a language on a lower abstraction level. However, the circumstance that a program implements all required features and runs without an error does not mean that it is flawless. On the contrary, the program can contain a myriad of flaws—flaws in terms of software quality. For example, its source code can be convoluted or difficult to extend. To eliminate such shortcomings of a program while preserving its behavior, a software developer can use *refactoring*.

A *refactoring* is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [FB99]. For instance, we can refactor the Java expression **double** `y=x*x*x*x*x*x*x;` to the more catchy **double** `y=Math.pow(x, 7);`. The result of `y` remains the same, but the depiction of its computation changes.

The word *refactoring* was coined in the 1980s and refers explicitly to object–oriented software [MT04]. It was originally only used in the context of the Smalltalk programming language [FB99]. But in the 1990s, computer scientists transferred the concept of *refactoring* to other object–oriented languages, such as C++ and Java [Opd92][Rob99]. As a result, *refactoring* became a more common software engineering practice.

Eventually, the spread of agile software methodologies during the last decade bolstered the use of refactoring [PR01]. The *Agile Manifesto* (a paper that summarizes the main principles of agile software development) calls for the principle of "Responding to change over following a plan" [HF01]. Refactoring is a key tool to respond to this change in the requirements, because it enables changeability in object–oriented software [Ker04]. Thus, it is an integral part of the software development cycle of agile methodologies such as *Extreme Programming* [Bec99]. Concomitantly, the introduction of support for refactorings by the numerous integrated development environments (e.g., Eclipse, VisualStudio, and NetBeans) also brought about an easy way to refactor.

The wide–spread use of refactoring led to numerous scientific studies about its influence on non–functional properties (e.g., see [DB06], [Als09] and [LM06]). These studies examine the effect of refactoring on adaptability, maintainability, extensibility, reusabil-

ity, and readability. By contrast, the coherence between refactoring and performance–related properties has been a minor topic in computer science.

It is a prevalent opinion that refactoring has a bad influence on performance [MT04][FB99]. However, there are also several studies that indicate that refactoring can improve the performance of a program [Dem02][GR04] and decrease its memory footprint [WKK07]. Moreover, [TS05], [SKAP10], and [TD99] suggest the use of refactoring to improve the performance of software.

Particularly in a performance–critical application, even slight performance gains or degradations can be decisive for its success [HW10]. Accordingly, on devices with scare memory resources, an increase of the memory consumption in order of kilobytes can break the whole application. Consequently, we discuss in this thesis the influence of the refactorings *Replace Inheritance with Delegation*, *Inline Class*, and *Inline Method* on performance and memory footprint of a Java program. Moreover, we analyze whether their application on purpose is useful to improve the performance and to decrease the memory consumption.

1.1 Goals

In this thesis, we show how refactoring influences the performance and the memory footprint. The analysis answers two questions:

- Is it beneficial to use refactoring to improve the performance of a software program?
- Is it meaningful to use refactoring to reduce the memory footprint of a software program?

The contribution of this thesis is to answer both questions. For that purpose, we exemplify the influence of refactoring with three representants of Fowler's refactoring list [FB99].

First, we compare the performance of a non–refactored program to a refactored program in small scale (i.e., using micro—benchmarks). The benchmarking of these programs allows us to clarify whether the respective refactoring has a positive effect on performance at all.

Second, we compare the memory footprint of a non–refactored program to a refactored program in small scale. The comparison should mark a tendency whether a refactoring increases or decreases the memory footprint.

Finally, we conduct an evaluation to verify whether the reached results of the small–scale comparisons are also relevant in a larger scale by using case studies.

1.2 Structure

This thesis is structured as follows. In Chapter 2, we introduce refactoring and relate it to performance optimization. Furthermore, we present the analyzed refactorings, give an introduction to conducting benchmarks, and explain the experimental setup. In Chapter 3, we analyze the influence of refactoring on runtime performance. Chapter 4 investigates

the effect of refactoring on memory footprint. In Chapter 5, we evaluate the results of the micro—benchmarks from Chapter 3 with large–scale programs. In Chapter 6, we present related work of using refactorings to improve non–functional properties. Chapter 7 summarizes this thesis and shows future perspectives.

Chapter 2

Background

This chapter contains an introduction to refactoring, Java, benchmarking, statistics, and the memory footprint of a Java program. In the first part, we outline the important characteristics of refactoring and relate it to performance optimization. Moreover, we introduce the analyzed refactorings. Subsequently, we give a short introduction to the Java platform, which is necessary to understand the results of this thesis. In the third part, we describe how we conduct the benchmarks and describe the experimental setup. We give an introduction to statistical analysis in the fourth part. Finally, we present how we analyze the memory footprint of a Java program.

2.1 Refactoring

2.1.1 Definition

Today, the term *refactoring* applies to various contexts. Developers use it as an synonym for fixing bugs, adding new functionality, and so forth [McC04]. However, the actual meaning of refactoring is more limited.

The *Encyclopedia of Information Science and Technology* [KP08] defines refactoring as "A structural transformation that provides a systematic way of eradication the undesirable(s) from an artifact while preserving its behavioral semantics." In [FB99], it is stated as: "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." According to Beck, refactoring is "A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality" [Bec99].

All definitions emphasize that refactoring is a structural transformation that preserves the behavior of a program. This means that adding functionality is not refactoring. So, where does the ambiguity come from? Certainly, the lack of a strictly formal definition is one reason [PR01]. What means *easier to understand*? What is an *undesirable* in an artifact? The prevalent publications about refactoring (e.g., [Opd92], [Rob99], [FB99], and [Ker04]) do not give an answer to this question or just answer it partly. For instance, although Fowler gives in [FB99] advices to identify targets for refactoring (he calls them *bad smells*), he does not suggest a heuristic in a scientifically sound manner. In addition, he fails to propose metrics to measure the influence of refactoring on software quality

characteristics. Therefore, developers depend often on vague principles like "routines longer than a screen are rarely needed" [McC04] when they refactor their source codes.

Another left–open question in numerous publications about refactoring is how we can preserve the observable behavior of a program. As shown in [MT04], a formal proof is extremely difficult to conduct for complex languages such as Java. Therefore, [Bec99] suggests rigorous unit testing. The downside of this approach is that refactoring can break existing unit tests [Pip02][vDM02]. The obvious solution for this problem, namely to fix the broken test involves another question—how can we guarantee that the modified tests are correct?

Apparently, there is no final definition of refactoring. The lack of formality leaves room for interpretation. Nonetheless, for the understanding of this thesis it is important to know that:

- A refactoring is a transformation of an object–oriented software system [Rob99]. There are other types of refactoring (e.g., model refactoring), but they are not relevant for this thesis.
- A refactoring preserves the observable behavior of the software program [KP08]. It is the task of the developer to determine which (possibly domain–specific) requirements a refactoring must meet and how the behavior preservation can be guaranteed.
- A refactoring is used to improve a non–functional software property [Bec99]. There are no general specifications to measure the attainment of this goal. Consequently, the developer can decide either on his own or with the help of a software metric whether a refactoring was successful or not.

2.1.2 Performance Optimization

Is it valid to use refactoring to optimize the performance of programs? The opinions on this question vary. Some computer scientists argue that the only purpose of refactoring is the improvement of the internal software quality (i.e., quality characteristics that users of a software are not aware of such as testability and adaptability). Fowler states in [FB99] that "the purpose of refactoring is to make the software easier to understand and modify [..] Like refactoring, performance optimization does not usually change the behavior of a component [..] However, the purpose is different." This view is shared, among others, by [McC04]. Furthermore, Fowler even claims that "Refactoring certainly will make software go more slowly" [FB99].

Other computer scientists apply a broader definition of refactoring and use it also to improve some external software quality characteristics (i.e., characteristics that users of a software are aware of). Consequently, they see performance optimization as a possible goal of refactoring. For example, Beck defines refactoring as "A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality—[..], performance" [Bec99]. Accordingly, Kerievsky advices "if you do happen to experience performance problems, you can often refactor to improve performance" [Ker04].

As we show in Subsection 2.1.1 on the preceding page, the lack of a strictly formal definition of refactoring leaves room for interpretation. Therefore, we can decide on our

own whether it is valid to use refactoring as performance optimization or not. Throughout this thesis, we follow the view of Beck and his colleagues, and use refactoring with the additional purpose to improve the performance.

2.1.3 Analyzed Refactorings

We exemplify the influence of refactoring on performance and memory footprint by three refactorings from Fowler's list [FB99]. The used refactorings are *Replace Inheritance with Delegation*, *Inline Method*, and *Inline Class*. They are presented in the following.

Replace Inheritance with Delegation

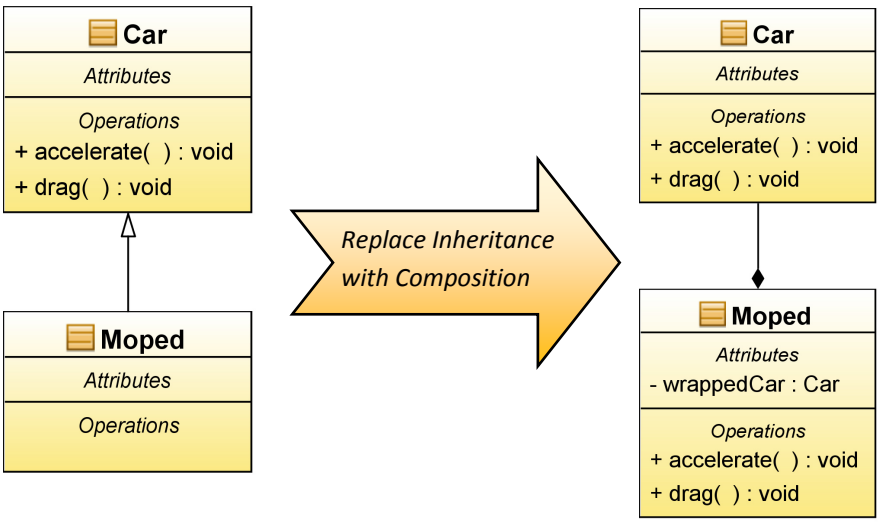

Figure 2.1: The `Moped` class is refactored by *Replace Inheritance with Delegation* to the wrapper `Moped` class.

Inheritance is a powerful tool to achieve code reuse, but it is not always appropriate. As [Sny86] outlines, inheritance violates encapsulation, because the subclass depends on the implementation details of its superclass for its proper function. As a result, the subclass is highly fragile. For instance, a subclass may break if the superclass changes, even though its source code has not been modified. We should thus apply the *Replace Inheritance with Delegation*[1] refactoring if the subclass has no genuine "is–a" relationship to its superclass. That is, instead of extending an existing class, we add a **private** field to the class that references to an instance of the former superclass. Subsequently, we add *forwarding methods* that have the same signature like the ones that the class inherited previously from its superclass. If we invoke a forwarding method, it delegates the call to the corresponding method of the contained instance (of the former superclass). In

[1]Be aware that the meaning of the term *delegation* used in this thesis is not equal to that one in [Lie86] and [GHJV95]. The reason is that our wrapper objects do not pass a reference to themselves to the wrapped objects.

Figure 2.1 on the previous page, we show an example. Although `Moped` and `Car` share some behavior, `Moped` is not a subtype of `Car`. Therefore, we refactor the `Moped` class (on the left–hand side of the figure) to the wrapper `Moped` class (on the right–hand side) that has a **private** field that references to an instance of `Car`. In this way, wrapper `Moped` can delegate calls to the methods `accelerate()` and `drag()` to the corresponding methods of `Car`.

Inline Method

The *Inline Method* refactoring puts the body of a method into the body of its caller and remove the method [FB99]. We can use it to resolve confusing indirection. For instance, if we have several methods that do nothing more than forwarding a method call, we can inline these methods and invoke directly the desired method. Additionally, we can also use method inlining to optimize our programs, because every method invocation introduces an overhead in Java [HC08]. Figure 2.2 shows an example. On the right–hand side, the method `calculate` invokes the `sum` method each time it is called. However, on the left–hand side, the `sum` method is inlined into the `calculate` method.

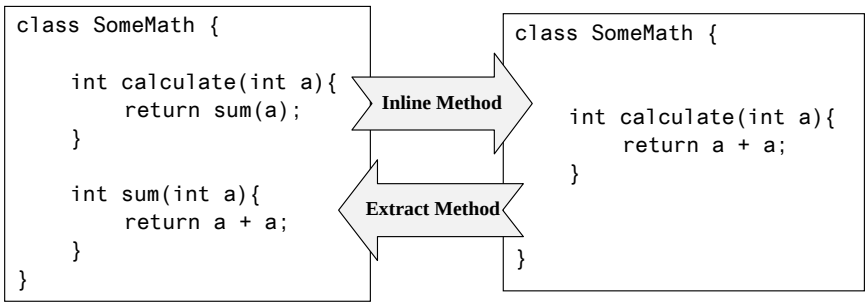

Figure 2.2: An example for method inlining.

Inline Class

If we inline a class, we move all fields and methods of one class into another class [FB99]. The refactoring is useful when two classes have (nearly) the same purpose (e.g., represent the same entity). Moreover, we can also inline a classes that is used by only a single class without difficulty.

In Figure 2.3 on the next page, we show an example. On the left–hand side of the figure, the classes `Person`, `Address`, `City`, and `Street` are self–contained. Each time we create an instance of the class `Person`, also objects of the other mentioned classes are created. On the right–hand side, we present the `Person` class after inlining the `Person`, `Address`, `City`, and `Street` into it.

2.2 Background Java

The purpose of Java is to provide an object–oriented and a platform–independent programming language. To reach the latter, a Java compiler does not directly compile the

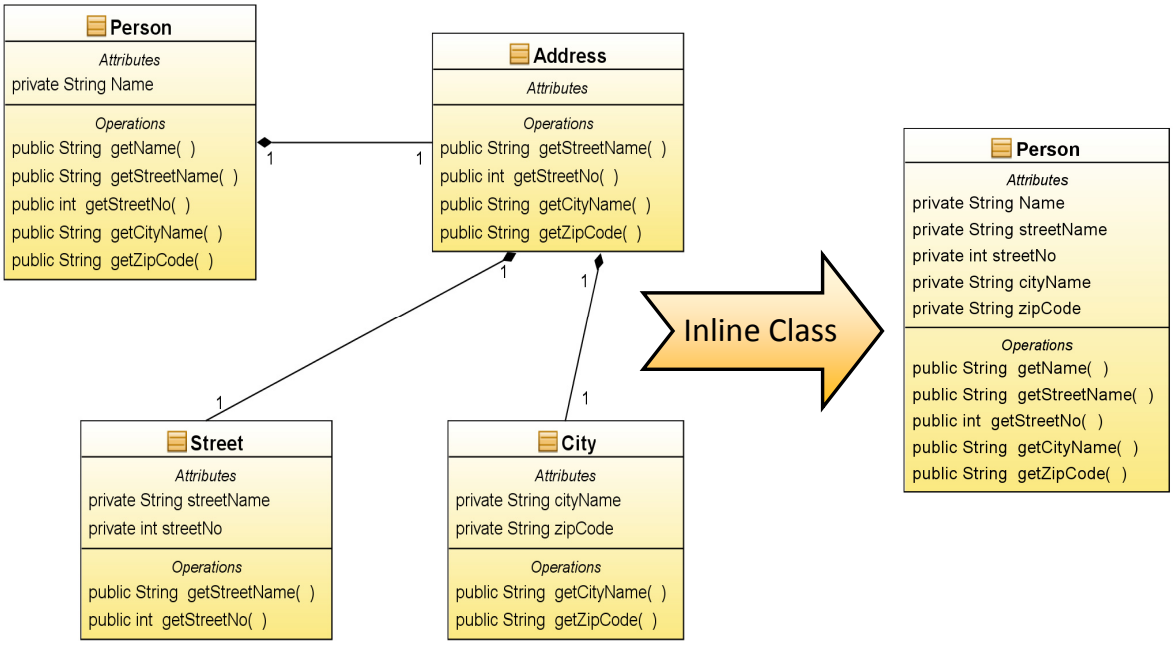

Figure 2.3: A class inlining example.

source code into native machine code. Instead, the compiler produces an intermediate form called *bytecode*. To execute the bytecode, we need a Java virtual machine (JVM)[2]. The Java virtual machine is itself a program running on top of a platform. Because the bytecode is platform independent, we can run it on any platform the JVM has been ported to.

2.2.1 JIT compiler

For this thesis, it is important to identify the influence of the JIT compilers. Their optimizations may affect considerably the results of our measurements and, thus, the effectiveness of the introduced refactorings.

Originally, the virtual machines only interpreted the bytecode. A program running on a first generation JVM was thus about 20 times slower than a comparable C program [Fla02]. The reason was that the system spent more time executing the interpreter than the program it was supposed to be running [Goe04]. Consequently, the next generation of JVMs compiled every path of bytecode into native machine code before it was about to be executed (i.e., just–in–time). However, this approach had drawbacks as well. The start–up time of a program was inadmissible (because the compilation of all bytecode to be executed introduced a significant overhead) and the level of optimization was mediocre [Goe04]. Therefore, current virtual machines use both an interpreter and a compiler. They immediately run the program using an interpreter and profile its execution. Then, the compiler is—due to the previously collected profiling data—capable to detect often used code paths (called *hot spots*) and generates optimized native code for them. How often a code path must be executed to be optimized is non–deterministic [GBE07]; for

[2]Notice that we use *virtual machine* (VM) as a synonym of *Java virtual machine* (JVM) throughout this thesis.

example, a piece of code is once optimized after it was executed 3000 times, and another time after 4007 executions.

Which optimizations a compiler of a Java virtual machine performs depends on its vendor, its version, and its configuration. The Sun Microsystems Java HotSpot VM 6 offers, for instance, two JIT–compiler modes—the client and the server mode. Depending on the mode, both the performed optimization and the moment of optimization vary [Sun05a].

2.3 Benchmarks and Experimental Setup

2.3.1 Micro—Benchmarks

Although there are numerous problems with micro—benchmarks [Goe05], this kind of benchmarks is necessary to get an outlook of the effect of a refactoring. The consequences of a slight code change are often not apparent or even counterintuitive. For example, Bentley discovered that eliminating code from loops in C can slow down their execution speed [Ben99].

To get meaningful measurements, we benchmark every program three times (with 1000, 10,000 and 30,000 iterations). We focus our analysis on the benchmark of 1000 iterations. The reason for this concentration on start–up performance is that it is not influenced through optimization by the JIT compiler.

Be aware that the use of the term *start–up performance* in this thesis is not equal to that one introduced in [GBE07]. That is, we do not include the start–up time of the virtual machine into our measurement. The first iteration after the start of the VM is significantly worse than the normal execution time. Consequently, we start the measurement only after the first iteration (the first iteration would be filtered out by the outlier test anyway). That is, the 1000 iterations benchmark, for example, consists of the iterations 2 to 1001.

All runtimes are measured with the method `System.nanoTime()`, because the call to the method `System.currentTimeMillis()` can have a granularity of 15 ms [Sun05b]. However, Sun Microsystems does not guarantee that `System.nanoTime()` is accurate [Sun04]. In addition, we found out that the measurement by `System.nanoTime()` is affected by the precedent and following expressions. For instance, let us consider Listing 2.1 on the facing page. Theoretically, the performance of the methods `sleepLess()`, `sleepAfter()`, and `sleepBefore()` should be nearly identical. In fact, the the execution of `sleepLess()` is about 30 percent faster than the executions of `sleepAfter()` and `sleepBefore()`.

2.3.2 Optimization breakpoint

To depict the influence of the JIT compiler, we determine the *optimization breakpoint* for each virtual machine other than interpreted–mode HotSpot VMs. The *optimization breakpoint* is the iteration number of an unoptimized code block that is needed so that the JIT compiler has tweaked the code block so significantly that its performance is comparable to a manually optimized (e.g., refactored) code block.

```java
public class SystemNanoTime {

    public static double SINUS;

    public long sleepLess() {
        long time = - System.nanoTime();
        SINUS = Math.sin(Math.PI);
        time += System.nanoTime();
        return time;
    }

    public long sleepBefore() {
        System.out.println(Math.sin(Math.pow(Math.cos(Math.E),
                                    111.333)));
        long time = - System.nanoTime();
        SINUS = Math.sin(Math.PI);
        time += System.nanoTime();
        return time;
    }

    public long sleepAfter() {
        long time = - System.nanoTime();
        SINUS = Math.sin(Math.PI);
        time += System.nanoTime();
        System.out.println(Math.sin(Math.pow(Math.cos(Math.E),
                                    111.333)));
        return time;
    }
}
```

Listing 2.1: An example how the results of System.nanoTime() can be adulaterated.

To determine the *optimization breakpoint*, we use the mean execution time of the refactored program in the 1000 iterations benchmark (notated as $\bar{x}_{r_{1000}}$). Then, we determine the iteration number of the first measurement in the 30,000 iterations benchmark of the normal program for that

$$x_i <= \bar{x}_{r_{1000}}$$

is true. Whereas, x is a measurement in the data set of the 30,000 iterations benchmark and i is the iteration number. For instance, $\bar{x}_{r_{1000}}$ is 1000 ns and x_{2844} is with 998 ns the first measurement that is less than or equal to $\bar{x}_{r_{1000}}$. The *optimization breakpoint* is thus at 2844 iterations. That is, after 2844 invocations of the normal code block, the JIT compiler has optimized it so significantly that the performance of the normal code block is comparable to its refactored version.

2.3.3 Used Java Virtual Machines and Hardware

The execution speed of a Java program depends not only on the hardware resources of the system executing it. Rather, its execution environment (i.e., the Java virtual machine)

has a considerable impact on its performance. Thereby, the version, the vendor, and the configuration of the virtual machine are important influencing variables.

To emphasize which influence the optimization of the just–in–time compiler has, we benchmark every program both in interpreted mode and in JIT–compiled mode. The measurements of the interpreted mode are important for applications that are not optimized by a JIT compiler (e.g., applications running on Google Android (until version 2.1) or K virtual machine). In reverse, the measurements for the virtual machines in JIT–compiled mode show until which number of iterations a refactoring has a performance benefit on a VM with JIT compiler. Moreover, we analyze the test programs on HotSpot VMs with diverse JIT compilers that perform different optimizations (see on page 9).

Beside the compilation mode, the version of the virtual machine has a significant influence on the execution speed of a program. For instance, [GPB+06] presents two Java locking mechanisms (locking by `ReentrantLock` and *instrinsic locking*) that show a significant performance difference on a HotSpot 5 VM. However, on a HotSpot VM in release 6, both locking mechanisms have nearly the same performance.

To compare the performance of the Sun Microsystems HotSpot VMs to the performance of a VM of another vendor, we conduct every benchmark also on an Oracle JRockit VM 6. The comparison may show that the effect of refactoring can vary from vendor to vendor. The achieved results are hence not necessarily true for other Java virtual machines such as SAP JVM, J9, and Azul VM.

To summarize, we perform all benchmarks with following virtual machines:

- Sun Microsystems Java HotSpot Client VM 1.6.0_21-b07 (abbreviated as HotSpot 6 VMCM)
- Sun Microsystems Java HotSpot Server VM 1.6.0_21-b07 (abbreviated as HotSpot 6 VMSM)
- Sun Microsystems Java HotSpot Client VM 1.6.0_21-b07 in interpreted mode (abbreviated as HotSpot 6 VMIM or HotSpot 6 (Xint))
- Sun Microsystems Java HotSpot Client VM 1.5.0_22-b03 (abbreviated as HotSpot 5 VMCM)
- Sun Microsystems Java HotSpot Server VM 1.5.0_22-b03 (abbreviated as HotSpot 5 VMSM)
- Sun Microsystems Java HotSpot Client VM 1.5.0_22-b03 in interpreted mode (abbreviated as HotSpot 5 VMIM or HotSpot 5 (Xint))
- Oracle JRockit 1.6.0_20-b02 (abbreviated as JRockit 6 VM)

All used virtual machines are the 32–bit versions. The memory defaults are an initial Java heap size of 1024 MB and a maximum Java heap size of 1536 MB for each VM.

All benchmarks were run on a machine with an Intel Core 2 Duo T9400 (2,53 GHz) processor, 3 GB RAM, and Windows Vista Home Premium (32–bit) as the operating system.

2.4 Statistical Analysis

To draw sound conclusions from the benchmarks, we statistically analyze the samples. A sample contains the measured values of a benchmark on a virtual machine. This means, the sample of the JRockit 1000 iterations benchmark contains of 1000 measured values for the respective benchmark program on the JRockit VM. In fact, it may contain less values, because we filter the outliers (see on page 15).

2.4.1 Descriptive Statistic

To analyze the measurements, we compute three statistical quantities—arithmetic mean, variance, and standard deviation.

Arithmetic Mean

The arithmetic mean ($\bar{x}$) provides information about the average runtime of the members of a sample. It is computed as the sum of the particular values of the data set (x_i) divided by the number of measurements (n):

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n}(x_i)$$

Dispersion

For all samples, we calculate the variance and the standard deviation. Both describe how much the individual values in a data set vary from the mean value. The variance is, for instance, a variable needed for calculating the confidence intervals.

Variance To calculate the variance of a sample (notated as s^2), it is necessary to calculate the squared deviation of all measurements of our data set from the mean. Subsequently, we divide their sum by $n-1$ (n represents the number of measurements and -1 is necessary because we lose one degree of freedom when we calculate the mean [BW08]). We thus get the formula:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Standard Deviation In calculating the variance, we have changed from our original units to squared units, which are not very convenient to interpret [Bor99]. To get back to the original units, we take the square root of the variance. Consequently, the formula for the sample standard deviation is:

$$s = \sqrt{s^2} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

In the benchmarks, we use the standard deviation to draw conclusions about optimizations by the JIT compiler. For instance, if the standard deviation is 40 in the 1000 iterations benchmark and 210 in the 10,000 iterations benchmark, we can conclude that the JIT compiler optimizes between the 1000th and the 10,000th iteration.

2.4.2 Inferential Statistic

Theses

To draw conclusions out of the benchmarks in a statistically sound manner, we develop a research hypothesis for each refactoring. For that purpose, we state a null hypothesis H_0 and an alternative hypothesis H_1. Both are mutually exclusive and exhaustive [BW08]. Throughout this thesis, the null hypothesis is always negative (e.g., H_0: *Replace Parameter with Method* does not improve the runtime performance). Because we test the effect of a refactoring on seven virtual machines, we also have to introduce at least seven hypotheses per refactoring. More hypotheses can be necessary if we test the effect of a refactoring on different performance properties such as object creation time and method invocation time.

To determine whether to accept or to reject a null hypothesis, we conduct a significance test (see on the current page). A significance test is sufficient, because we want to compare only two alternatives—the non–refactored alternative and the refactored one.

Let us consider an example. We want to compare whether the refactoring *Replace Parameter with Method* improves the performance of our program. Consequently, we state the null and the alternative hypotheses:

- H_0: *Replace Parameter with Method* does not improve the runtime performance.
- H_1: *Replace Parameter with Method* does improve the runtime performance.

Subsequently, we benchmark the non–refactored and the refactored program. Due to the significance test, we can determine whether the differences in the performances of both alternatives is statically significant. Moreover, the comparison of the benchmark means show us whether the refactored alternative is faster or slower than its non–refactored counterpart. If the difference is statistically significant and the performance of the refactored version is improved, we must reject the null hypothesis. That is, the refactoring *Replace Parameter with Method* improves the runtime performance.

However, there is still a probability of α that the null hypothesis is true, but rejected by mistake in our study. α is also known as Type we error. Throughout this thesis, α is set at 0.03 (as suggested in [SGCM05]).

Significance Test

To determine whether the differences in the runtimes of a non–refactored and a refactored program are statistically significant, we have to compare the confidence intervals of both alternatives. If they do overlap, we can conclude that the differences in the mean values are results of random fluctuation. However, we do not need to compute the confidence intervals for each of them; we can immediately compute a combined confidence interval for both and draw our conclusion based on it [GBE07].

Let us assume that the measurements of the non–refactored program are Alternative 1 with a measurement number of n_1, an arithmetic mean of $\bar{x_1}$ and a standard deviation of s_1. Consequently, the measurements of the refactored program are Alternative 2 with a measurement number of n_2, an arithmetic mean of $\bar{x_2}$ and a standard deviation of s_2.

To figure out whether the two alternatives are statistically significantly different, we have to compute the difference of their arithmetic means as:

$$\bar{x} = \bar{x_1} - \bar{x_2}$$

and, subsequently, the standard deviation of the difference of the mean values as:

$$s_x = \sqrt{\frac{{s_1}^2}{n_1} + \frac{{s_2}^2}{n_2}}$$

The value $z_{1-\alpha/2}$ is defined such that a random variable Z that is Gaussian distributed with a mean of null ($\mu = 0$) and a variance of one ($\sigma^2 = 1$), obeys following property:

$$Pr[Z \leq z_{1-\alpha/2}] = 1 - a/2$$

Because we use a significance level of 0.03, the value $z_{1-\alpha/2}$ is 2.17.

Finally, we can compute a confidence interval for the difference of the means by

$$c_1 = \overline{x} - z_{1-\alpha/2} s_x$$

$$c_2 = \overline{x} + z_{1-\alpha/2} s_x$$

If this confidence interval includes zero, we can conclude at the chosen confidence level that there is no statistically significant difference between the two alternatives.

Because our outlier test filters only the 10 percent of the highest values (see Section 2.4.2, the measurement number of both Alternative 1 and Alternative 2 is at least 900.

Outlier Detection

There are several reasons that can cause outliers such as background operating system activity, garbage collection of the JVM, and other unwanted computer processes. Especially for samples with a relatively low number of values, outliers can significantly adulterate the statistical quantities. Therefore, [GBE07] suggests to remove the outliers before analyzing the benchmarks. For this thesis, the filtering is done by the following method:

1. We sort all values of a sample S.
2. We split the sorted values into two samples. The first sample (called S_n) contains the 90 percent lowest values of S. The second sample S_o contains the remaining values. That is, S_o contains those values that are possible outliers. (The creation of three samples is not necessary, because we want to filter only values that are abnormally high.)
3. In this step, we test whether a value is an outlier. Since S_o contains multiple values (e.g., 100 for the 1000 iterations benchmark), the step is repeated several times. (Only if all values of S_o are outliers, we do not repeat it.)
   (a) We set the value to test (called x_o) to the lowest value of S_o.
   (b) We merge x_o and the values of S_n into a new sample S_t.

(c) We compute the standard deviations for S_n and S_t. The computed standard deviations are called s_n (for sample S_n) and s_t, respectively.

(d) To test whether the modification of the standard deviation through x_i is acceptable, we have to calculate the *standard deviation distance* (notated as θ) by:

$$\theta = \frac{\ln (s_r)^3}{\sqrt{n_r}}$$

Whereas n_r represents the number of values in S_r. The square root of n_r accommodates the fact that the influence of x_o decreases with an increasing number of values in S_r. The cubed *logarithmus naturalis* of s_r has a balancing effect. That is, it increases low standard deviations and decreases high ones.

(e) Finally, we can determine whether x_o is an outlier by the inequality:

$$s_r > s_t + \theta$$

If the inequation is true, than x_o is an outlier. Thus, we can end the testing of our values and the current S_n is the final and filtered sample.

On the contrary, if the inequation is false, x_o is no outlier. Therefore, we add it to the sample S_n and, in reverse, eliminate it from sample S_o. Subsequently, we start from (a) again. If there is no more value left in S_o, there were no outliers in our initial data set S.

2.5 Memory Footprint

The JVM has two main memory areas: the heap and the non–heap area. The memory footprint of a Java program contains of its memory consumption[3] in both areas.

Within the heap all runtime data for the class instances and arrays are allocated. It is created during the Java virtual machine start—up. If the free memory of the heap is below a determined threshold, an automatic memory management system (called garbage collector) reclaims memory.

In addition, the Java virtual machine has a method area that is shared among all threads. The method area belongs to non–heap memory. It stores per class structures such as a runtime constant pool, field and method data, and the code for methods and constructors. The method area is logically part of the heap of the JVM. Similar to the heap, the method area may be of a fixed size or may be expanded and reduced.

Although the partitioning of the random access memory (RAM) in heap and non–heap is the case for all virtual machines, the actual memory footprint of a class and its instances may vary. The reason is that the *Java Language Specification* does not specify the memory consumption of objects, primitive data types, instructions and so forth [LY99]. Consequently, it depends on the JVM implementation which memory footprint a Java program has.

The footprint of classes and their instances is particularly important for devices with limited memory resources. Such devices have mostly a virtual machine without a JIT

[3]Throughout this thesis, we use *memory consumption* as synonym for *memory footprint*.

compiler (such as K virtual machines). Thus, we disregard the profiling data for the JIT compiler and analyze the influence of refactoring on memory footprint on a HotSpot VM 6.0 in interpreted mode.

According to the two memory areas of the JVM, we examine the influence of the refactorings on the class memory footprint (CMF) and the instance memory footprint (IMF) of a Java program. Because the IMF is not subject to fluctuations, we compute it. That is, we only measure the CMF in the benchmarks.

2.5.1 Instance Memory Footprint

The instance memory footprint of a class reflects the memory footprint of its instances. It is the allocated memory for the instances of a class in the heap area of the JVM at a time. On the HotSpot VMIM 6, each Java object consumes at least 8 byte. Its fields consume additional memory. Table 2.1 shows the memory consumption of the instance fields. For example, the IMF of a class containing two double fields is 24 byte for one class instance, 48 byte for two instances, and 72 byte for three instances. Consequently, the IMF depends on the number of concurrently existing class instances. We can compute the IMF (notated as I) for HotSpot VM in version 5 and 6 by:

$$I = n_I * (8\,\text{byte} + 1\,\text{byte} * (B + Z) + 2\,\text{byte} * (C + S) + 4\,\text{byte} * (I + F) + 8\,\text{byte} * (D + J))$$

Whereas n_I is the number of class instances, B is the number of **byte** fields, C is the number of **char** fields, D is the number of **double** fields, F is the number of **float** fields, I is the number of **int** fields, J is the number of **long** fields, L is the number of reference fields, S is the number of **short** fields, and Z is the number of **boolean** fields.

Java field type	Memory footprint in byte
boolean	1
byte	1
char	2
double	8
float	4
int	4
long	8
Reference	4
short	2

Table 2.1: Heap memory consumption of a Java field on HotSpot VMs in version 5 and 6.

If we compute the IMF of a class applying the formula, we have to be aware that I must always be a multiple of eight. That is, if the computed IMF is not a multiple of eight, we must round up it to the next multiple of eight:

$$\frac{I}{8} \notin Z => I = I + 8 - I \bmod 8$$

The computed I is the swallow memory footprint of the objects. It comprises only the memory consumption of the fields that reference to instances of other classes (notated

as L) and not the memory footprint of the referenced instances themselves. In reversal, the deep memory footprint of an object comprises also of the memory footprint of the objects that it references to, the memory footprint of the object that the referenced objects points to, and so on.

In addition, there are objects for which footprints cannot be computed by the formula above. These objects are called *flyweights*, because they are implementations of the *Flyweight Pattern* described in [GHJV95]. For instance, `java.lang.String` implements the flyweight pattern [Coo00].

2.5.2 Class Memory Footprint

The class memory footprint of a class is the memory that a class requires in the non–heap area of the JVM. Among other things, the CMF of a class comprises of the memory footprint of the field and method names as well as the code for the methods. The CMF is independent of the class instance number. That is, the memory for the class is allocated only once when an instance of the class is initiated for the first time. Although the CMF does not fluctuate, the computation of the CMF is not straightforward. For instance, if a constant of a class (such as a method name or a number) already exists in the runtime constant pool of the JVM, it requires no additional memory. This means that the CMF of a class can be influenced by classes that are loaded beforehand into the non–heap area of the JVM.

A prediction of the order in which the classes are loaded into the non–heap area of the JVM is difficult or impossible at all. Consequently, we develop a formula for each presented refactoring to estimate its influence on the CMF. The example calculations showed that using the median memory consumption as an indicator causes too optimistic results with respect to the decrease in memory consumption. Moreover, it is significantly influenced by padding. As a result, we chose a pessimistic approach. For instance, Table 2.2 on the next page shows the memory footprint of reference fields in the non–heap area of the JVM. If we use the computed average memory consumption of a field, the elimination of a field reduces the CMF by about 35 byte. However, we take the lowest measured value. That is, we assume that the elimination of a reference field reduces the CMF by 24 byte. Accordingly, an **int** field requires at least 24 byte (see Table 2.3 on the facing page).

To determine the memory consumption of a class, we measure the non–heap consumption of the JVM before and after the class is loaded. For this purpose, we make use of the `getNonHeapMemoryUsage()` method in the `java.lang.management.MemoryMXBean` class. In Listing 2.2 on page 20, we show an example.

In our benchmarks, we try to foreclose that constants of the benchmark program already exists (e.g., method names) in the runtime constant pool of the JVM. For that purpose, we start a new instance of the JVM for each benchmark.

To minimize the influence of Strings on the CMF, all method, field, and class names in the benchmark programs contain three characters. The package names consists always of 37 characters.

Number fields	CMF in byte	Memory footprint per parameter in byte
0	928	-
1	976	48
2	1,000	24
3	1,040	40
4	1,080	40
5	1,120	40
6	1,144	24
7	1,184	40
8	1,216	32
9	1,256	40
10	1,280	24

Table 2.2: Non–heap memory consumption of reference fields on HotSpot VM 6 in interpreted mode.

Number fields	CMF in byte	Memory footprint per field in byte
0	928	-
1	968	40
2	992	24
3	1032	40
4	1072	40
5	1112	40
6	1136	24
7	1176	40
8	1208	32
9	1248	40
10	1272	24

Table 2.3: Non–heap memory consumption of int fields on HotSpot VM 6 in interpreted mode.

2.5.3 Hypotheses

We cannot provide hypotheses for the influence of refactorings on memory footprint. The reason is that the measurements of memory footprint are not subject to fluctuations. The heap memory consumption of an instance is constant. Moreover, the non–heap memory footprint of a class is constant if the measurements conditions are unaltered (i.e., the runtime constant pool is in the same state). Consequently, there is no standard deviation and each difference in measurement is statistically significant.

```
import java.lang.management.ManagementFactory;
import java.lang.management.MemoryMXBean;

public class NonHeapMeasurement {
    static MemoryMXBean mxBean =
              ManagementFactory.getMemoryMXBean();
    public static void main(String[] args) {
        long nonHeap;

        //prevents MXBean from adulterating the result:
        mxBean.getHeapMemoryUsage().getUsed();

        nonHeap   = -mxBean.getNonHeapMemoryUsage().getUsed();

        //class of which the CMF should be measured:
        new ClassToMeasure();

        nonHeap += mxBean.getNonHeapMemoryUsage().getUsed();
        System.out.println(nonHeap);
    }
}
```

Listing 2.2: The measurement of the non–heap memory footprint.

Chapter 3

Performance

In this chapter, we analyze the influence of refactoring on runtime performance. For that purpose, we exemplify the effect of refactoring on object creation time and method invocation time with the refactorings *Inline Class*, *Inline Method*, and *Replace Inheritance with Delegation*. Our assumptions are that:

1. *Replace Inheritance with Delegation* improves the performance, because it removes overhead that is caused by inheritance.
2. *Inline Method* improves the performance, because it reduces the number of method invocations.
3. *Inline Class* improves the performance, because it lessens the number of object allocations.

To analyze the assumptions in a statistically sound manner, we conduct a significance test for each analyzed refactoring.

3.1 Replace Inheritance with Delegation

Several publications [CL01][Shi03] indicate that inheritance introduces a performance overhead. In this section, we analyze thus whether applying the *Replace Inheritance with Delegation* refactoring can improve the performance.

3.1.1 Methodology

The analysis is two–folded. First, we compare the object creation speed between a composed (i.e., a wrapper) and a subtype object. For that purpose, we create a class with a one parameter constructor (called `BasisClass`). If we invoke the constructor, it computes `Math.sin(value*Math.PI) * Math.cos(value*Math.E)` , whereas `value` is passed as a **double** parameter. Subsequently, we create subclasses of `BasisClass` having constructors that do nothing else than invoke the constructor of its superclass. That is, we create an *inheritance chain*. For instance, the constructor of `SubClass2` invokes the constructor of `SubClass1`, and the constructor of `SubClass1` calls the constructor of `BasisClass`. We measure the creation time for the inheritance depths of 2, 5, 10, 30, and 50. In Java,

every class inherits from the `java.lang.Object` class [GJSB05], consequently the class `SubClass1` (extending `BasisClass`) has an inheritance depth of two. Then, we create a `WrapperClass` with a **private** field that references to an instance of `BasisClass`. The field is initialized when the constructor of the `WrapperClass` is invoked.

Second, we add the method **public void** `calculate(`**double** `value)` to all classes created in step one. Only the `calculate` method in the `BasisClass` computes the value (by `Math.sin(value*Math.PI) * Math.cos(value*Math.E)`). The methods in the subclasses and in the wrapper classes delegate the call to the corresponding method of their ancestors and, respectively, their wrapped instances of `BasisClass`. We measure the invocation speed of the methods on objects with an inheritance depth of 2, 5, 10, 30, and 50. In addition, we measure the method invocation speed on the wrapper instances. Afterwards, we compare the measured times of the subclasses and the wrapper classes.

3.1.2 Hypotheses

The first purpose of this section is to determine whether *Replace Inheritance with Delegation* does speed up the object creation performance. To focus our study, we set up the following hypotheses. For each hypothesis, H_0 represents the null hypothesis and H_1 represents the alternative hypothesis of the null hypothesis:

- Hypothesis A1

  H_0: For HotSpot 6.0 in client mode, *Replace Inheritance with Delegation* does not speed up the object creation performance.

  H_1: For HotSpot 6.0 in client mode, *Replace Inheritance with Delegation* does speed up the object creation performance.

The hypotheses are accordant for HotSpot 6 VMSM (A2), HotSpot 6 VMIM (A3), HotSpot 5 VMCM (A4), HotSpot 5 VMSM (A5), HotSpot 5 VMIM (A6), and JRockit 6 VM (A7).

The second purpose of this section is to determine whether *Replace Inheritance with Delegation* does speed up the method invocation performance. To focus our study, we set up the following hypotheses. For each hypothesis, H_0 represents the null hypothesis and H_1 represents the alternative hypothesis of the null hypothesis:

- Hypothesis B1

  H_0: For HotSpot 6.0 in client mode, *Replace Inheritance with Delegation* does not speed up the method invocation performance.

  H_1: For HotSpot 6.0 in client mode, *Replace Inheritance with Delegation* does speed up the method invocation performance.

The hypotheses are accordant for HotSpot 6 VMSM (B2), HotSpot 6 VMIM (B3), HotSpot 5 VMCM (B4), HotSpot 5 VMSM (B5), HotSpot 5 VMIM (B6), and JRockit 6 VM (B7).

3.1.3 Results

Object Creation

Figure 3.1 shows that the creation of an object with a depth of two is slightly (about 1.6 percent) faster than the creation of a wrapper object on a client–mode HotSpot VM 6. However, the creation time advantage of subtype classes is only true for a low inheritance depth, because every level of inheritance increases the creation time of an instance. If we take into account the measurements until an inheritance depth of 50, we identify that *each* inheritance level increases approximately linearly the creation time of an object. On a client–mode HotSpot VM 6, the increase is about 30.4 ns per inheritance level.

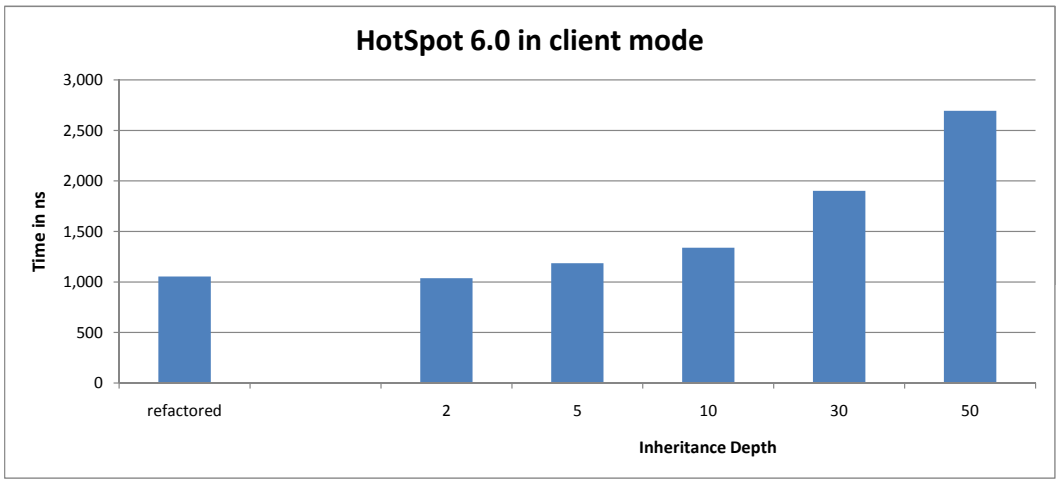

Figure 3.1: A comparison of the mean object creation times for a (refactored) wrapper object and objects with a varying inheritance depth on a client–mode HotSpot VM 6.

The other virtual machines also show a nearly linear increase in object creation time for additional levels of inheritance (see Figure 3.2 on the next page). On HotSpot VMs, the increase per inheritance level reaches from 20.9 ns for version 5 in interpreted mode to 31.9 ns for version 6 in interpreted mode. The performance penalty per level of inheritance is significantly less on a JRockit VM—it is just 5.3 ns. Depending on the overhead per inheritance level, a considerable degradation of object creation performance can occur. For our benchmark program on HotSpot VMs, the creation of an object with a depth of 50 is from 132.6 to 172.9 percent more expensive than the creation of an object with a depth of two. Because the overhead per inheritance level is significantly smaller on a JRockit VM, 48 additional levels of inheritance increase the creation time just by 44.2 percent on this virtual machine.

Apparently, a high number of inheritance levels decreases the object creation performance. However, inheritance depths greater than ten levels are rare [Kab06]. Consequently, we focus the comparison between the refactored (i.e., wrapper) and non–refactored (i.e., subtype) classes on inheritance depths of up to ten levels. In the beginning of this section, we already emphasized that the creation time of a wrapper class is slightly slower than the creation of an object with an inheritance depth of two for a client–mode HotSpot 6 VM. We can observe this circumstance also on all tested virtual machines other than HotSpot 5 in server–mode. Figure 3.3 on the following page depicts

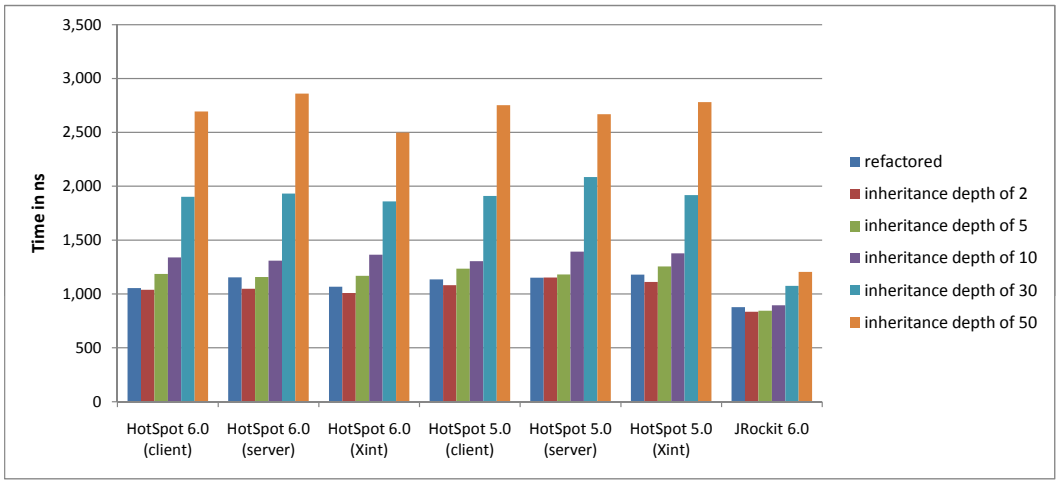

Figure 3.2: A comparison of the mean object creation times for a wrapper object and objects with a varying inheritance depth on different virtual machines.

that the creation of an object with an inheritance depth of two is for six of seven virtual machines faster than the creation of the refactored wrapper object. The speed benefit is 1.7 to 10.1 percent.

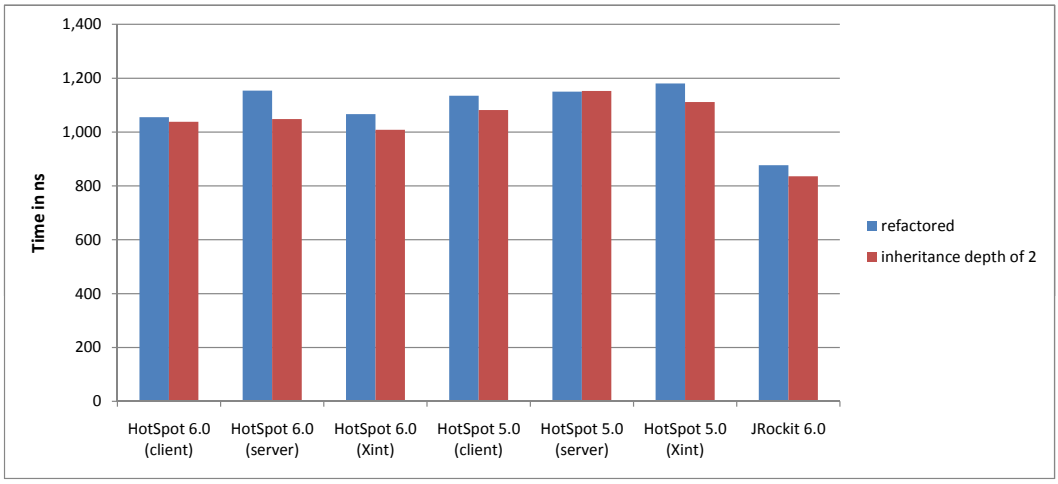

Figure 3.3: A comparison of the mean times for object creation for a wrapper object and objects with an inheritance depth of two.

For objects with an inheritance depth of five, the situation is different. In this case, the creation of a wrapper object is faster for all HotSpot VMs. The performance benefit is on average 6.1 percent. Only on the JRockit VM 6, the creation of a wrapper object is still 3.9 percent slower than the creation of an object with the depth of five. For an inheritance depth of ten the creation of an instance of the wrapper class is faster on all tested virtual machines. In this case, the performance benefit of wrapper object is in average 16.6 percent for the HotSpot VMs and 2.0 percent for the JRockit VM 6.

Let us consider an example. We want to calculate whether the refactoring *Replace Inheritance with Delegation* would cause a creation time benefit for our subclass. The inheritance depth is called d. The overhead per inheritance level is notated as l. Conse-

quently, we can compute the total overhead of inheritance i by:

$$i = (d - 1) * l$$

The reason for $d - 1$ is that every Java class is a child of the `java.lang.Object` class. The overhead for creating a wrapper object is w. Therefore, *Replace Inheritance with Delegation* speeds up the object creation if:

$$w < i$$

If we apply the mean values for HotSpot 6 VMCM, we can compute the overhead per inheritance level by:

$$i = (d - 1) * 30.4\,\text{ns}$$

for this VM. Moreover, the time difference between creating the `BasisClass` and creating the `WrapperClass` (i.e., w) is about 123 ns.

Based on the derived formula, Table 3.1 shows the instance creation overhead for subclasses with a varying number of inheritance levels and the instance creation overhead for a wrapper class. It depicts that *Replace Inheritance with Delegation* speeds up the object creation time beginning from an inheritance depth of five. That is, the *Replace Inheritance with Delegation* refactoring is not beneficial to improve the instance creation times of classes with an inheritance depth smaller than five on a HotSpot 6 VMCM. In reversal, the figures indicate that the replacement of delegation with inheritance can be meaningful in these cases.

For instance, the refactoring of a `WrapperClass` to a `SubClass1` could reduce its instantiation time by about 92.6 ns. An example calculation shows that this performance benefit is slight: the instantiation of two million `SubClass1` classes would only save 185.2 ms compared to the creation of `WrapperClass` instances.

Inheritance depth	Subclass	Wrapper Class
2	30.4	123
3	60.8	123
4	91.2	123
5	121.6	123
6	152.0	123
7	182.4	123
8	212.8	123

Table 3.1: The instantiation time overhead (in ns) for subclasses with a varying inheritance depth and a wrapper class on a client–mode HotSpot 6.

Finally, we test our Theses A1 to A7. The results achieved so far show that the *Replace Inheritance with Delegation* refactoring does not generally speed up the object creation time. We conduct a significance test to statistically verify this observation. Table 3.2 on the following page shows that the difference (i.e., $\bar{x}$) between the mean object creation of an object with an inheritance depth of two and a refactored object is negative for six of seven virtual machines (i.e., the non–refactored subtype object is faster to create). Moreover, in the cases where the subtype object is faster to create, the results are statistically significant. In contrast, the creation time of a subtype object on a

JVM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	-17.2	2.5	-11.8	-22.6
HotSpot 6 (server)	-105.9	4.4	-96.3	-115.5
HotSpot 6 (Xint)	-58.5	2.0	-54.2	-62.8
HotSpot 5 (client)	-52.8	1.7	-49.0	-56.6
HotSpot 5 (server)	2.1	1.7	5.8	-1.6
HotSpot 5 (Xint)	-68.5	2.2	-63.7	-73.3
JRockit 6	-41.4	1.9	-37.2	-45.5

Table 3.2: The confidence intervals for the respective virtual machines. Sample 1 contains the measurements of creation time for an object with the depth of two and sample 2 contains the measurements of creation time for a refactored object.

server–mode HotSpot 5.0 is neither faster nor statistically significant. As a consequence, we must accept the null hypotheses of Thesis A1 to A7.

Nevertheless, we have to emphasize that *Replace Inheritance with Delegation* can improve the object creation time for higher inheritance depths, because the class instantiation overhead increases approximately linearly per additional inheritance level. How many inheritance levels are necessary to reach a speed benefit for the refactoring depends on the virtual machine and the given hardware.

Method Invocation

Figure 3.4 exhibits that the measurements for method invocation on a client–mode HotSpot 6 are very similar to those for the object creation time. This means that: (1) method invocation is slightly faster (about 52.7 ns) on a subclass instance with the depth of two than on a wrapper object, and (2) the method invocation time increases nearly linearly per additional level of inheritance. On a HotSpot 6 VMCM, it increases by about 30.5 ns per inheritance level.

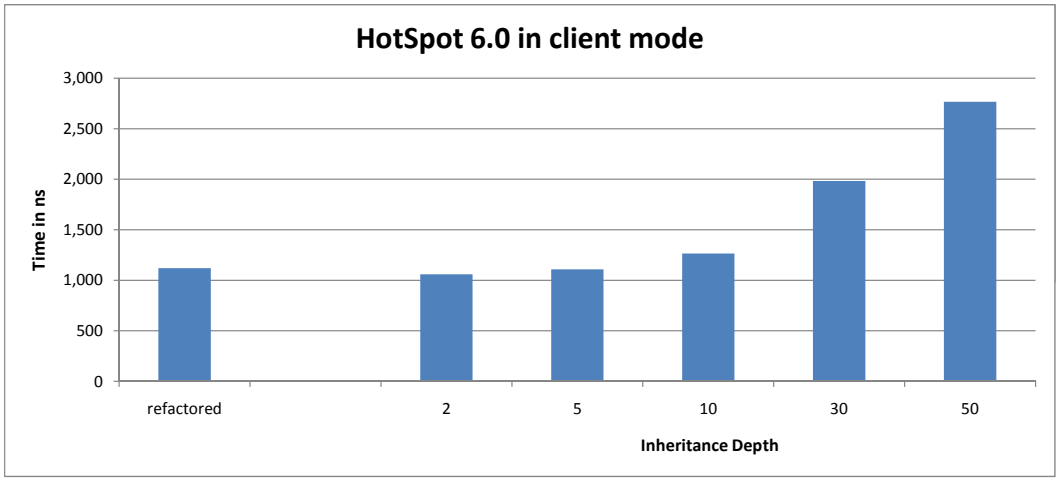

Figure 3.4: A comparison of the mean mehtod invocation times on a composed object and objects with a varying inheritance depth on a client–mode HotSpot 6.

Figure 3.5 on the facing page shows that these observations are not exclusively true

for client–mode HotSpot VM 6. Rather, they are valid for all virtual machines. On HotSpot VMs, the overhead of method invocation per inheritance level is 18.8 ns (for version 6 in server mode) to 25.1 ns (for version 5 in client–mode) and 21.8 ns on average. By contrast, the overhead of method invocation is just 2.2 ns per inheritance level on a JRockit 6 VM.

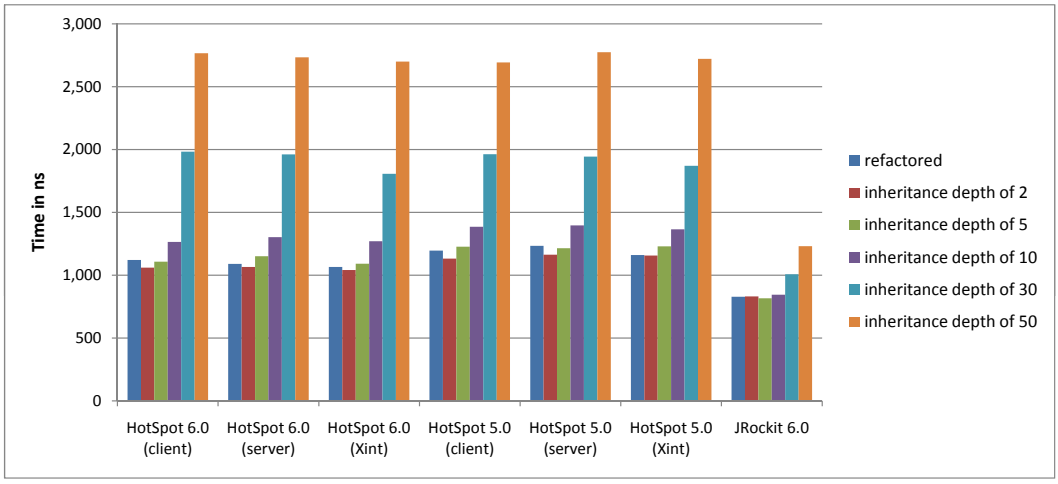

Figure 3.5: A comparison of the mean method invocation times for a composed (i.e., refactored) object and objects with a varying inheritance depth on different virtual machines.

A crucial condition that the method invocation overhead is really as large as depicted in Figure 3.5 is that the invoked method is located in the `BasisClass`. For instance, if `Subclass5` overwrote the `calculate` method, the invocation of the `calculate` method on an instance of `SubClass5` would have nearly the same performance as the invocation of the `calculate` method on an instance of `BasisClass`. The method invocation performance does not depend on the inheritance depth; it depends on where the used implementation of the method is located. Consequently, for determining the method invocation overhead it is important to figure out how many inheritance levels are between the class on that the method is invoked and the class that provide the used method implementation (notated as k).

In general, we can estimate method invocation overhead by:

$$u = k * r$$

Whereas r is the method invocation overhead per inheritance level. Since the implementation of `calculate` is located in the `BasisClass` (that has an inheritance depth of one), we can replace k by $d - 1$ (d is the inheritance depth of the class for that we want to estimate the method invocation overhead). The overhead for a method invocation on an instance of a wrapper class is m. *Replace Inheritance with Delegation* speeds up therefore the method invocation if:

$$m < u$$

The results are very similar to those for the influence on object creation speed. Accordingly, we have to accept the the null hypotheses of Theses B1 to B7. In Table 3.3, we present the confidence intervals for the various virtual machines. The difference is

statistically significant for all HotSpot VMs other than version 5 in client mode. For JRockit VM 6, we can observe that the creation of a wrapper object is slightly faster (the difference in method invocation speed is 2.5 ns). However, the difference in performance between the refactored and non–refactored version is not statistically significant. Therefore, the general claim that *Replace Inheritance with Delegation* speeds up the method invocation time is not true.

JVM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	-61.1	2.0	-56.7	-65.5
HotSpot 6 (server)	-24.6	1.8	-20.7	-28.5
HotSpot 6 (Xint)	-24.8	1.7	-21.2	-28.4
HotSpot 5 (client)	-64.2	2.2	-59.4	-69.0
HotSpot 5 (server)	-70.5	2.3	-65.6	-75.5
HotSpot 5 (Xint)	-3.3	1.9	0.8	-7.5
JRockit 6	2.5	2.5	8.0	-3.0

Table 3.3: The confidence intervals for the respective virtual machines. Sample 1 contains the measurements of method invocation on an object with the depth of two and sample 2 contains the measurements of method invocation on a refactored object.

3.1.4 Influence of the JIT compiler

Table 3.4 depicts that the JIT compilers of the HotSpot VMs can outweigh the performance penalty for creation of a subtype object after a varying amount of iterations. The client–mode JIT compilers optimize in seven of eight tested cases earlier than their counterparts in server–mode. For instance, the server–mode HotSpot VM 6 needs averagely 10,194 iterations to outweigh the overhead. In contrast, the HotSpot VM 6 in client–mode can outweigh it after 2382 iterations on average.

Inheritance Depth	HotSpot 6.0 (client)	HotSpot 6.0 (server)	HotSpot 5.0 (client)	HotSpot 5.0 (server)
5	1,555	10,821	1,495	11,545
10	1,493	9,982	1,497	11,359
30	2,769	9,984	1,495	15,005
50	3,710	9,988	13,989	11,259

Table 3.4: The number of iterations needed to outweigh the overhead of object creation for the respective inheritance depth.

The *optimization breakpoint* does seemingly not depend on the number of inheritance levels. For example, a server–mode HotSpot VM 6 outweighs the overhead for creation of an object with the depth of 50 even 833 iterations earlier than it does for an object with the depth of five. Nonetheless, there is a significant increase for client–mode HotSpots between the *optimization breakpoint* of an object with the depth of thirty and the depth of fifty. For version 6, the number of iterations increases from 2769 to 3710 (i.e., by 40.0 percent), and for release 5, it jumps up from 1495 to 13,989 (i.e., by 835.7 percent).

The results for the optimization of method invocation are similar to those for object creation (see Table 3.5). The client–mode HotSpot VMs optimize earlier than their server–mode equivalents. For instance, a HotSpot 6 in client–mode needs about 2667 iterations to outweigh the overhead of method invocation on objects with a depth of 10 to 50. In contrast, the server–mode version outweighs the overhead only after 15,069 iterations. The *optimization breakpoint* for method invocation on objects with a depth of five is strikingly low. The reason for that low number is that it does not represent the number of iterations needed to optimize the method invocation on these object. Rather, random fluctuations in performance cause these numbers of iterations. That is, there is hardly any difference between method invocation speed on an object with the depth of five and on the (refactored) wrapper object. Like for object creation time, it is ambiguous whether the number of inheritance level influences the optimization of method invocation. For HotSpot 6, the number of iterations to optimize the method invocation speed increases considerably by 228.9 percent in client mode and by 42.9 percent in server mode between an object with the depth of 30 and one with the depth of 50. However, on HotSpot 5, the compiler needs only four iterations (about 0.3 percent) more to outweigh the performance penalty of an inheritance depth of 50 in client mode. In server mode, it needs even 193 iterations less to outweigh the overhead.

Inheritance Depth	HotSpot 6.0 (client)	HotSpot 6.0 (server)	HotSpot 5.0 (client)	HotSpot 5.0 (server)
5	50	2,972	59	650
10	1,576	9,989	1,495	11,310
30	1,498	14,501	1,495	10,727
50	4,927	20,718	1,499	10,534

Table 3.5: The number of iterations needed to outweigh the overhead of method invocation for the respective inheritance depth.

The performance of object creation and method invocation does not improve with an increasing iteration number on a JRockit VM (at least until the 30,000th iteration). The low fluctuations in standard deviation and mean between the 1000 iterations benchmarks and the 30,000 iterations benchmarks prove this circumstance. In Table 3.6 and in Table 3.7 on the next page, we show comparisons of these statistical values for the object creation and the method invocation benchmarks, respectively.

	Iterations	JRockit 6.0	HotSpot 6.0 (server)	HotSpot 5.0 (server)
Mean	1,000	894.2	1,308.9	1,392.5
	30,000	904.3	989.5	1,084.7
Standard Deviation	1,000	45.2	50.4	42.6
	30,000	59.1	262.9	248.1

Table 3.6: A comparison between the mean and the standard deviation (in ns) of the 1000 and 30,000 iteration benchmark for creation of an object with the depth of ten.

	Iterations	JRockit 6.0	HotSpot 6.0 (server)	HotSpot 5.0 (server)
Mean	1,000	845.1	1,303.3	1,396.1
	30,000	838.5	961.0	1,129.7
Standard	1,000	39.5	56.5	50.0
Deviation	30,000	52.8	243.0	237.8

Table 3.7: A comparison between the mean and the standard deviation (in ns) of the 1000 and 30,000 iteration benchmark for method invocation on an object with the depth of ten.

3.1.5 Conclusion

Replace Inheritance With Delegation has a number of advantages [Blo08]. Especially, the introduction of independence from implementation details of another class can significantly improve the software quality. However, the refactoring is no effective performance optimization for current desktop and server systems. Even though the benchmarks show a performance benefit beginning from five inheritance levels, this performance benefit is at a very low level. For example, a method invocation on an object with an inheritance depth of five takes about 30.6 ns longer than the method invocation on a wrapper object for client–mode HotSpot VM 5. Consequently, the refactoring causes after about 32.7 million method invocations a performance benefit of one second. But even this slight benefit is theoretical; the JIT compilers of the HotSpot VMs can outweigh the overhead of method invocation on a subclass after a few thousand iterations.

Accordingly, it is unlikely that *Replace Inheritance With Delegation* can cause a performance benefit on devices with limited hardware resources; manually programmed classes with an inheritance depth greater than five are seldom [Kab06]. Rather, its counterpart *Replace Delegation with Inheritance* seems to be a refactoring that could increase the performance on such devices. Benchmarks on these devices are needed to validate this assumption.

An exception could be automatic generated programs. These programs can have classes with deep inheritance depths [Bat06]. Because each inheritance level increases nearly linearly the creation time of a class instance and the method invocation time on an object, an automatic refactoring tool as suggested in [SKAP10] may speed up considerably the execution time of these programs.

Thereby, an impediment is the difficulty to apply the refactoring [KS08]. For instance, **protected** members of the former superclass become invisible if the refactored class is not in the same package as the former superclass. Also the visibility of the constructors of the former superclass can prevent the refactoring if the refactored class is a different package than the superclass.

3.2 Inline Method

Several publications show that method inlining has a positive effect on performance in Java [GP09][SKAP10][TS05]. Another analysis of this circumstance is not useful. Consequently, we do not only examine the influence of the refactoring, but also analyze

whether *Inline Method* is beneficial in the presence of a JIT compiler.

3.2.1 Methodology

To reveal the cost of a method invocation within a class, we chain methods that do nothing else than forwarding a call. Only the last called method writes a **double** value into an instance field. All methods have no return value (i.e., their return type is **void**). The depths of the method chain are 5, 10, 20, and 30. We measure the execution time of all created method chains. Subsequently, we inline all methods in the delegation chain other than the one that is called initially. For instance, if we have five methods (`method1`, `method2`, and so on) in a class, only `method1` remains in the class. The other methods are all inlined into `method1`. Because we analyze also whether the refactoring is beneficial in presence of a JIT compiler, the significance test in this section is based on the measurements of a 30,000 iterations benchmark.

3.2.2 Hypotheses

The purpose of this section is to determine whether method inlining is beneficial to improve the performance of a Java program. To focus our study, we set up the following hypotheses. For each hypothesis, H_0 represents the null hypothesis and H_1 represents the alternative hypothesis of the null hypothesis:

- Hypothesis IM1

  H_0: For HotSpot 6.0 in client mode, *Inline Method* does not improve the performance.

  H_1: For HotSpot 6.0 in client mode, *Inline Method* does improve the performance.

The hypotheses are accordant for HotSpot 6 VMSM (IM2), HotSpot 6 VMIM (IM3), HotSpot 5 VMCM (IM4), HotSpot 5 VMSM (IM5), HotSpot 5 VMIM (IM6), and JRockit 6 VM (IM7).

3.2.3 Results

In Figure 3.6 on the following page, we show the performance benefit if a varying number of methods is inlined into a caller method. The figure exhibits that inlining has the expected positive influence for 1000 iterations benchmarks. On a HotSpot 6 VM, the mean overhead of a method invocation is 25.3 ns (3.3 percent). The measurements show that the benefit of method inlining is nearly linear to the number of inlined methods. Table 3.8 on the next page clarifies that the effect of *Inline Method* on the other tested HotSpot VMs is similar. The performance gain per inlined method lies between 19.8 ns (2.4 percent) on a HotSpot 6 VMSM and 31.6 ns (4.2 percent) on a HotSpot 5 VMSM. Therefore, we can estimate the decrease in execution time due to an inlining of a parameterless method (notated as d) for the HotSpot VMs by:

$$d = i * p$$

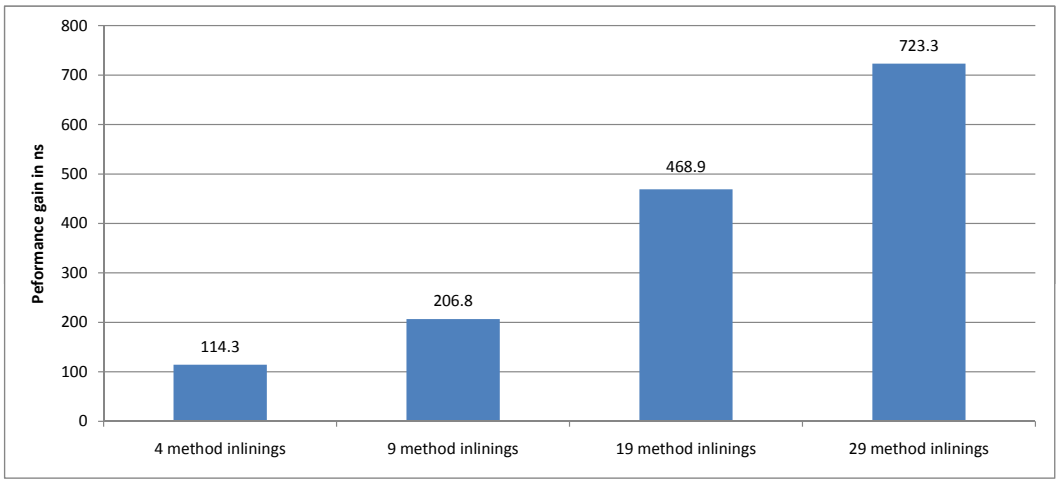

Figure 3.6: The mean performance benefit due to inlining a varying number of methods in the 1000 iterations benchmark on a HotSpot 6 VMCM.

Whereas, i is the number of method invocations and p is the performance gain on the respective virtual machine. In the case of HotSpot 6 VMCM, the formula is thus:

$$d = i * 25.3\,\text{ns}$$

Applying the formula for the HotSpot 6 VMCM, we want to estimate the performance improvement for the inlining of the `output` method into the `compute` in Listing 3.1 on the facing page for three million invocations of the `compute` method. The computed speedup is 75.9 ms. This improvement is slight, but with respect to the low effort reasonable.

Method Inlinings	HotSpot 6.0 (client)	HotSpot 6.0 (server)	HotSpot 6.0 (Xint)
4	114.3	34.5	106.6
9	206.8	212.5	183.7
19	468.9	460.8	467.6
29	723.3	655.4	676.6
Method Inlinings	**HotSpot 5.0 (client)**	**HotSpot 5.0 (server)**	**HotSpot 5.0 (Xint)**
4	107.1	144.6	120.6
9	283.8	261.6	34.1
19	460.5	534.2	447.9
29	778.3	882.2	695.6

Table 3.8: The mean performance benefit (in ns) due to inlining a varying number of methods in the 1000 iterations benchmark.

However, the 10,000 and the 30,000 iteration benchmarks reveal that the computed improvement for the inlining of the `output` method is too optimistic. In Table 3.10 on page 34, we show the difference in performance between the refactored (i.e., the inlined) benchmark program and the benchmark programs with additional method calls for a different number of iterations. On the client–mode and server–mode HotSpot VMs, the

```
class Math {
    int result;

    void compute(int value1, int value2){
        result = value1 + value2;
        output();
    }

    void output() {
        System.out.println(result);
    }
}
```

Listing 3.1: The `output` method can be inlined into the `compute` method.

performance benefit of the method inlinings is in every case smaller in the 30,000 iteration benchmark than in the 1,000 iteration benchmark. The reason is that the JIT compilers of these VMs can inline frequently used methods. As a consequence, the formula for the decrease in execution time due to a method inlining is only true until the *optimization breakpoint*. For example, given that the JIT compiler inlines the `output` method after 1500 iterations, the total performance gain of the manual method inlining would only be $1500 * 25.3\,\text{ns}$ or rather $37,950\,\text{ns}$.

Delegation chain length	HotSpot 6.0 (client)	HotSpot 6.0 (server)	HotSpot 5.0 (client)	HotSpot 5.0 (server)
5	1,568	10,525	1,499	10,031
10	1,580	10,994	1,506	11,554
20	2,377	10,180	1,499	13,043
30	2,989	10,030	1,501	15,232

Table 3.9: The *optimization breakpoints* for the HotSpot VMs with a JIT compiler.

Table 3.9 shows the *optimization breakpoints* for the HotSpot VMs with a JIT compiler. For the server–mode HotSpot 6 VM and the client–mode HotSpot 5 VM, there is seemingly no correlation between the number of called methods and the *optimization breakpoint*. For instance, the JIT compiler of the HotSpot 6 VMSM optimizes the execution time of the delegation chain with a length of 30 earlier than the execution time of the delegation chain with a length of five.

In contrast, on HotSpot 6 VMCM and HotSpot 5 VMSM, the JIT compiler needs the more iterations to optimize the more methods are called. Nonetheless, the manual inlining of programs with a high method number causes most likely no additional benefit on both VMs. There are two main reasons. First, the optimization by the JIT compiler is non–deterministic [GBE07]. The correlation can be random. Second, the benchmark program calls equally often all methods. However, in practice, a small number of methods causes the most execution time of a program [Mey05][ALSU06]. The JIT compiler detects these *hot spot methods* and optimizes them (e.g., it inlines methods that are called by the *hot spot methods*). Thus, the manual method inlining causes only slight performance gains on the HotSpot VMs with a JIT compiler.

Method Inlinings	Benchmark iterations	HotSpot 6.0 (client)	HotSpot 6.0 (server)	HotSpot 6.0 (Xint)
4	1,000	114.3	34.5	106.6
	10,000	14.7	56.5	119.3
	30,000	2.1	-15.3	106.3
9	1,000	206.8	212.5	183.7
	10,000	30.7	236.0	214.0
	30,000	5.1	52.4	210.8
19	1,000	468.9	460.8	467.6
	10,000	88.8	498.6	472.0
	30,000	23.4	158.8	453.7
29	1,000	723.3	655.4	676.6
	10,000	175.3	744.4	693.8
	30,000	62.8	213.2	690.3
Method Inlinings	Benchmark iterations	HotSpot 5.0 (client)	HotSpot 5.0 (server)	HotSpot 5.0 (Xint)
4	1,000	107.1	144.6	120.6
	10,000	5.6	160.0	124.1
	30,000	-2.0	29.3	133.4
9	1,000	265.6	283.8	261.6
	10,000	43.0	287.5	257.5
	30,000	19.7	103.4	270.3
19	1,000	460.5	534.2	447.9
	10,000	64.3	571.0	454.5
	30,000	23.2	219.7	454.2
29	1,000	778.3	882.2	695.6
	10,000	153.9	911.2	740.2
	30,000	83.6	389.9	763.9

Table 3.10: The performance gain (in ns) due to *Inline Method* for a varying number of inlinings.

The situation for the interpreted mode (i.e., non–JIT) HotSpot VMs is different. These VMs do not optimize the programs at runtime. Consequently, the method inlings must be applied before the classes are loaded by the JVM. Since the (Sun Microsystems) bytecode compiler performs only few optimizations [Sun05a], the *Inline Method* refactoring can be more beneficial on non–JIT VMs such as KVM than on VMs with a JIT compiler. Moreover, devices that run without a JIT compiler have usually less hardware resources than the used benchmark system.[1]

Accordingly, the JRockit VM shows differences to the HotSpot VMs with a JIT compiler. Figure 3.7 on the next page exhibits that the mean performance benefit due to the method inlinings is not generally smaller after 30,000 iterations than after 1,000 iterations on a JRockit VM. That it, the JRockit VM does not optimize the method

[1]There are only a few reasons to run a desktop or a server system without a JIT compiler. For example, the use of AOT compilation [Mik04].

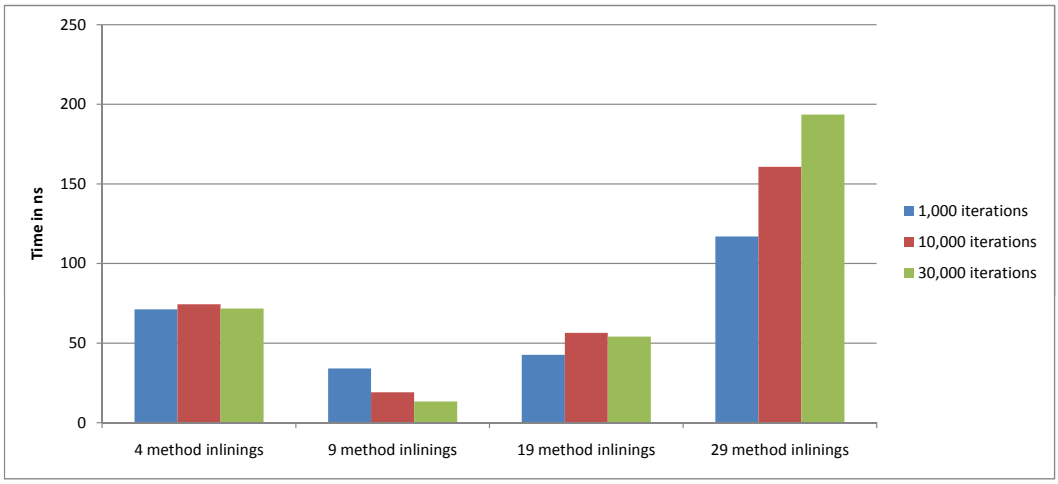

Figure 3.7: The mean performance benefit due to *Inline Method* on a JRockit VM for a varying number of method inlinings and benchmark iterations.

invocation until the 30,000th iteration.[2] However, in the 1000 iterations benchmarks, a method invocation on a JRockit VM is significantly faster than on the HotSpot VMs. In average, it takes only 7.0 ns—the method invocation is 3.63 times faster than on a HotSpot 6 VMCM. The fast method invocation means that method inlining causes on the JRockit VM—at least for short–running applications—a smaller benefit than on the HotSpot VMs. Figure 3.8 on the following page reveals that the situation may be different for long–running applications. In the shown example, the cumulated performance benefit due to method inlining is significantly smaller for the JRockit VM than for the HotSpot VMs after 500 iterations. However, after 20,000 iterations, the cumulated performance benefit on a JRockit VM is bigger than on a HotSpot 6 VMCM. The reason is that the HotSpot VMs with a JIT compiler optimize the method invocation speed at runtime.

Another difference of the JRockit VM to the HotSpot VMs is that the measurements show no clear linear correlation between the number of inlined methods and the performance benefit (see Table 3.7). For example, the inlining of 9 or 19 methods causes a slighter performance gain than the inlining of four methods on this VM. This aberration is possibly caused by random fluctuations in performance. As we hinted in Section 2.3 on page 10, micro—benchmarks are fault–prone.

Table 3.11 on page 37 shows the computed confidence intervals for the 30,000 iterations benchmarks. All number of inlinings have a positive effect on the tested VMs (i.e., $\bar{x}$ is positive). Moreover, no confidence interval includes zero. Consequently, we have to reject the null hypotheses of Theses IM1 to IM7. That is, *Inline Method* improves the execution time of a Java program.

Nonetheless, the results deserves a critical questioning. First, even though the performance benefit is statistically significant, it is arguable whether it is of practical relevance. For example, the decrease due to the inlining of 19 methods is only 23.4 ns on a HotSpot 6 VMCM. Second, we show on page 33 that the *optimization breakpoint* is reached for all HotSpot VMs with a JIT compiler before the 30,000th iteration. The mean values of

[2] In fact, it does not optimize the method invocation between the 1st and the 30,000th iteration. It is possible that the JRockit VM optimizes the method calls before the first iteration.

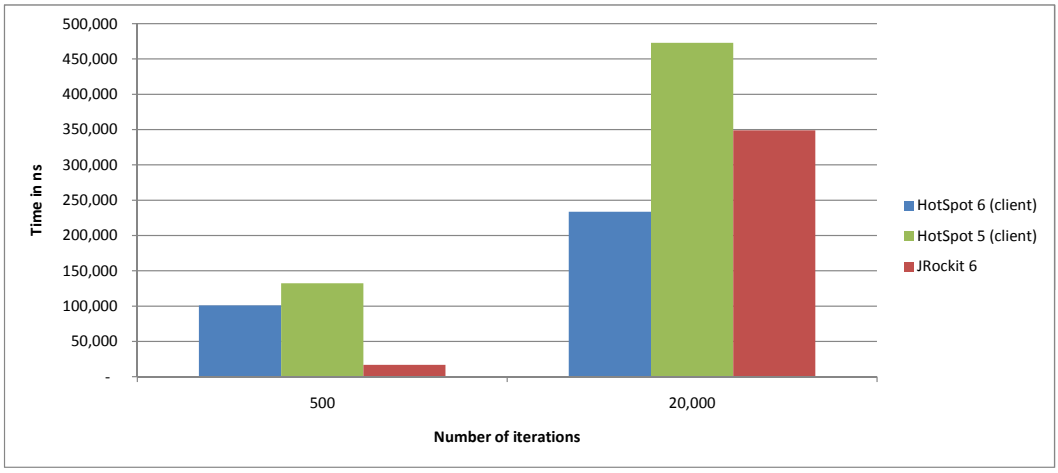

Figure 3.8: The cumulated performance benefit due to the inlining of nine methods after 5000 and 20,000 iterations.

the non–refactored and the refactored measurements approximate, thus, more and more for an increasing iteration number on these VMs. Eventually, there is no more difference between the refactored and the non–refactored version. This means that if we used a significance test based on a 300,000th iteration benchmark, there would be most likely no statistically significant differences in the runtimes of the refactored and non–refactored program for the HotSpot VMs with a JIT compiler.

3.2.4 Conclusion

The benchmarks confirm the results of previous studies—the *Inline Method* refactoring improves the performance. However, the benefit of the manual method inlining to improve the performance is arguable. First, the JIT compiler of the HotSpot VMs can automatically inline frequently used methods. Second, the performance overhead of (static) method invocation in Java is slight. For instance, the avoidance of 29 parameterless method invocations saves only about 700 ns on a HotSpot 6 VMIM. Consequently, even on a VM without a JIT compiler, one million method invocations cause only an overhead of 24 ms. Third, method inlining may increase the code duplication (e.g., if we inline the body of a method into two other methods). As a result, the code quality deteriorates in terms of maintainability and readability [McC04]. Moreover, code duplication causes an increased memory footprint [AFSS00]. Fourth, [DC94] shows that overinlining can even harm the performance, because it causes cache misses.

Therefore, we discourage from manually inlining Java methods to improve the performance. In reverse, the low overhead of method invocation indicates that we can apply *Extract Method* without facing serious performance penalties. For example, [SFP+08] suggests the extracting of each additional level of indentation in a method into a new method to improve the understandability of a program.

A special case is the development of performance–critical applications. For such applications, the cumulated effect of method invocation can cause a considerable performance problem. Accordingly, on devices with scarce hardware resources, a method invocation takes presumably more time than on the used benchmark system. In addition,

VM	# Method Inlinings	$\overline{x}$	s_x	c_1	c_2
HotSpot 6 (client)	4	2.1	0.5	3.0	1.1
	9	5.1	0.6	6.3	3.9
	19	23.4	0.8	25.2	21.6
	29	62.8	1.2	65.4	60.2
HotSpot 6 (server)	4	15.3	0.8	17.0	13.5
	9	52.4	1.2	55.1	49.8
	19	158.8	1.8	162.6	154.9
	29	213.2	2.5	218.7	207.8
HotSpot 6 (Xint)	4	106.3	0.4	107.1	105.5
	9	210.8	0.4	211.7	210.0
	19	453.7	0.3	454.5	453.0
	29	690.3	0.4	691.1	689.4
HotSpot 5 (client)	4	2.0	0.4	2.9	1.1
	9	19.7	0.6	21.0	18.4
	19	23.2	0.8	24.9	21.6
	29	83.6	1.1	85.9	81.2
HotSpot 5 (server)	4	29.3	0.8	31.1	27.5
	9	103.4	1.2	105.9	100.9
	19	219.7	1.9	223.8	215.6
	29	389.9	2.8	396.0	383.9
HotSpot 5 (Xint)	4	133.4	0.3	134.0	132.8
	9	270.3	0.3	271.0	269.5
	19	454.2	0.3	454.8	453.5
	29	763.9	0.4	764.7	763.0
JRockit 6	4	71.7	0.4	72.5	70.9
	9	13.4	0.4	14.1	12.6
	19	54.1	0.4	54.9	53.3
	29	193.6	0.6	194.8	192.3

Table 3.11: The confidence intervals for a varying number of method inlinings in the 30,000 iterations benchmark.

such devices have usually no JIT compiler. As a consequence, *Inline Method* may cause a significant performance improvement on these devices; the benchmarks show that the benefit due to *Inline Method* is nearly linear to the number of inlined methods. However, we must be aware that devices with limited hardware resources are often subject to tight memory constraints. Thus, to decide whether to inline a method requires also a balancing of runtime performance and memory footprint for such devices.

3.3 Inline Class

Although the inlining of classes can violate encapsulation, [Shi03] indicates that it is in some cases reasonable to speed up the object creation time. Especially if an application must create a lot of class instances, object allocation can seriously harm its performance

[Ruo98][Goo10]. Consequently, we test in this section whether the refactoring *Inline Class* is an efficient way to tune the object allocation performance or not.

3.3.1 Methodology

First, we create several wrapper classes (called `Wrapper`). All of these classes wrap a `GetterSetter` class with one, three, or five fields. Each field of such a `GetterSetter` class has the same type and is set in the constructor of the class. There are getter–setter classes for the data types **int**, **double**, `java.lang.Integer`, `java.lang.Double`, and `java.lang.String`. Beside the fields and the constructor, every `WrappedGetterSetter` class contains a setter and getter method for each of its fields. When we invoke a constructor of a `Wrapper` class, it invokes the constructor of the respective `GetterSetter` class (i.e., it creates an instance of the `GetterSetter` class) and saves a reference to the instance into a **private** field. In addition, every `Wrapper` has *forwarding methods* that forward their invocation to the respective method of the wrapped class.

Second, we measure the creation time of a flattened getter–setter object. That is, we inline every `WrappedGetterSetter` class into the containing class. The resulting class is called flattened, because [Shi03] refers merging multiple classes as *flattening*.

Finally, we analyze the creation time of more complex examples. Therefore, we create three `MultiWrapper` classes. Every `MultiWrapper` class contains five flattened `GetterSetter` classes created in step one (i.e., one getter–setter class per data type). Three `MultiWrapper` classes are necessary, because the wrapped getter–setter classes have constructors with a varying number of parameters (one, three, and five). Afterwards, we inline the `GetterSetter` classes into a single class.

3.3.2 Hypotheses

The focus of this section is to determine whether class inlining improves the object creation time. To focus our study, we set up the following hypotheses. For each hypotheses, H_0 represents the null hypothesis and H_1 represents the alternative hypothesis of the null hypothesis:

- Hypothesis IC1

  H_0: For HotSpot 6.0 in client mode, class inlining does not improve the object allocation speed.

  H_1: For HotSpot 6.0 in client mode, class inlining does improve the object allocation speed.

The hypotheses are accordant for HotSpot 6 VMSM (IC2), HotSpot 6 VMIM (IC3), HotSpot 5 VMCM (IC4), HotSpot 5 VMSM (IC5), HotSpot 5 VMIM (IC6), and JRockit 6 VM (IC7).

3.3.3 Results

The results indicate that *Inline Class* can improve the object creation time. Table 3.12 on the next page shows that the creation of a flattened getter–setter object is faster than

its wrapped counterpart on a HotSpot 6 VM in client mode. For this VM, the inlining of the `GetterSetter` classes improves the time to create an instance of the `Wrapper` class by 6.3 to 14.4 percent and by 11.3 percent on average.

Number of Parameters:	1		3		5	
Type	Wrapped	Flattened	Wrapped	Flattened	Wrapped	Flattened
int	855.9	759.0	895.3	805.5	935.8	844.1
double	876.1	788.4	986.8	860.9	996.8	934.5
Integer	877.0	793.8	944.8	834.8	996.6	891.5
Double	894.0	774.4	960.9	822.6	1,025.9	894.3
String	897.3	795.5	946.3	820.2	1,011.5	883.3

Table 3.12: The creation time (in ns) of deep getter–setter objects and refactored (i.e., flattened) getter–setter objects.

The positive effect of flattening on object creation time is also true for all other tested virtual machines. In Table 3.13, we present the creation time benefit of creating a flattened getter–setter object with a five parameters constructor. *Inline Class* reduces the creation time of such objects on average by 99.9 ns. The creation of a `GetterSetter` object with **int** fields on a JRockit VM is the only exception. On this virtual machine, the creation of a flattened getter–setter **int** object is 44.0 ns slower than the instantiation of a wrapper with the same functionality. Generally, the performance difference between the creation of a flattened and a `Wrapper` object is less noticeable on a JRockit VM than on a HotSpot VM. The performance gain due to inlining the `GetterSetter` objects is on average 109.9 ns for a HotSpot VMs, but it is just 40.1 ns for the JRockit VM.

Type	HotSpot 6.0 (client)	HotSpot 6.0 (server)	HotSpot 6.0 (Xint)	HotSpot 5.0 (client)
int	91.7	118.9	145.8	117.5
double	62.3	103.9	96.6	25.5
Integer	105.1	250.4	111.4	102.4
Double	131.6	41.8	189.9	121.8
String	128.2	109.6	160.6	96.1

Type	HotSpot 5.0 (server)	HotSpot 5.0 (Xint)	JRockit 6.0
int	87.7	56.2	-44.0
double	72.9	58.0	43.8
Integer	203.3	69.4	41.6
Double	155.2	80.9	118.5
String	99.0	103.2	40.5

Table 3.13: Performance gain (in ns) due to flattening a Wrapper class with five parameters. Negative numbers mean that there is a performance degradation.

Figure 3.9 on the next page shows that the decrease in runtime due to *Inline Class* is nearly linear to the number of created class instances. The figure depicts the decrease in creation time on a HotSpot 6 VMIM due the inlining of a `GetterSetter` class with

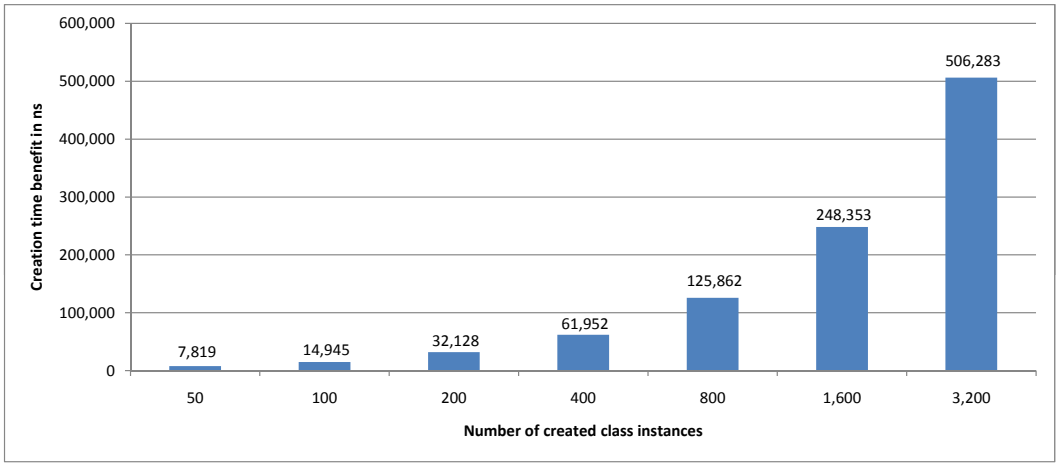

Figure 3.9: The cumulated decrease in runtime due to class inlining on an interpreted–mode HotSpot 6 VM.

five **int** fields into a `Wrapper` class. In the example, the mean performance improvement per created class instance is 145.8 ns. Assuming a linear gradient, we can compute the total decrease in creation time (notated as d) due to the inlining of the `GetterSetter` class by:

$$d = n_I * 145.8\,\text{ns}$$

Whereas, n_I is the number of created instances of the `Wrapper` class. Consequently, the creation of an half million instances of a flattened getter–setter class would save about 72.9 ms. However, Figure 3.10 on the facing page exhibits that the assumption of a linear performance improvement is not true for all JVMs. The figure shows d for a HotSpot 6 VM in client mode. On this VM, the performance improvement due to *Inline Class* nearly stagnates between 1,600 and 3,200 object instantiations. The reason is that a JIT compiler can optimize the allocation of objects. Therefore, we cannot assume that the decrease in runtime due to *Inline Class* is linear to the created instances of a flattened class on a JVM with a JIT compiler.

Beside the performance benefit with respect to object creation, *Inline Class* can also improve the method invocation time. The refactoring can make dispensable *forwarding methods*. Let us consider the Person example again (see on page 9). To access the methods of its wrapped classes, the `Person` class must provide *forwarding methods*. These methods forward the calls to the methods of the wrapped classes. For instance, we could determine the street name of the person's address with a call to `person.getStreetName()`. This method call implies two additional invocations, because it forwards the call to `address.getStreetName()` that forwards it to `street.getStreetName()`. The inlining of the `Street` and the `Address` class into the `Person` class would make dispensable two *forwarding methods*. Consequently, *Inline Class* can bring about the performance advantages of *Inline Method* (see Section 3.2 on page 30).

The benchmarks of the `MultiWrapper` classes (i.e., classes with five wrapped `GetterSetter` classes) also show that flattening has a positive effect on object creation speed. Table 3.14 on the next page depicts that flattening speeds up the creation of a `MixedWrapper` class instance by averagely 507.8 ns on a HotSpot VM and by 206.3 ns on

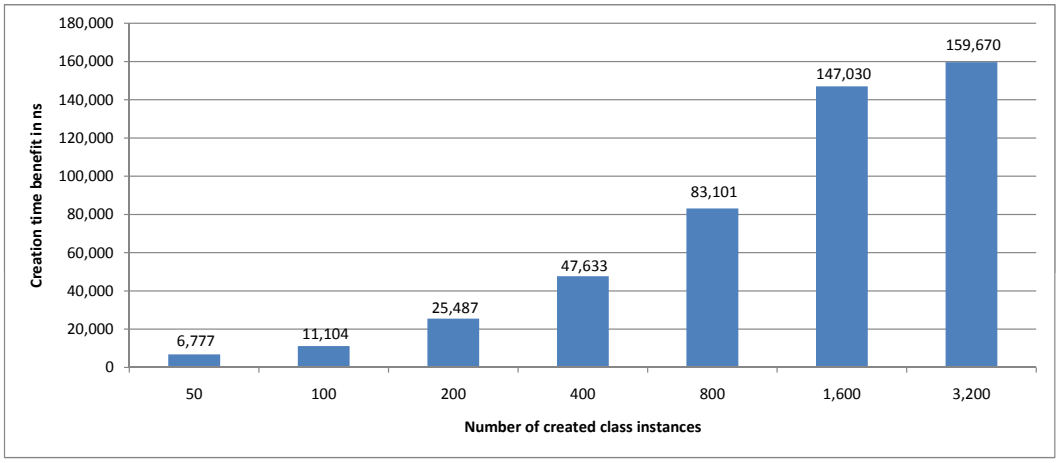

Figure 3.10: The cumulated decrease in runtime due to class inlining on an client–mode HotSpot 6 VM.

a JRockit VM. The inlining improves the object creation performance for all numbers of parameters and all virtual machines. Moreover, the performance benefit in terms of object creation time increases with a rising number of parameters on the HotSpot VMs. Therefore, we can assume that the more parameters supplied to the constructor of the class to inline the higher the performance benefit due to *Inline Class*.

The significance test prove that the observed performance improvements are statistically significant—none of the calculated confidence intervals in Table 3.3.3 on the following page include zero. The difference between the instantiation time of the non–refactored and the refactored classes (i.e., $\bar{x}$) is positive in all cases. That is, class inlining does improve the object allocation speed for all virtual machines. We have to reject the null hypotheses of Hypothesis IC1 to IC7.

Parameters	HotSpot 6.0 (client)	HotSpot 6.0 (server)	HotSpot 6.0 (Xint)	HotSpot 5.0 (client)
1	403.8	498.5	533.7	304.4
3	509.3	545.7	598.2	417.9
5	660.6	578.2	625.7	611.3

Parameters	HotSpot 5.0 (server)	HotSpot 5.0 (Xint)	JRockit 6.0
1	328.4	296.7	232.4
3	507.5	494.2	191.8
5	591.4	630.9	194.8

Table 3.14: Performance improvement (in ns) due to flattening a MultiWrapper class.

Influence of the JIT Compiler

Table 3.16 on the next page depicts the JIT breakpoints for the HotSpot virtual machines in client and sever mode. On a HotSpot VM in client mode, the JIT compiler can outweigh the cost of the additional object allocations after averagely 1496 iterations for

VM	Parameter	$\bar{x}$	s_x	c_1	c_2
HotSpot 6	1	403,8	4,8	407,8	399,8
(client)	3	509,3	9,1	516,9	501,7
	5	660,6	9	668,2	653,1
HotSpot 6	1	498,5	5,3	502,9	494,1
(server)	3	545,7	12,1	555,8	535,6
	5	578,2	12,5	588,6	567,8
HotSpot 6	1	533,7	6,6	539,2	528,2
(Xint)	3	598,2	6,6	603,8	592,7
	5	625,7	9,3	633,5	617,9
HotSpot 5	1	304,4	5,1	308,7	300,2
(client)	3	417,9	10	426,2	409,6
	5	611,3	10,7	620,3	602,4
HotSpot 5	1	328,4	5,1	332,7	324,1
(server)	3	507,5	6,8	513,1	501,8
	5	591,4	9,2	599,2	583,7
HotSpot 5	1	296,7	3,9	299,9	293,4
(Xint)	3	494,2	10,2	502,7	485,7
	5	630,9	9,4	638,7	623,1
JRockit	1	232,4	5	236,6	228,3
	3	191,8	6,5	197,2	186,4
	5	194,8	7,6	201,1	188,5

Table 3.15: The confidence intervals for the sets of measurements.

version 5 and about 1790 iterations for release 6. The JIT compiler of the server–mode HotSpot VM takes longer. Here, the JIT compiler needs about 11,395 iterations for HotSpot 5 VM and approximately 9951 iterations on a HotSpot 6 VM to outweigh the performance penalty of the additional object allocations.

Parameter/ Constructor	HotSpot 6.0 (client)	HotSpot 6.0 (server)	HotSpot 5.0 (client)	HotSpot 5.0 (server)
1	1,822	9,958	1,507	11,708
3	1,759	9,972	1,500	11,184
5	1,813	9,948	1,505	11,317
Mean:	1,790	9,951	1,496	11,395

Table 3.16: The *optimization breakpoints* for the HotSpot VMs with a JIT compiler.

The JRockit VM performs no optimizations.[3] Table 3.17 on the facing page shows that the mean runtimes of the 1000, and the 30,000 iterations benchmarks are nearly the same. However, the object creation on a JRockit VM is significantly faster than the unoptimized object creation on a HotSpot VM. For example, the creation of a `MultiWrapperWrapper` object with five wrapped objects is about 48.3 percent faster than on a HotSpot 6 VM in the client mode for the 1000 iterations benchmark.

[3]In fact, it does not optimize the method invocation between the 1st and the 30,000th iteration. It is possible that the JRockit VM optimizes the method calls before the first iteration.

Parameters	Iterations	Deep	Flatten
1	1,000	915.3	682.8
	30,000	928.1	699.5
3	1,000	954.7	762.9
	30,000	963.9	766.2
5	1,000	1,038.5	843.8
	30,000	1,072.3	871.8

Table 3.17: The mean time (in ns) for creation of a MultiWrapper object with five parameters on a JRockit VM.

Conclusion

Although the relative performance improvement of *Inline Class* is significant, it reduces the creation time of an instance by at most a few hundred nanoseconds. Consequently, only if the VM allocates vast numbers of new objects wrapping other objects, the refactoring can improve the performance. For instance, on the used system with a non–JIT VM, we had to create over one million instances of an inlined class to decrease the runtime of a program by one second in comparison with the initiation of a behavior–equivalent wrapper class. The creation of such amount of objects is rarely necessary and can mostly be avoided by *object pooling* or similar techniques. Furthermore, the JIT compiler of the HotSpot VM 5 and 6 can outweigh the cost of wrapped objects after a few thousand iterations. Therefore, the claim that JIT compilers cannot improve the object allocation time [Ruo98] is not true anymore.

Given these facts, we have to emphasize that *Inline Class* is no effective way to tune the performance of Java programs on current desktop and server devices with a modern JVM. The reason is that JVM can nowadays allocate new objects very fast—the allocation requires fewer than 10 cycles on most processors [Hei07].

However, *Inline Class* can be beneficial for devices with scarce hardware resources. The allocation of a new object could be significantly more expensive on such devices. Consequently, the positive effect of *Inline Class* would be more considerable. Furthermore, such devices have often no JIT compiler. The JVM can thus not outweigh the overhead of often created instances. Another benefit is that *Inline Class* makes dispensable *forwarding methods*. In this vein, it can improve additionally the runtime performance. Therefore, we argue to consider *Inline Class* if the object allocation time is high on embedded, mobile, or comparable devices.

Inline Class brings about some disadvantages as well. First, too much class inlining can harm the performance, because large classes can cause cache misses [TS05]. Second, *Inline Class* causes code duplication when inlining a class into two or more classes. In such cases, the refactoring can increase the memory footprint of a program (see Section 4.3 on page 63). Third, class inlining may deteriorate internal software quality characteristics (such as maintainability) of a program [McC04][Shi03].

3.4 Summary

In this section, we give a short review about the results of Chapter 3.

Replace Inheritance with Delegation :

- Each additional level of inheritance increases the object allocation time nearly linearly. The increase per inheritance level lies between 20.9 ns (about 1.8 percent) and 31.9 ns (3.2 percent) for the HotSpot VMs and it is 5.3 ns (about 0.6 pecent) for the JRockit VM.
- Accordingly, each inheritance level increases the method invocation time approximately linearly. The increase per inheritance level is 18.8 ns to 25.1 ns (1.7 to 2.2 percent) for the HotSpot VMs. For the the JRockit 6 VM, it is only 2.2 ns (about 0.3 percent).
- In fact, the method invocation performance does not depend on the nominal inheritance depth of a class. Rather, it depends on where the used implementation of the method is located. Thus, for determining the method invocation overhead, it is important to figure out how many inheritance levels are between the class on that the method is invoked and the class that provide the used method implementation.
- The JIT compilers of the HotSpot VMs can outweigh the cost of method invocation and object allocation that comes with inheritance for often used classes.
- *Replace Inheritance with Delegation* improves the performance for classes with an inheritance depth greater than four on HotSpot VMs. On the JRockit VM, the refactoring improves the performance for classes with an inheritance depth greater than six. For lower inheritance depths, the refactoring causes a performance degradation.

Inline Method :

- A static, parameterless method invocation is very cheap on the tested VMs. It takes between 19.8 ns and 31.6 ns on a HotSpot VM. On a JRockit VM, the method invocation takes only 7.0 ns.
- The performance gain due *Inline Method* is nearly linear to the number of avoided method invocations on the HotSpot VMs.
- The JIT compilers of the HotSpot VMs can nullify the benefit of manual method inlining after invoking the concerned (i.e., not inlined) method a few thousand times.
- The JRockit VM shows no clear correlation between the number of inlined methods and the caused performance gain. Possibly, this aberration is caused by random fluctuations in performance.

Inline Class :

- The performance benefit of *Inline Class* is approximately linear to the number of created instances.

- On HotSpot VMs, the inlining of a class with five **double** fields reduces on average the object creation time by 66.1 ns (7.0 percent). On a JRockit VM, the same inlining causes a performance benefit of 43.8 ns (about 5.8 percent).
- Class inlining can make obsolete *forwarding methods* and, thus, reduce the method invocation time as well.
- The JIT compilers of the HotSpot VMs can outweigh the object allocation overhead after a varying number of iterations. That is, modern JIT compilers can optimize the object allocation time (unlike stated in [Ruo98]).

For the refactorings *Inline Method* and *Inline Class*, the assumptions from the beginning of this chapter prove true: both refactorings improve the execution time of a Java program. However, the benefit is so slight that it is arguable whether the improvement is of practical relevance. In addition, the JIT compilers of the HotSpot VMs can nullify the benefit of both refactorings for frequently traversed code paths after few thousand iterations. The benefit for devices with scarce hardware resources may be more significant. Consequently, benchmarks on such devices are needed to validate this supposition.

The use of *Replace Inheritance with Delegation* is only true for high inheritance depths (e.g., on HotSpot VMs greater than four). Therefore, the general claim that *Replace Inheritance with Delegation* improves the performance is not tenable.

Chapter 4

Memory Footprint

In this chapter, we analyze the influence of refactoring on memory footprint. For that purpose, we exemplify the effect of refactoring on the heap and non–heap memory of the JVM with the refactorings *Inline Class*, *Inline Method*, and *Replace Inheritance with Delegation*. Our assumptions are that:

1. *Replace Inheritance with Delegation* increases the memory footprint, because *forwarding methods* and a field that references to the wrapped instance of the (former) superclass are needed.
2. *Inline Method* decreases the memory consumption if the method to inline is used by only a single method. Otherwise, the refactoring causes code duplication and, thereby, increases the memory footprint.
3. *Inline Class* decreases the memory footprint if the class to inline is used by only a single class. Contrariwise, the refactoring causes code duplication and, thus, increases the memory consumption.

For each refactoring, we develop a formula by which we can estimate its influence on memory consumption.

4.1 Replace Inheritance with Delegation

The refactoring *Replace Inheritance with Delegation* changes the association of two classes. Instead of inheriting methods and fields, a class acts a wrapper of the former superclass. The refactoring brings about numerous advantages such as the observance of encapsulation. We present further details in Section 2.1.3 on page 7.

However, the need for *forwarding methods* and a field that references to the component (i.e., the wrapped object) indicates that the refactoring increases the memory consumption. In this section, we analyze whether this assumption is true or not.

4.1.1 Methodology

First, we determine the variables that influence the memory footprint of an inherited class. Second, we do the same for a composed class. Finally, we compare the memory

consumption of both inherited classes and composed classes in a practical example. For that purpose, we create a `BasisClass`, which directly inherits form `java.lang.Object`. This class has ten methods. The methods have zero to nine **int** parameters.

In case of inheritance, each subclass overrides a single method. For instance, `SubClass1` inherits from `BasisClass` and overrides the method with one parameter. Then, `SubClass2` inherits from `SubClass1` and overrides the method with two parameters. The last element in the inheritance chain—namely `SubClass9`—inherits thus only the parameterless `method1` from `BasisClass`.

The example programs with the composed classes work similarly. The `ComposedBasisClass` has a field that references to an instance of `BasisClass`. This way, it can forward all calls to its methods to the corresponding ones in `BasisClass`. In accordance with `ComposedBasisClass`, all methods of `ComposedClass1` forward their invocation to `BasisClass`. The only the exception is `method2` that works exactly like the overridden `method2` in `SubClass1`. Accordingly, `method2` and `method3` in `ComposedClass2` are equal to `method2` and `method3` of `SubClass2`. The other methods of `ComposedClass2` forward their invocation to the corresponding ones in `BasisClass`. The replacement of the *forwarding methods* goes on in this vein until the `ComposedClass9` only forwards the call of its `method1` to the corresponding one in `BasisClass`.

Every inherited and composed class has ten **int** fields. That is, all the classes have one field per method. For instance, if we invoke `method3` and `method4`, they allocate the corresponding value to `field3` and, respectively, to `field4`. Listing 4.1 on the next page shows the bodies of `method3` and `method4` in `BasisClass`. Their overridden versions are shown in Listing 4.2 on the facing page.

By following the described procedure, inherited and composed classes are generated that behave identically. This means that, for example, `ComposedClass4` behaves like `SubClass4` and the invocation of the methods of `ComposedClass8` yield the same results like calls to the methods of `SubClass8`. Consequently, it is legitimate to compare the CMF and IMF of the classes that use different reuse techniques.

4.1.2 Results

The analysis of the results is divided into three parts. First, we discuss the memory consumption of an inherited class. Second, we explain how to determine the memory footprint of a composed class. And finally, we conduct the benchmark explained on the previous page.

Inheritance

The JVM loads normally a Java class into the RAM when one of its constructors is called the first time. The loading of this class causes also the loading of its ancestor classes. Hence, the deep memory footprint of the class comprises its own memory footprint and the memory that its ancestor classes consume. However, there are exceptions to this procedure. First, if an ancestor class is already loaded into the non–heap area of the JVM, it is not loaded again (see Section 2.5 on page 16). Second, the `java.lang.Object` class is a common ancestor of all Java classes. It is loaded into the RAM when the JVM starts [LY99]. The class memory footprint does not increase in both cases.

```
1    int method3(int variable1, int variable2) {
2        field3 = variable1 + variable2;
3        return field3;
4    }
5    int method4(int variable1, int variable2, int variable3) {
6        field4 = variable1 + variable2 + variable3;
7        return field4;
8    }
```

Listing 4.1: The original versions of method3 and method4.

```
1    int method3(int variable1, int variable2) {
2        field3 = (int) Math.pow(2,
3                    Math.abs(variable1 - variable2));
4        return field3 + method1() + method2(variable1);
5    }
6    int method4(int variable1, int variable2, int variable3) {
7        field4 = (int) Math.pow(2,
8                    Math.abs(variable1 - variable2 - variable3));
9        return field4 + method1() + method2(variable1)
10                   + method3(variable1,variable2);
11   }
```

Listing 4.2: The overridden versions of method3 and method4.

Barring these exceptions, each inheritance level increases the memory footprint of a class by at least 928 byte. This value is the minimal memory consumption of a *class torso*[1] on an interpreted–mode HotSpot VM 6 (see 4.3 on page 63). However, there is an additional overhead for more than eight inheritance levels (see Figure 4.1 on the following page). The minimal memory footprint of these classes increases additionally by about 4 byte per inheritance level[2]. That is, each inheritance level increases at least linearly the non–heap memory consumption. For instance, an empty class that inherits from 19 empty ancestor classes has a class memory footprint of 19,040 byte.

The increase in memory footprint is even more noticeable if the ancestor classes have methods that the child class does not use. Such superclasses have methods that are either overridden or not accessible (e.g., if they are **private**) in the subclass. In Listing 4.3 on page 51, we show such an example. Even though `Math3` only makes use of the inherited `sub` method, `Math1` and `Math2` are loaded completely to the non–heap area of the JVM. If `Math1` and `Math2` are not initiated, the needlessly loaded methods `add` and `square` waste 728 byte.

Accordingly, all fields of the ancestor classes are loaded into the RAM regardless whether they are usable from the class instance or not. For example, although the **private** field `hundred` in the `MathField1` class of Listing 4.4 on page 52 is not accessible from a `MathField2` instance, it increases the CMF by 72 byte.

[1] A *class torso* is the declaration of a class without its body. That is, it consists of the modifiers, the class name, the declaration of the implemented interfaces, and the superclass declaration.

[2] The memory footprint is always a multiple of eight. Thus, each two inheritance levels have the same memory footprint.

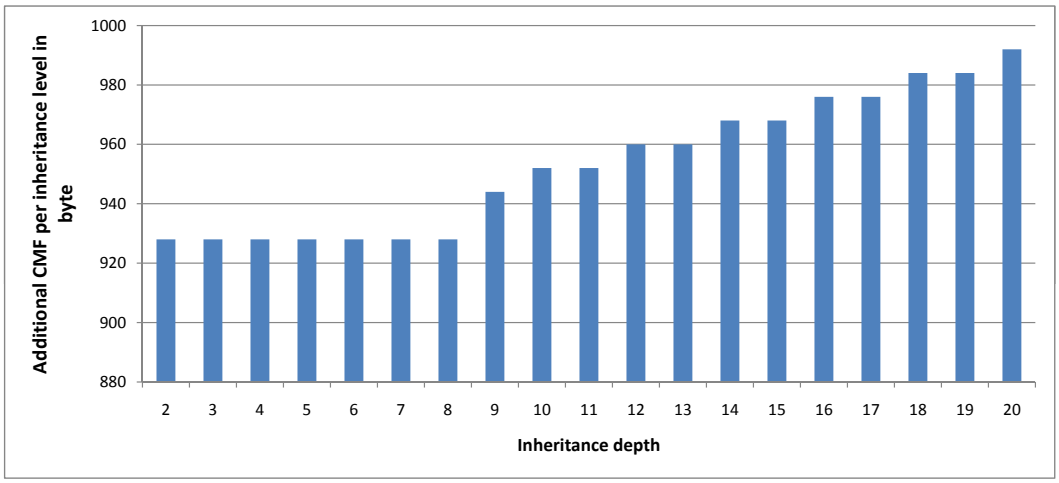

Figure 4.1: The increase of the CMF per level of inheritance for an empty class.

Moreover, annotations consume additional memory. For instance, the `@Override` in `Math3` of Listing 4.3 on the next page consumes 96 byte. The memory consumption is independent of the number of annotated methods.[3] Interestingly, the retention level of the `@Override` annotation is `SOURCE`. The compiler should thus discard the annotation [Sun04]. Nevertheless, the annotation causes an increase in memory footprint. As a result, the use of `@Override` blows up the memory footprint without having any benefits at runtime. Annotations with the retention type `RUNTIME` (i.e., that have a meaning at runtime) consume memory per declaration. For example, a `@Deprecated` annotation consumes at least 24 byte per declared method on a HotSpot 6 VMIM.

To summarize, the deep CMF of a subtype class depends on the memory consumption of its ancestor classes. If the ancestor classes have numerous methods and fields, the increase in CMF per inheritance level can be considerably larger than linear.

Because it can be difficult to predict when a class is loaded and, especially, unloaded [GJSB05], we ignore the fact that an ancestor class is possibly already loaded into the non–heap area of the JVM. This means that we compute the deep CMF of a class (abbreviated as k):

$$k = \sum_{i=2}^{d} u_i$$

Whereas d is the inheritance depth of the class, and u_i is the class memory footprint of the corresponding class. We do not count the first class in the inheritance chain, because it is `java.lang.Object`. For the HotSpot 6 VMIM, we can estimate the minimal CMF k_{min} by:

$$k_{min} = \sum_{i=2}^{d} (928\,\text{byte} + n_{m_i} * 176\,\text{byte} + n_{p_i} * 30\,\text{byte} + n_{f_i} * 24\,\text{byte})$$

Whereas n_{m_i} is the method number, n_{p_i} is the parameter number of all methods, and n_{f_i} is the field number of the corresponding class in the inheritance chain.

[3] Using the `@Override` annotation, we can declare that a method is supposed to override a method of a superclass. If the method fails to override it (e.g., if it has a mismatching signature), the compiler throws an error.

```
class Math1 {
    public int add(int value1, int value2){
        return value1 + value2;
    }
    public int sub(int value1, int value2){
        return value1 - value2;
    }
}

class Math2 extends Math1 {
    public int add(int value1, int value2){
        return square(value1) + value2;
    }
    private int square(int value){
        return value * value;
    }
}

class Math3 extends Math2 {
    @Override
    public int add(int value1, int value2){
        return value1 * 2 + value2 * 2;
    }
}
```

Listing 4.3: An inheritance chain with only a single reused method in Math3.

When applying the formula for the minimal CMF, we have to be aware that it is a rough estimation. The formula does not consider the memory footprint of the instructions within the method bodies. Moreover,—for the sake of simplicity—we disregard the overhead for annotations, the special properties of constructors (see on page 67), and the additional overhead for inheritance levels larger than eight. We assume that all parameters and fields are of the data type **int**.

Table 4.1 on the next page shows the increase in CMF per inheritance level. In this example calculation, we assume that each class has two **int** fields and three methods with each one **int** parameter.[4]

The instance memory footprint of a subtype class comprises of the memory footprint of its header, its fields, and the fields of its ancestor classes. It does not matter whether a field of an ancestor class cannot be accessed from a class instance. For example, the memory footprint of an instance of `MathField2` in Listing 4.4 on the following page is 32 byte. The three **double** fields consume thereof 24 byte, although the **private** field `minus` is not accessible from a `MathField2` instance. We can compute the instance memory

[4]We want to emphasize that the shown increase in CMF is based on the assumption that none of the ancestor classes is loaded into the non–heap area of the JVM when the corresponding subclass is initiated for the first time. This circumstance is not necessarily the case in practice.

Inheritance depth	Deep CMF
1	1,594
2	3,188
3	4,782
4	6,376
5	7,970
6	9,564
7	11,158
8	12,752
9	14,346
10	15,940

Table 4.1: The increase in CMF (in byte) per inheritance level.

footprint of a subtype class s for a by:

$$s = h + \sum_{i=2}^{d} f_i$$

Whereas h is the memory consumption of the instance header and f_i is the IMF of the fields of corresponding class in the inheritance chain. We explain how to compute f_i for different data types on HotSpot JVMs on page 17. Applying the formula to a HotSpot 6 VMIM, we get:

$$s = 8\,\text{byte} + \sum_{i=2}^{d} (n_{f_i} * 24\,\text{byte})$$

Whereas d is the inheritance depth of the class, n_{f_i} is the number of **int** fields of corresponding class in the inheritance chain.

```
1 class MathField1 {
2     protected double thousand = 1000;
3     private double hundred = 100;
4 }
5
6 class MathField2 extends MathField1 {
7     protected double thousand = 1000d;
8 }
```

Listing 4.4: The inheritance of fields.

Eventually, we can estimate the total memory footprint of a subtype class e by:

$$e = k + n_I * s$$

Whereas k is the class memory footprint of the subtype class and its ancestor classes, n_I is the number of instances at a time, and s is the memory consumption per instance of the subtype class. Be aware that k and s must be a multiple of eight. If the computed values are not multiples of eight, we must round up them to the next multiple of eight.

Delegation

The class memory footprint of a composed class consists of its own footprint plus the memory consumption of its component. The CMF of two *class torsos* is 1856 byte (i.e., 928 byte per class). A component is referenced by a field in the containing class. As we explain in Section 2.5.2 on page 18, each reference field consumes at least 24 byte per class. The field allocation costs additional 32 byte. Assuming that both the containing and the contained class inherit only from `java.lang.Object`, the deep class memory footprint of a composed class is 1912 byte.

The composed class mostly makes use of the methods of its component. Consequently, the minimal memory footprint is increased by the methods of the component and the *forwarding methods* that invoke them in the containing class. The memory footprint of a *forwarding method* consists of its torso, its parameters, and the call to the destination method. In Section 4.2 on page 56, we show that we can compute the CMF of a *forwarding method* (notated as w) by:

$$w = t + n_P * (p + v_P) + v$$

Whereas t is the footprint of the method's torso, n_P is the parameter number of the method, p is the memory consumption per parameter, v_P is the memory consumption of a parameter in a method invocation, and v is the footprint of the method invocation instruction. If we apply the computed values for an interpreted–mode HotSpot 6 VM, we get following formula:

$$w = 176\,\text{byte} + n_P * (30\,\text{byte} + 4\,\text{byte}) + 32\,\text{byte}$$

The need of *forwarding methods* is apparently a disadvantage of composition compared to inheritance. It increases the memory footprint. For instance, seven *forwarding methods* with one **int** parameter consume about 1.65 kilobyte. Such methods are not necessary in a subtype class, because a subclass can intrinsically invoke the (for it visible) methods of its ancestor classes. Consequently, a composed class has a larger CMF than a non–refactored subclass with an inheritance depth of two.

Accordingly, composition has disadvantages with respect to the instance memory footprint. First, each wrapper instance must hold a reference to the component instance. A reference field consumes 4 byte per class instance. Second, the instantiation of the component class is necessary to use its methods.[5] The IMF of the component object consists of the memory consumption of its header and its fields. In the case of inheritance, only the fields of the superclasses are initiated. Taken together, the deep IMF of a composed object is 12 byte larger than that one of its counterpart with an inheritance depth of two.

Comparison

Figure 4.2 on the following page exhibits that the subtype class has a lower CMF until an inheritance depth of four. For inheritance depths greater than four, the CMF of a composed class is smaller. The reason for this circumstance is that a subtype class needs

[5] Only if the methods of the component are static, the instantiation can be avoided.

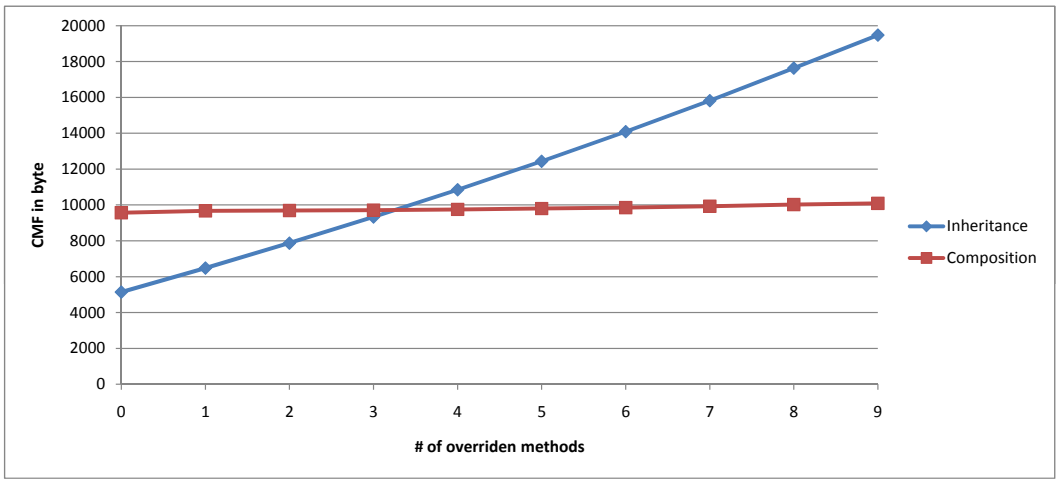

Figure 4.2: Comparison between the CMF of an inherited class and the CMF of a composed class. The inheritance depth of the inherited classes equals the number their overridden methods plus one. For example, the inheritance–based class with four overridden methods has an inheritance depth of five.

neither *forwarding methods* nor a reference field to the class that it makes use of. Thus, an inheritance–based class is smaller for low inheritance levels. However, the greater the inheritance depth, the more the overhead of the *class torso* becomes apparent. The size of the *class torsos* remains constant for composed classes, but it increases steadily for inherited classes.

The apparent conclusion would be that we should avoid *Replace Inheritance with Delegation* for low inheritance levels, but apply the refactoring for higher inheritance depths. However, as we show in Table 4.2, the number of class instances influences considerably the total memory footprint (i.e., the CMF and the IMF together). Hence, it depends on the number of concurrently existing class instances whether a composed class reduces not only the CMF, but also the total memory footprint.

# of instances	IMF for inheritance	IMF for composition
1	48	104
50	2,400	5,200
250	12,000	26,000
5,000	240,000	520,000
100,000	4,800,000	10,400,000

Table 4.2: A comparison between the IMF (in byte) of the inheritance–based class instances and their composed counterparts.

Moreover, the example has a shortcoming. It is most likely unrealistic. First, instead of gradually override one method per inheritance level, we could override all methods that should behave differently in a subclass at once. Thus, the CMF of the subclass that is functionally equal to `SubClass9` would shrink to 9608 byte; it would be 440 byte smaller than its composed counterpart. Second, the composed classes are no real surrogates for the subtype classes in our example. For instance, the program containing

`ComposedClass3`, which is the refactored version of `SubClass3`, has no comparable classes to `SubClass1` and `SubClass2`. If the program needs the behavior of these classes (i.e., if it needs to initiate them), the refactoring breaks it. To overcome this problem, we have to introduce a composed class for `SubClass1` and `SubClass2`. In this case, the CMF of the refactored program is about 267.5 percent larger than the CMF of its inheritance–based counterpart.

A subclass consumes less memory than a composed class in the most cases. Is it therefore meaningful to *Replace Delegation with Inheritance?* The answer depends on the RAM resources of the target device (i.e., the device running the program). If numerous instances of a class exists simultaneously and they consume a considerable stack of the device's RAM (e.g., more than 1 percent), we should consider *Replace Delegation with Inheritance* as a way to reduce the IMF. However, *inheritance* gives rise to disadvantages such as the fragility of the subclasses [Blo08].

In addition, it can be useful to introduce an artificial superclass. An artificial superclass has no genuine child–parent relationship to its subclass. In Listings 4.5 on the following page and 4.6 on the next page, we show such an example. Even though neither `Harddrive` nor `CPU` has a parent–child relationship to `Computer`, we amalgamate both classes into the superclass `Peripherals`. The CMF of the `Computer` class that inherits from `Peripherals` is 1776 byte smaller than the CMF of the composed `Computer` class. Furthermore, the deep IMF per subtyped instance of `Computer` is 24 byte smaller than its composed counterpart. Consequently, the introduction of the `Peripherals` class saves about 2.29 megabyte if there are 100,000 instances of `Computer`.

Beside its positive effect on memory footprint, the introduction of an artificial superclass brings some problems. First, a Java class can only inherit from a single superclass. The refactoring of a class inheriting from a class other than `java.lang.Object` can be difficult or not possible at all. Second, the introduction of artificial superclasses violates object—orientation. The resulting classes have often bad software quality characteristics (e.g., they are difficult to maintain). Therefore, an artificial superclass should be only used when other memory–saving techniques such as *object pooling* do not yield the desired outcome.

4.1.3 Conclusion

The analysis proves the assumption we had at the beginning of this section—*Replace Inheritance with Delegation* increases in the most cases the memory consumption of a Java program. Thus, we discourage from applying the refactoring when programming for devices with scarce RAM.

Even if the computation of the CMF shows that *Replace Inheritance with Delegation* reduces the non–heap memory consumption, this benefit can be negligible in comparison to the caused increase of the IMF. For instance, in the example program, a composed (i.e., refactored) class is 9608 byte smaller than a behaviorally equal class with an inheritance depth of nine. However, if on average 10,000 instances of that class exist concurrently, the IMF of the composed class is 560,000 byte *larger* than the IMF of the inheritance–based class. A refactoring decision based on the CMF would thus be misleading. To prevent such wrong decisions, the profiling of both versions under production conditions is necessary.

```
class Computer {
    private Harddrive harddrive
              = new Harddrive();
    private CPU cpu = new CPU();

    void write(byte data){
        harddrive.write(data);
    }
    byte read(){
        return harddrive.read();
    }
    byte compute(byte instruction){
        return cpu.compute(instruction);
    }
}

class Harddrive {
    void write(byte data){
        /* some instructions */
    }
    byte read(){
        byte data = 0;
        /* some instructions */
        return data;
    }
}

class CPU {
    byte compute(byte instruction){
        byte result = 0;
        /* some computations */
        return result;
    }
}
```

Listing 4.5: A composed `Computer` class.

```
class Computer extends Peripherals {
    /* inherits all methods from
    the class Peripherals */
}

class Peripherals {
    void write(byte data){
        /* some instructions */
    }
    byte read(){
        byte data = 0;
        /* some instructions */
        return data;
    }
    byte compute(byte instruction){
        byte result = 0;
        /* some computations */
        return result;
    }
}
```

Listing 4.6: A `Computer` class that inherits from an artificial superclass.

In contrast, *Replace Delegation with Inheritance* (the counterpart of *Replace Inheritance with Delegation*) seems to be an effective way to reduce the memory consumption of classes with a large instance number if the application is subject to tight memory constraints. Moreover, the introduction of an *artificial superclasses* can be beneficial to reduce the memory footprint of a program. However, if we do not face a memory shortage, we should apply the reuse technique that is appropriate to our use case. In Section 2.1.3 on page 8, we show that *Delegation* has several advantages compared to inheritance.

Another point to consider with respect to memory consumption are other JVM languages than Java. If we program for devices with low RAM, we have to be aware that those languages can blow up the memory consumption. For instance, Clojure (a JVM–based Lisp dialect) uses composition to allow a higher program adaptability at runtime. Its compiler generates an own class for each method of a class [Hal09]. Thus, a program written in Clojure has often a larger memory footprint than a comparable Java program.

4.2 Inline Method

Method inlining is the replacement of a method call with the body of the called method. It is not only useful to improve the runtime performance of programs, it can also reduce

the memory footprint of them. For instance, the Closure Compiler from Google[6] uses inlining of functions to reduce the size of JavaScript files. Accordingly, we analyze in this section the influence of *Inline Method* on memory footprint in Java.

A more detailed introduction to *Inline Method* can be found in Section 2.1.3 on page 8.

4.2.1 Methodology

To reveal how much *Inline Method* decreases the footprint of a program, we have to determine the influencing variables. First, we analyze the memory footprint of a *method's torso*. A torso of a method comprises of its signature, visibility modifier, and return type. We compare the effect of different visibility modifiers (**public**, **protected**, **private**, and without) and analyze the influence of the return type. Second, we examine the effect of parameters on the memory footprint. For that purpose, we benchmark methods with a parameter number of 0 to 5. Third, we analyze the memory footprint of a method invocation instruction. A method invocation instruction is a statement like `callMethod1()`.

In the end, we develop a formula by which we can estimate how much *Inline Method* reduces the memory footprint of a Java program. To test the formula, we measure the memory footprint of twelve benchmark programs. The programs contain 5 to 30 methods that call each other in a delegation chain. The methods do nothing else. For each delegation chain length, there are benchmarks with one, three, and five supplied **int** parameters. Consequently, we have 15 benchmark programs.

First, we measure the memory footprint of these programs. Second, we inline all methods in the delegation chain other than the one that is called initially. For instance, if we have five methods (`method1`, `method2`, and so on) in a class, only `method1` remains in the class. The other methods are all inlined into `method1`. After the inlining, we measure the memory footprint of the refactored program. Third, we compare the measured decreases in memory footprint to the estimated decreases by our derived formula.

4.2.2 Results

Figure 4.3 on the following page presents the memory footprint of parameterless **void** methods without any instructions in their bodies. Every empty method increases nearly linearly the footprint of a class. The increase per method is 176 to 184 byte. Hence, we assume that a *method torso* requires 176 byte. An empty class requires about 928 byte. This means that a *method torso* increases the memory footprint of these classes by about 19 percent. Even empty methods can thus significantly increase the class memory consumption. For example, five empty methods nearly double the memory footprint of a class—they increase it by 95 percent.

The benchmarks reveal that the change of the visibility modifier (**public**, **protected**, **private**, and **package-private**) has no influence on the method footprint. The same is true for other return types than **void**. Consequently, *Method Voiding* (i.e., the refactoring of a method's return type to void) causes no benefit in terms of memory consumption.

[6] http://code.google.com/closure/compiler/

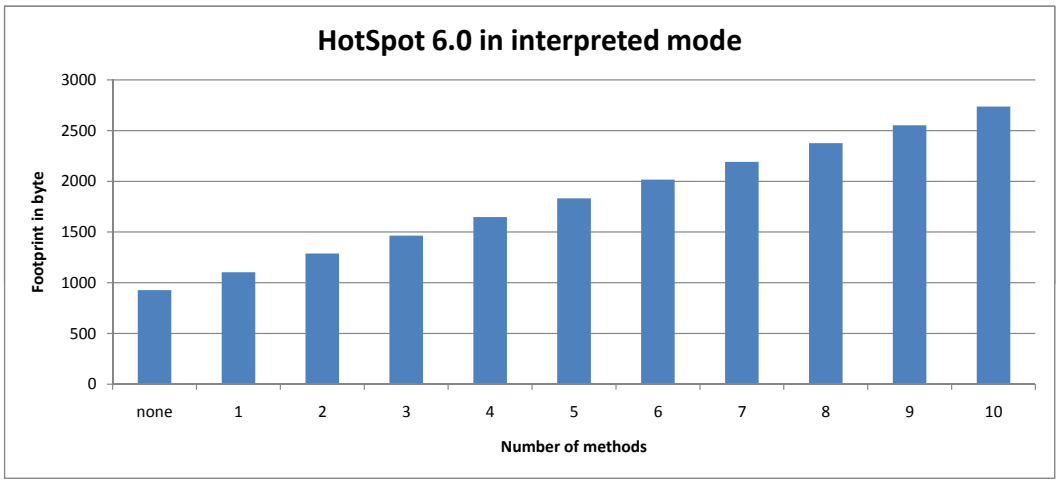

Figure 4.3: The increase of memory footprint of an empty class by adding empty methods.

Nonetheless, we have to emphasize that although the return *type* does not influence the memory footprint, the return *value* can. For instance, a constant return value (i.e., a return value that is clear at compile time like **return** `13;`) is saved within the *constant pool* of the class [LY99]. Consequently, it consumes additionally memory.

To figure out the effect of the parameter number, we measure different combinations of method and parameter numbers. The results are stated in Table 4.3 on the next page. We compute the memory footprint per parameter (notated as p) for each measurement $x_{j,a}$ by:

$$p_{j,a} = \frac{x_{j,a}}{j * a}$$

Whereas j is the parameter number and a is the number of methods. Subsequently, we compute a set of all memory footprints per parameter:

$$P = \{p_{1,1}, p_{1,2}, \ldots, p_{h_j,h_a-1}, p_{h_j,h_a}\}$$

Whereas h_a is the highest number of attributes and h_j is the highest number of parameters in our measurement. Then, we can determine the minimal memory footprint of a parameter by:

$$p_{min} = \min(P)$$

On an interpreted–mode HotSpot VM 6 and for our measurements, p_{min} is 30 byte. That is, each **int** parameter increases the memory footprint of a Java program at least by 30 byte.

Although the focus of this study is the parameter type **int**, we want to emphasize that the footprint of a parameter depends on its type. In Figure 4.4 on the facing page, we show the influence of different parameter types on the memory footprint of a method. Each example program contains a method with 20 parameters. Every method has only parameters of the same type. We can observe that parameters of the types `java.lang.Object`, **double**, and **long** have a considerably larger memory footprint than parameters of the type **int**. The overhead for these data types is 17 to 19 byte per parameter. This means that not only the number, but also the type of the parameters

# of Methods \ # of Parameters	0	1	2	3	4	5
1	1,104	1,144	1,184	1,216	1,248	1,304
2	1,288	1,360	1,432	1,488	1,560	1,648
3	1,464	1,568	1,672	1,768	1,872	1,984
4	1,648	1,776	1,928	2,040	2,192	2,328
5	1,832	1,984	2,168	2,320	2,496	2,672
6	2,016	2,200	2,416	2,592	2,816	3,016
7	2,192	2,408	2,664	2,872	3,128	3,352
8	2,376	2,616	2,912	3,144	3,440	3,696
9	2,552	2,824	3,160	3,424	3,752	4,040
10	2,736	3,040	3,408	3,696	4,064	4,384

Table 4.3: The memory footprint (in byte) of a class with an increasing number of methods and parameters per method.

affects the memory footprint of a class. In addition, it shows that the calculation of the decrease in memory footprint through *Inline Class* is more complex in practice—methods have normally not only parameters of the type **int**.

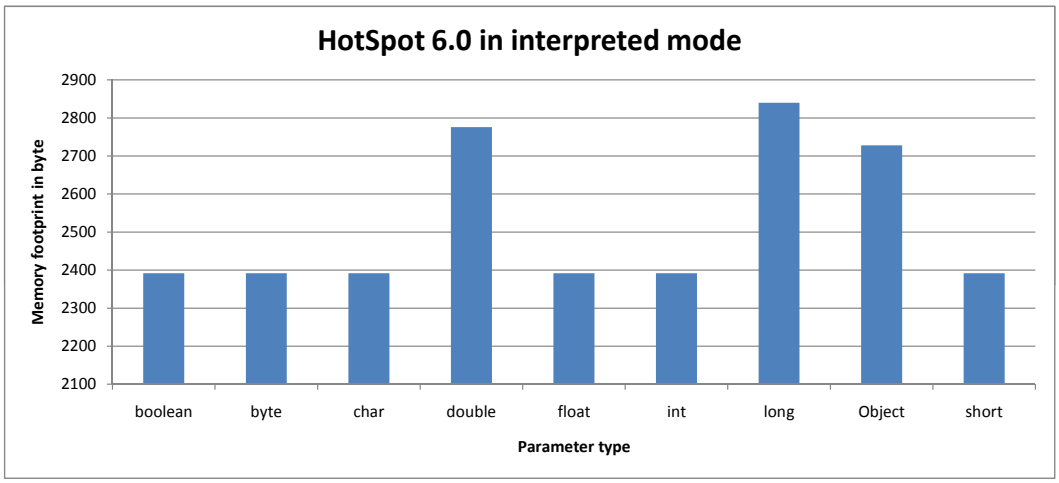

Figure 4.4: A comparison of the memory footprints of parameters with different types.

Beside a substantial influence of the method parameters, the benchmarks reveal also a significant effect of the method invocation instruction on memory footprint. In Table 4.5 on page 61, we exhibit that a call instruction of a method consumes additional memory. It increases the memory consumption by at least 32 byte. Moreover, each supplied **int** parameter consumes additionally 4 byte. Hence, we can compute the memory footprint of a method call instruction a for an IM HotSpot VM 6 by:

$$a = n_P * 4\,\text{byte} + 32\,\text{byte}$$

and in general by:

$$a = n_P * p + v$$

Whereas v is the memory footprint of the method invocation, n_p is the number of supplied **int** parameters, and p is the memory consumption of each supplied parameter.

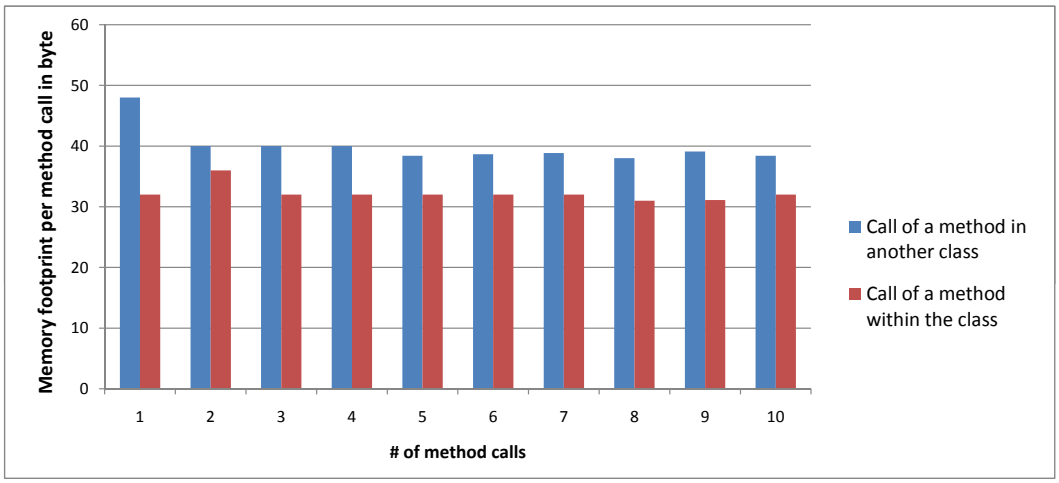

Figure 4.5: A comparison of the memory footprint of a method invocation within the class and outside of it.

With respect to the memory footprint of a method invocation, several circumstances are noticeable. First, our benchmarks indicate that a method invocation instruction is more expensive if we call a method in another class (see Figure 4.5). Second, the memory consumption of a method invocation instruction is only as large as shown in Table 4.5 on the next page for a single invocation of a method. If the method is called several times within a class, the memory footprint of a does not multiply. Instead, the memory footprint increases considerably less, as shown in Table 4.4. However, we assume—for the sake of simplicity—that a method invocation consumes 32 byte in our calculation. Third, the overhead per supplied parameter depends on its data type. For instance, 10 method invocations with each five supplied parameters of the types **double** and **long** consume 16 byte more memory than the same method invocations with parameters of the type **int**.

# of method invocations	Memory footprint	Memory footprint per additional invocation
none	2,736	-
1	2,768	32
2	2,776	8
3	2,776	0
4	2,784	8
5	2,792	8

Table 4.4: The memory footprint (in byte) per additional invocation instruction of the same method.

To summarize, the variables that influence the memory footprint of a method to inline (MTI) are: (1) its torso, (2) its parameter number, (3) the type of those parameters, and (4) the invocation of it. Therefore, we can calculate the decrease in memory footprint through method inlining by:

$$i = t + n_P * (p + q) + v$$

# Method Calls \ # Parameters	0	1	2	3	4	5
1	1,096	1,104	1,112	1,112	1,120	1,120
2	1,128	1,144	1,160	1,168	1,176	1,184
3	1,168	1,192	1,208	1,216	1,232	1,240
4	1,208	1,232	1,256	1,272	1,288	1,304
5	1,248	1,272	1,296	1,312	1,336	1,352

Table 4.5: The memory footprint (in byte) of a method call instruction for a varying number of methods and parameters.

Whereas t is the footprint of the MTI's torso, n_P is the parameter number of the MTI, p is the memory consumption per parameter, q is the memory consumption of a parameter in a method invocation, and v is the footprint of its invocation instruction. If we apply the computed values, we get following formula for HotSpot VMIM 6:

$$i = 176 + n_P * (30\,\text{byte} + 4\,\text{byte}) + 32\,\text{byte}$$

We have to note that the benefit `i` can only be achieved if the method is inlined into a single method. If the method is inlined into several methods, it depends on the memory footprint of its body (notated as u) whether *Inline Method* decreases or increases the memory consumption of a Java program. Consequently, $i > u$ must be true in order that method inlining is beneficial with respect to memory consumption if a method is inlined into multiple methods. Furthermore, to obtain a benefit of i, it is necessary that the inlined method is deleted after the inlining of its body. Otherwise, the benefit is reduced by the memory footprint of the inlined method's declaration.

Another fact to consider is that i is independent of the class instance number. That is, the refactoring affects only the non–heap memory consumption. The IMF of a class is not changed by *Inline Method* as well as *Extract Method*.

Example

Parameter Number:	1		3		5	
No. Inlined Methods	Measured	Computed	Measured	Computed	Measured	Computed
4	976	968	1,240	1,240	1,528	1,512
9	2,192	2,176	2,792	2,784	3,440	3,400
19	4,632	4,584	5,888	5,872	7,280	7,168
29	7,072	6,992	8,992	8,968	11,120	10,936

Table 4.6: A comparison between the actual and the computed memory footprint decrease (in byte) with a varying number of inlined methods and parameters.

In Table 4.6, we show the computed and measured reductions in memory consumption due to method inlining. The computed values deviate from the measured values by 0 to 184 byte. All except one computed reductions in memory footprint are too pessimistic.

That is, the estimations for the decrease in memory consumption are too low for them. However, this underestimation of the effect of *Inline Method* is justifiable. First, we chose deliberately a pessimistic approach (see Section 2.5 on page 16). Second, the computed values give anyway an eligible outlook of the influence of the refactoring. The aberration from the measured values is at most 1.7 percent. Consequently, we can use the formula to estimate the decrease in memory footprint as shown in Figure 4.6.

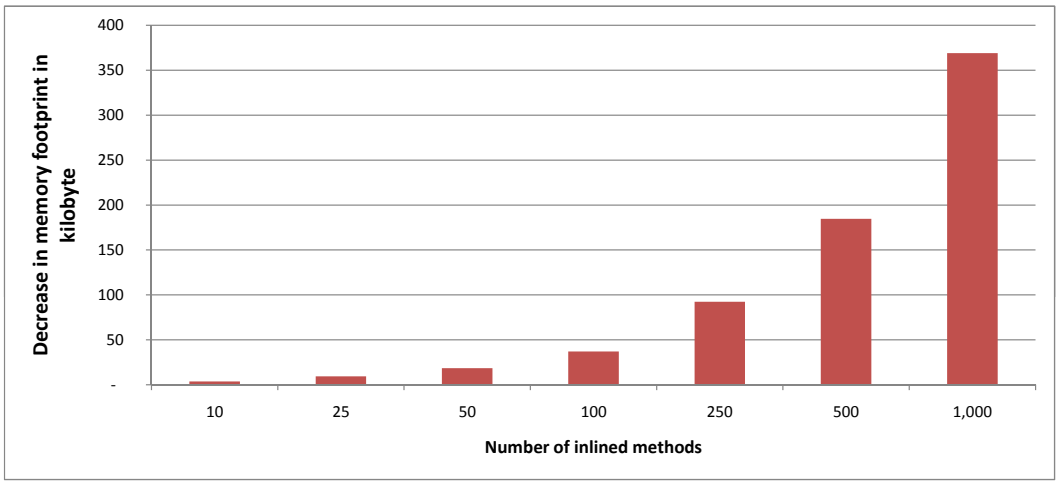

Figure 4.6: The estimated decrease in memory consumption of a Java program through method inlining.

4.2.3 Conclusion

The benchmarks reveal that *Inline Method* reduces approximately linearly the memory footprint of a Java class. Because an empty method requires minimally 176 byte on a HotSpot VM 6 in interpreted mode, each method inlining reduces the memory footprint at least by this magnitude. In fact, the refactoring normally decreases more significantly the memory footprint; the parameters of a method and its call instruction consumes additional memory. The invocation of a method consumes at least 32 byte. The memory consumption of a parameter depends on its type. For example, one **int** parameter increases the memory footprint of a method by 30 byte and the method invocation instruction by 4 byte. Thus, the inlining of a method with one parameter of the type **int** reduces the memory footprint of a Java program by 242 byte.

A decrease in this order of magnitude may be negligible for systems running on gigabytes or even terabytes of RAM. However, the decrease of memory footprint is beneficial for devices with scarce RAM (e.g., a RAM size below 1 MB). For instance, the inlining of 35 methods with 5 parameters of the type **int** reduces the memory footprint of a program by about 12.9 kilobyte on an IM HotSpot VM 6. For a device with a RAM size of 512 kilobyte, the inlining of those methods would save 2.5 percent of its total RAM. In addition, the actual reduction in memory footprint is supposably larger. The benchmarks on the preceding page show that our formula may underestimate the influence of *Inline Method*.

Even though the decrease in memory consumption seems promising, it is only as large under the condition that a method is inlined into a *single* method and then deleted. That

is, the method to inline must be exclusively used by the absorbing method to cause the described decrease in memory consumption. Otherwise, *Inline Method* can even increase the memory footprint of a Java program. The programming style of the developer is thus decisive for the effect of the refactoring. If the developer used a lot of small helper methods in the application to refactor, *Inline Method* can be expedient to reduce its memory footprint.

Inline Method can have also other disadvantages. For instance, method inlining can cause cache misses if the resulting method is too large for the cache. In this case, the refactoring degrades the performance of a program. We present a more detailed discussion about the advantages and disadvantages of *Inline Method* on page 36.

4.3 Inline Class

In Section 2.1.3 on page 8, we introduce the *Inline Class* refactoring. To recap, a class is inlined into another class if its methods and fields are moved to the other class. Because an inlining of a class makes needless the torso of the inlined class and the reference to it as well, we suppose that *Inline Class* reduces the memory footprint if the inlined class is exclusively used by the absorbing class. Thus, we analyze in this section whether this assumption is true or not.

4.3.1 Methodology

To predict the reduction in memory consumption, we have to determine the influencing variables. We analyze the memory footprint of the *class torso*, the *forwarding methods*, and other possible factors before and after a class inlining. This way, we develop a formula to estimate the decrease in memory footprint due to *Inline Class*. Finally, we test the formula by comparing the measured and estimated values. For that purpose, we create wrapper classes that contain one to three classes. There are wrapped classes with one, three, and five getter methods. The getter methods return **int** values that are previously supplied to the constructor of the wrapped class. That is, a wrapped class with five getter methods has a constructor with five **int** parameters. The wrappers access these methods by *forwarding methods*. There are totally nine wrapper classes. We measure the memory footprint of these classes and the classes they reference to. Afterwards, we inline the contained classes into the (former) wrappers and measure the memory footprint again.

4.3.2 Results

On an IM HotSpot VM 6, the CMF a *class torso* (notated as g_C) is about 928 byte. The IMF of the *class torso* is 8 byte. Thus, the overhead for the *class torso* (notated as g) is its IMF multiplied by the number of class instances (n_I) plus its CMF:

$$g = g_I * n_I + g_C$$

A class is normally loaded into the RAM, when one of its constructors is called the first time [HC08]. Thus, we have to consider also the memory footprint of an instruction

# of assignments	Total memory footprint	Memory footprint per additional assignment
0	1,032	-
1	1,056	24
2	1,088	32
3	1,112	24
4	1,152	40
5	1,176	24

Table 4.7: The memory footprint (in byte) of a field assignment on a HotSpot 6 VMIM.

like **new** `ClassInstance()`. In Table 4.10 on page 69, we show that a call of an empty constructor using the keyword **new** consumes at least 32 byte. *Inline Class* makes dispensable the *class torso* as well as the constructor call. Consequently, the refactoring reduces the memory footprint of a Java program by at least 968 byte.

However, the decrease in memory footprint through *Inline Class* is normally larger than 968 byte. The absorbing class holds usually a reference field to the class to inline. Hence, the inlining shrinks the memory footprint by additional 52 byte. A reference field consumes at least 24 byte per class (see Table 2.2 on page 19) and 4 byte per class instance. The field assignment has a memory footprint of 24 byte per class (see Table 4.7). This means that we can compute the memory consumption of a reference field r for a HotSpot VMIM 6 by:

$$r = 24\,\text{byte} + 24\,\text{byte} + n_I * 4\,\text{byte}$$

and for the Java virtual machines in general by:

$$r = r_C + a + n_I * r_I$$

Whereas r_C is the class memory footprint of the field that references to the class to inline, a is the class memory consumption of the field assignment, n_I is the number of class instances, and r_I is the instance memory footprint of the field that references to the class to inline.

Apparently, the number of class instances influences significantly the memory consumption of a reference field. For instance, if there are at most five instances of the class that holds a reference to the class to inline, the removal of the reference field saves only 0.07 kilobyte. By contrast, if there are 10,000 objects at one time, the removal of this field reduces the memory footprint by about 39.1 kilobyte. In this case, the deletion of the field would save 3.8 percent of the total memory of a device with 1 megabyte RAM.

If the absorbing class and the class to inline have a bidirectional relationship, their inlining implies even the deletion of two reference fields and the corresponding field assignments. That is, we have to multiply r by two. A bidirectional relationship means that both the absorbing class and the class to inline hold a reference to each other [ALR10].

Moreover, the refactoring can make obsolete further fields. If a field exists in both the absorbing class and the class to inline, the field is redundant after the class inlining. For example, the `getStreeNo()` method of `Customer` in Listing 4.7 on the facing page

```
public class Customer {
    private final String name;
    private final String streetNo;
    private final Address address;

    public Customer(String name, String streetName, String city, String streetNo) {
        this.name = name;
        this.streetNo = streetNo;
        this.address = new Address(streetName, city, streetNo);
    }

    public String getStreetNo() {
        return streetNo;
    }

    public String getStreetName() {
        return address.getStreetName();
    }
    /* further getter methods for name, city  */
}

class Address {
    private final String streetName;
    private final String city;
    private final String streetNo;

    public Address(String streetName, String city, String streetNo) {
        this.streetName = streetName;
        this.city = city;
        this.streetNo = streetNo;
    }

    public String getStreetName() {
        return streetName;
    }
    /* getter methods for city and
     * postal code
     */
}
```

Listing 4.7: The `Address` class can be inlined into the `Customer` class.

is frequently invoked. To avoid the invocation of `address.getStreetNo()` and, thus, to improve the performance, the `Customer` class holds itself the street number. If we inline the `Address` class into the `Customer` class, we need only a single `streetNo` field. The field type determine the decrease in memory consumption. For a fields with a primitive data type (like `streetNo`), we can compute the reduction by:

$$f_P = u_C + n_I * u_I$$

Whereas u_C is the CMF of the redundant field, n_I is the number of instances of the class to inline, and u_I is the heap memory footprint of the redundant field type per class instance. We present an overview of the primitive field types and their memory footprints on page 17. Because we analyze the memory consumption of **int** fields on HotSpot VMIM 6, we assume that:

$$f_P = 24\,\text{byte} + n_I * 4\,\text{byte}$$

If the redundant field references to an object, the memory overhead comprises of the class memory footprint of the reference field u_C, the memory consumption of the reference

field per instance u_I multiplied by the number of instances n_I, and the footprint of object that is referenced o. The corresponding formula is:

$$f_I = u_C + n_I * (u_I + o)$$

For instance, in Listing 4.8, each `Delivery` instance contains an own `Parcel` object. In addition, both classes have an own instance of `GregorianCalendar`. If we inline the `Parcel` class into the `Delivery` class, one `calendar` field and one instance of `GregorianCalendar` is redundant. The inlining of `Parcel` into `Delivery` causes thus an I_F of about 1828 byte for one instance (i.e., $n_I = 1$) on an interpreted–mode HotSpot VM 6. The memory footprint of the `Java.util.GregorianCalendar` instance (i.e., o) looms large on this memory reduction, because it and its field values consume 1800 byte.

```
1  import java.util.Calendar;
2  import java.util.GregorianCalendar;
3
4
5  public class Delivery {
6      private final Parcel parcel;
7      private final Calendar calendar;
8
9      public Delivery() {
10         this.parcel = new Parcel();
11         this.calendar = new GregorianCalendar();
12     }
13     /* further fields and methods*/
14 }
15
16 class Parcel{
17     private final Calendar calendar;
18
19     Parcel() {
20       this.calendar = new GregorianCalendar();
21     }
22     /* ... further fields and methods ... */
23 }
```

Listing 4.8: The `Parcel` class can be inlined into the `Delivery` class.

The elimination of one `calendar` field decreases substantially the memory footprint. For example, if there are 5000 instances of the `Delivery` class, *Inline Class* can reduce the footprint by about 8.6 megabyte only through eliminating the redundant `calendar` field. However, the decrease in memory footprint through eliminating a redundant field is in the most cases smaller, because an object consumes normally considerably less memory than a `GregorianCalendar` instance [Blo08].

Not only the fields of the inlined class influence the effect of *Inline Class* on memory footprint, also the deletion of *forwarding methods* reduces the memory consumption of a Java program. *Forwarding methods* are methods that do nothing else than delegate their invocation to another method. For example, the `getStreetName()` method of the

`Customer` class in Listing 4.7 on page 65 is a *forwarding method*. Its only purpose is the forwarding of its call to the `getStreetName()` method of the `Address` class.

In fact, the *forwarding methods* are not deleted. Rather, we replace the invocation of the called method in the *forwarding method* with the body of the called method. This means that we inline the body of the called method into the *forwarding method*. The reduction in memory footprint comprises of the overhead of called method's torso plus the call instruction of the invoked method. Thus, we can compute the decrease in memory footprint through inlining i as shown on page 61:

$$i = t + n_P * (p + v_P) + v$$

Whereas t is the footprint of the callee method's torso, n_P is the callee parameter number, p is the memory consumption per parameter, v_P is the memory consumption of a parameter in a method invocation, and v is the footprint of the callee invocation instruction. In Section 4.2 on page 56, we show that the footprint of a parameterless **void** method is at least 176 byte. A call instruction consumes minimally 32 byte. Consequently, the elimination of a parameterless *forwarding method* reduces the memory footprint of a class at least by 208 byte. This value can be considerably larger if the method has parameters.[7] A special type of method is the constructor. It is used to allocate a new instance of a class.[8] Due to its special properties, we cannot apply the formula for estimating memory footprint of methods to the constructors. For example, if there is only a single empty constructor in a class, and it is parameterless, the deletion of it does not save memory. The reason is that a Java compiler automatically adds a parameterless constructor at compile time if the Java class source file provides no constructor [HC08]. In Table 4.8, we exhibit that the class memory footprint of a constructor increases with a rising parameter number. On the HotSpot VM 6, the footprint increases at least by 32 byte per parameter.

# of Parameters	Memory consumption	Memory consumption per additional parameter
0	928	-
1	968	40
2	1,000	32
3	1,040	40
4	1,072	32
5	1,104	32
6	1,136	32
7	1,168	32
8	1,200	32
9	1,232	32

Table 4.8: The memory consumption (in byte) of a constructor with an increasing number of `Object` parameters on an interpreted–mode HotSpot VM 6.

In addition, a Java class can have more than one constructor. Although the constructor of a class cannot have different names, *constructor polymorphism* is possible

[7] Be aware that we assume in our formula that all parameters have the same footprint.

[8] In Java, there are several other ways to create an instance, but they are not considered in this thesis.

through the rest of its signature—the parameter type and number. For instance, a class `Abc` can simultaneously have the constructors `Abc()`, `Abc(String str)`, `Abc(int num)`, and `Abc(String str1, String str2)`. In Table 4.9, we show that every constructor has at least a memory footprint of 200 byte. Because each additional constructor has also an additional parameter in our example, we have to subtract the memory footprint for the parameters. Each parameter consumes at least 32 byte. Consequently, the minimal memory consumption of a constructor is 168 byte.

# Parameters	Total CMF	CMF for constructor & its parameters	CMF for parameters	CMF per constructor
1	1,128	200	32	168
2	1,160	232	64	168
3	1,200	272	96	176
4	1,232	304	128	176
5	1,288	360	160	200

Table 4.9: Memory consumption (in byte) of a constructor with an increasing number of `Object` reference parameters on an interpreted–mode HotSpot VM 6.

Like in the case of other methods, also the invocation instruction of a constructor consumes memory. Table 4.10 on the facing page presents that a parameterless invocation of a constructor consumes at least 32 byte. Every supplied parameter consumes 4 byte. Hence, we can compute the decrease in memory footprint per eliminated constructor (notated as e) through *Inline Class* by:

$$e = n_P * (32\,\text{byte} + 4\,\text{byte}) + 168\,\text{byte}$$

and for Java virtual machines in general by:

$$e = n_P * (p_C + v_{C_P}) + v_C + u$$

Whereas n_P is the number of constructor parameters, p_C is the class memory footprint per constructor parameter, v_{C_P} is the memory consumption per parameter in a constructor invocation, v_C is the memory footprint of the constructor invocation, and u is the memory footprint of the constructor.

To summarize, the effect of *Inline Class* on memory consumption is influenced by the memory footprint of:

- the torso of the class to inline,
- the field that references to the instance of the class to inline,
- the fields that exits both in the absorbing class and in the class to inline,
- the methods that forward their calls to the corresponding methods in the class to inline,
- the constructors of the class to inline, and
- the parameters of the methods and constructors.

# of constructor invocations	Memory consumption	Memory consumption of the additional constructor invocation
0	872	-
1	904	32
2	952	48
3	992	40
4	1,032	40
5	1,072	40
6	1,112	40
7	1,152	40
8	1,192	40
9	1,224	32
10	1,272	48

Table 4.10: Memory consumption (in byte) of a constructor invocation using the keyword **new**.

Thus, we can compute the benefit of *Inline Class* (notated as l_C) in terms of CMF by:

$$l_C = g_C + r_C + a + u_C + \sum_{y=1}^{n_M} (i_y) + \sum_{z=1}^{n_C} (e_z)$$

Whereas g_C is the memory consumption of the *class torso*, r_C is the class memory footprint of the field that references to the class to inline, a is the class memory consumption of the field assignment, u_C is the memory footprint of the redundant fields, n_M is the number of inlined methods, i_i is the decrease in memory footprint through the inlining of the corresponding method, n_C is the number of inlined constructors, and e_y is the decrease in memory consumption through the inlining of the corresponding constructor.

Accordingly, we can estimate the decrease in IMF due to *Inline Class* (notated as l_I) by:

$$l_I = n_I * (g_I + u_I + r_I)$$

Whereas n_I is the number of instances of the class to inline, g_I is the heap memory consumption of the *class torso*, u_I is the heap memory footprint of the redundant fields, and r_I is the heap memory consumption of the field that references to the class to inline.

The total memory benefit caused by class inlining l is thus:

$$l = l_C + l_I$$

To obtain the benefit of l, it is essential that a class is inlined into only one class. That is, the absorbing class must be the only class that uses the class to inline. If we inline a class into several classes, the duplication of its fields and methods can cause an increase in memory consumption. In such cases, it is important for an inlining decision whether the memory footprint of the not deleted fields and the remaining (i.e., not inlined) methods of the inlined class (notated as b) are bigger than l. If b is bigger than l, an inlining of a class is not beneficial.

Example

Table 4.11 depicts that we can estimate the decrease in CMF quiet exactly applying our formula. The computed values deviate at most by 5.2 percent from the measured value. The mean deviation of the estimated values from the measured values is 2.2 percent. In seven of nine cases, the computed values are too low. However, this underestimation of the effect of *Inline Class* is justifiable. We chose deliberately a pessimistic approach (see Section 2.5 on page 16).

Inlined classes:	1		2		3	
# Inlined methods per class	Measured	Computed	Measured	Computed	Measured	Computed
1	1,272	1,224	2,536	2,512	3,784	3,880
3	1,688	1,640	3,376	3,344	5,072	5,128
5	2,168	2,056	4,264	4,176	6,384	6,376

Table 4.11: A comparison between the actual and the computed memory footprint decrease (in byte) due to class inlining.

The effect of *Inline Class* on the IMF is shown in Figure 4.7. The figure presents the decrease in memory footprint if we inline three classes. The shown decrease is linear to the class instance number. Moreover, the figure points out that for classes with a high instance number, the refactorings influences significantly more the IMF than the CMF. For instance, the inlining of three classes and 15 methods reduces the non–heap consumption by about 6.2 kilobyte. In contrast, the same inlining saves 75 kilobyte if there are 6400 instances of the absorbing class.

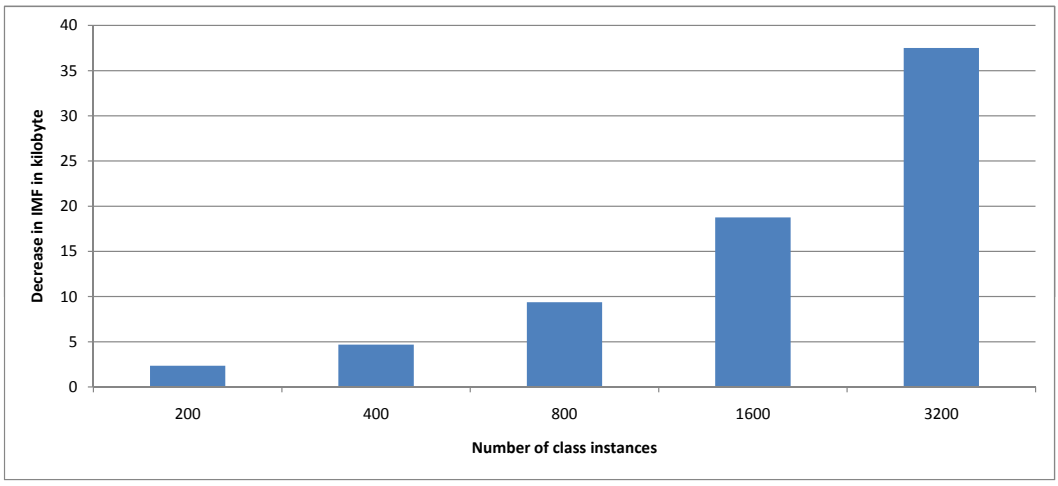

Figure 4.7: The decrease in IMF if three classes are inlined.

4.3.3 Conclusion

According to our formula, an inlining of a class can save about 391 kilobyte for 100,000 instances. If there are 260,000 instances of the refactored class, the memory footprint is reduced by 1 megabyte. Thus, the *Inline Class* might not be an effective refactoring to reduce the memory footprint of a machine running on gigabytes of RAM. The refactoring has a larger influence if it eliminates redundant objects that have a large memory footprint (see on page 66). However, the introduction of a **public static** field that can be accessed from both the absorbing class and the class to inline has the same positive effect without breaking encapsulation.

On devices with low RAM, *Inline Class* can be an important refactoring to decrease the memory consumption. For example, on a device with 10 megabyte RAM, 391 kilobyte corresponds to about 4 percent of its total RAM. However, the class instance number influences significantly the benefit of the refactoring. That is, if the number of concurrently existing class instances is low, the reduction in memory consumption is low. For example, the inlining of a class of which at most 10 instances exist concurrently, saves only about 2 kilobyte. Therefore, a profiling for suitable classes to inline may be needed. Alternatively, a good solution represents a tool that can automatically inline classes that are used by only a single class.

We present a more detailed discussion about advantages and disadvantages of *Inline Class* on page 43.

4.4 Summary

In this section, we give a short review about the results of Chapter 4.

Replace Inheritance with Delegation :

- Each inheritance level increases at least linearly the non–heap memory consumption.
- The CMF of a subclass consists of the CMF of its ancestor classes and its own CMF. Thereby, it does not matter whether the fields and methods of the ancestor classes are accessible in the subclass.
- Unlike a subclass, a composed class needs *forwarding methods* to give access to the methods of its (former) superclass. Consequently, these methods increase the CMF of a composed class.
- For low inheritance depth (in our example until a depth of four), a subclass has a lower CMF than a composed class.
- The IMF of a subclass consists of the heap memory consumption of the fields of its ancestor classes, its own fields, and its own header. It does not matter whether the fields of the ancestor classes are accessible in the subclass.
- Unlike a subclass, a composed class needs a reference field to the (former) superclass. Moreover, the instantiation of the component class is required. As a consequence, the deep IMF of a composed object is 12 byte larger than that one of its counterpart with an inheritance depth of two on a HotSpot VM.

- Because the influence of the refactoring depends largely on the instance number, profiling of the application to refactor under production conditions is needed to make sound decisions.
- The introduction of an artificial superclass can be beneficial in terms of memory footprint for devices with low memory resources. We can create an artificial superclass by applying *Replace Delegation with Inheritance* for classes that have no genuine parent–child relationship to the (former) wrapper class. Furthermore, the merging of several component classes into one artificial superclass is possible.

Inline Method :

- The effect of a method inlining on the memory footprint depends on the memory consumption of: the torso, the parameter number and types, as well as the invocation instruction(s) of the method to inline.
- The visibility modifier and the return type have no influence on the method footprint.
- An inlining of a method that is used by only a single method reduces always the memory footprint (if the inlined method is deleted after the inlining of its body).
- If a method is inlined into several methods, it depends on the memory footprint of its body whether the inlining is beneficial with respect to memory footprint. If the body has a smaller memory footprint than the components (e.g., the method torso and the method invocation instructions) that become obsolete by the inlining, the refactoring causes a decrease in memory footprint.
- *Inline Method* does not influence the heap memory footprint of a Java program. That is, it is independent of the class instance number.

Inline Class :

- The effect of a method inlining on the memory footprint depends mainly on the memory consumption of: the torso of the class to inline (CTI), the field that references to the instance of the CTI, the methods that forward their calls to the corresponding methods in the CTI, the constructors of the CTI, and the parameters of the methods and constructors of the CTI.
- The decrease in IMF due to *Inline Class* is nearly linear to the class instance number. Each created instance of the absorbing class has a reference field less after the inlining of a formerly referenced class. On a HotSpot VM, a reference field consumes 4 byte.
- An inlining of a class that is used exclusively by a single class reduces always the memory footprint (if the inlined class is deleted after the inlining of its body).
- If a class is inlined into several classes, it depends on the memory footprint of its body whether the inlining is beneficial with respect to memory footprint. If the body has a smaller memory footprint than the components (for example, the *class torso* and reference fields) that become obsolete by the inlining, the refactoring reduces the memory consumption.

The analysis proves partly the assumptions from the beginning of this chapter. *Replace Inheritance with Delegation* increases in the most cases the memory footprint of a Java program. However, if the inheritance depth of the refactored class is high (e.g., it has more than four inheritance levels), the refactoring can be beneficial in terms of memory consumption. *Inline Method* decreases the memory footprint if the method is used by only a single method. Also for a method that is inlined into several methods, the inlinings may be beneficial if the method's body is small. That is, the general assumption that the inlining of one method into multiple methods increases the memory footprint is not true. Accordingly, the claim that the inlining of one class into several classes increases generally the memory footprint is not true. If the class' body is small, the inlining may reduce the memory consumption of a Java program. Moreover, the inlining of a class that is exclusively used by a single class decreases the memory footprint.

Generally, the deviations in memory footprint caused by the refactorings are slight. In our benchmarks, the refactorings change the CMF on the magnitude of bytes. Accordingly, the effect of the refactorings on the IMF is small. Consequently, we can disregard the influence of the presented refactorings on memory consumption if we do not face a memory shortage.

Chapter 5

Evaluation

To evaluate the results of the micro—benchmarks with respect to performance on large scale, we applied the analyzed refactorings to the Java programs HyperSQL (HSQL), Kryo, and PrettyTime. For that purpose, we programmed two tools. First, we built an analyzer (called Janalyzer) that identifies candidates for refactoring and saves them into an XML file. Second, we developed a tool (called Jefactor) that applies the refactorings described in the XML file supplied by the Janalyzer. Both tools are based on the ASM framework [BLC02] for bytecode manipulation. They work on bytecode level. The bytecode–based approach has several advantages. First, we can refactor any program that is compiled to Java bytecode. For example, Jefactor can refactor applications written in Scala, JRuby, or Groovy. Second, the refactoring of applications is possible for which no source code is available (e.g., due to licensing reasons). Third, the maintained code base of a program is (explicitly) excluded from the refactoring. Consequently, non–functional properties that are related to the source code (such as readability and maintainability) are not affected by the applied refactorings. That is, the refactorings at bytecode level do not pollute the source code. Fourth, the optimization of the performance–related properties of a program takes place at the end of the development process. Therefore, premature optimization, which causes often problems [Knu74][Ray03], can partly be prevented [TS05]. Fifth, we could use a latter version of the tool to obfuscate the code (i.e., to prevent decompiling) or to perform other bytecode–related tasks such as erasing debug information.

The downside of the bytecode–based approach is, for example, that refactorings to meet coding style requirements (as suggested in [SKAP10]) are not applicable. In addition, we have to emphasize that the current version of the Jefactor is in a premature development state. It needs debug information (e.g., the *LocalVariableTable*) to apply the refactorings. If an application is compiled without debug information, the refactoring of it fails. Moreover, Jefactor can currently only apply the *Inline Method* and the *Inline Class* refactoring.

5.1 Benchmarking

To test the refactorings on large–scale programs, we chose the following three Java–based applications and frameworks:

- HyperSQL 2.0.0, a completely Java–based SQL database,
- Kryo 1.0.3, an object graph serialization program for Java, and
- PrettyTime 1.0.6, a time formatting library.

Table 5.1 shows the lines of code (LOC) and the percentage of comments. The micro–benchmarks in Chapter 3 on page 21 show that the refactorings are particularly useful to decrease the runtime of a non–JIT JVM. Such JVMs are mostly used at embedded and mobile devices; the applications running on these devices tend to be small. Consequently, we tested with Kryo and PrettyTime two tools that have a low LOC number.

We benchmarked HyperSQL using PolePosition[1]. To test the influence of the refactorings on HSQL, we used the workloads Melbourne and Bahrain which are included in the PolePosition benchmark suite.[2] For both workloads, we computed cumulated result values that ease the comparison of the normal and the refactored versions. That is, for the Melbourne benchmark, we cumulated the results of *write*, *read*, *read_hot*, and *delete* for each 100,000 objects. For the Bahrain benchmark, we cumulated the results of *write*, *query_indexed_string*, *query_string*, *query_indexed_int*, *update*, and *delete* for each 900 selects, 30,000 objects, and 100 updates.

Program	LOC	% Comments
HSQL 2.0.0	220,744	28.3
Kryo 1.0.3	5,249	18.1
PrettyTime 1.0.6	2,818	20.3

Table 5.1: The lines of code (LOC) of the tested programs.

We analyzed the influence of the refactorings on Kryo using the Eishay JVM serializers benchmark[3]. Furthermore, we used the PrettyTimeBench benchmark[4] to examine the influence of the refactorings on PrettyTime.

To prevent adulteration of the results due to random fluctuations in performance, we ran every benchmark 30 times. The results of all benchmarks are measured in ms.

5.2 Inline Method

Because Jefactor supports the inlining of methods, we had to overcome several problems regarding method inlining. Listing 5.1 on the facing page shows an example for method inlining. The `writeToField` method should be inlined into the `calculate` method. Therefore, Jefactor must replace the **return** statement in `writeField`. A method with return type **int** cannot return without supplying a value. Also, if the the `calculate` method would have the return type **void**, Jefactor must delete the **return** statement. The return statement would prematurely end the execution of the method `calculate` and, thus,

[1] http://polepos.sourceforge.net/

[2] Although both Melbourne and Bahrain are actually workloads of the PolePosition benchmark suite, we regard them as self-contained benchmarks in the following.

[3] https://github.com/eishay/jvm-serializers

[4] http://code.google.com/p/prettytimebench/

avoid (falsely) the execution of the `System.out.println(sum)` statement. Moreover, an **else** block must be added to the body of `writeToField`. Otherwise, the code following the **if** block would be executed even if the supplied `sum` value is smaller than zero. Listing 5.2 on the next page shows how the correct method inlining of `writeToField` must look like.

Because Jefactor applies the refactorings on bytecode level, it must handle bytecode–related obstacles, too. For instance, if `writeToField` would return a value, that returned value would not be used by the `calculate` method. Consequently, the Java compiler adds a `POP` instruction after the invocation of `writeToField` to throw away the unused returned value. However, this throwing away causes an error if the `writeToField` is inlined into `calculate` and the **return** statements of `writeToField` are deleted. In such cases, the `POP` instruction either throws away the wrong value or has no value to throw away. A detailed discussion about the handling of bytecode by the JVM is presented in [LY99].

The Jefator inlined all **private** methods of the tested applications.

```
1  public class InliningExample {
2
3          int sum;
4          int calculate(int val1, int val2){
5              int sum = val1 + val2;
6              writeToField(sum);
7              System.out.println(sum);
8              return sum;
9          }
10
11         void writeToField(int sum){
12             if(sum < 0){
13                 System.out.println("value␣is␣smaller␣than␣zero!");
14                 return;
15             }
16             this.sum = sum;
17         }
18 }
```

Listing 5.1: The calculate method before the inlining of writeToField.

5.2.1 Hypotheses

The focus of this section is to determine whether *Inline Method* improves the performance of a Java program. To focus our study, we set up the following hypothesis A. H_0 represents the null hypothesis and H_1 represents the alternative hypothesis of the null hypothesis:

- Hypothesis A

  H_0: The *Inline Method* refactoring does not improve the performance of a Java program.

  H_1: The *Inline Method* refactoring does improve the performance of a Java program.

```
1  public class InliningExample {
2
3          int sum;
4          int calculate(int val1, int val2){
5              int sum = val1 + val2;
6              if(sum < 0){
7                  System.out.println("value␣is␣smaller␣than␣zero!");
8              } else {
9                  this.sum = sum;
10             }
11             System.out.println(sum);
12             return sum;
13         }
14 }
```

Listing 5.2: The calculate method after the inlining of writeToField.

5.2.2 Results

HyperSQL

VM:	HotSpot 6 (client)		HotSpot 6 (Xint)		JRockit 6	
Benchmark:	Melbourne	Bahrain	Melbourne	Bahrain	Melbourne	Bahrain
Normal	2152.9	6486.5	2156.4	6397.2	2145.6	6393.9
Refactored	2144.0	6276.4	2135.1	6324.7	2141.3	6378.8
Benefit	0.41%	3.24%	0.99%	1.13%	0.20%	0.24%
Statistically significant	no	yes	yes	yes	no	no

Table 5.2: The results (in ms) before and after applying Inline Method for the Melbourne and the Bahrain benchmark.

In HyperSQL, the Jefactor can inline 121 methods. Table 5.2 shows that the refactored versions of HyperSQL score better in both the Melbourne benchmark and the Bahrain benchmark than the non–refactor version of HyperSQL on all test virtual machines. The performance benefit for the Melbourne benchmark is 8.9 ms for a HotSpot 6 VM in client mode to 21.3 ms for a HotSpot 6 VMIM. More significant are the performance gains for the Bahrain benchmark. Here, the decrease in runtimes are 15.3 ms for a JRockit 6 VM to 210.2 ms for a client–mode HotSpot 6 VM. The performance benefit on a HotSpot 6 VM in client mode is even bigger than on a HotSpot 6 VM in interpreted mode.

It is remarkable that the *Inline Method* refactoring has a larger influence on a VM with JIT compiler than on a VM without one—a JIT compiler can automatically inline often used methods after a while (see Section 2.2.1 on page 9). A possible explanation is that the benchmark runs too short and the JIT compiler is thus not able to optimize it.

To verify the results in a statistically sound manner, we conducted a significance test for the Melbourne as well as the Bahrain benchmark measurements. Table 5.3 on the facing page depicts that the differences in runtimes of the Melbourne benchmark

are only statistically significant for the HotSpot 6 VMIM. The confidence intervals for client–mode HotSpot 6 VM and JRockit 6 VM include zero. This means that we cannot rule out that the differences in runtimes for these virtual machines are caused by random fluctuations in performance.

VM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	8.9	5.1	19.9	-2.1
HotSpot 6 (Xint)	21.3	2.1	26.0	16.7
JRockit	4.3	3.3	11.5	-2.9

Table 5.3: The confidence intervals for the Melbourne benchmark results.

Table 5.4 shows that it is at our chosen confidence level of 0.97 assured that the difference in runtime of the HotSpot 6 VM in client mode and in interpreted mode are statistically significant. In accordance with results of the significance test for the Melbourne benchmark, the measured difference for JRockit 6 VM is statistically not significant.

VM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	210.2	40.3	297.7	122.7
HotSpot 6 (Xint)	72.5	12.2	99.0	46.0
JRockit	15.1	12.3	41.8	-11.5

Table 5.4: The confidence intervals for the Bahrain benchmark results.

To summarize, three of the six benchmarks of the refactored HyperSQL version show a statistically significant improvement. The refactoring can statistically significantly decrease the runtime of the Melbourne and Bahrain benchmark for the interpreted–mode HotSpot VM. Therefore, we can assume that the *Inline Method* refactoring has a positive influence on performance on this virtual machine. For the client–mode HotSpot 6 VM, *Inline Method* reduces statistically significantly only the runtime of the Bahrain benchmark. Although the results for the refactored versions of the JRockit 6 VM show an improvement, we could not verify that this improvement is caused by the *Inline Method* refactoring.

Kryo

The Jefactor can inline 23 method invocations in Kryo. Table 5.5 on the following page demonstrates that the refactoring improves the runtimes of the Eishay benchmark on the interpreted–mode HotSpot 6 VM and on the JRockit 6 VM. The improvement on a HotSpot 6 VMIM is with 808 ns the largest. On a JRockit VM, the improvement is only 31 ms. However, the Eishay benchmark runs on a JRockit 6 VM about 38.4 times faster than on a interpreted–mode HotSpot VM. Therefore, the relative performance improvements are comparable on both VMs (0.6 percent on a JRockit VM to 0.4 percent

on a HotSpot 6 VMIM). In contrast, the refactored Kryo version performs about 26 ms worse than the non–refactored Kryo version on a client–mode HotSpot 6 VM.

	HotSpot 6 (client)	HotSpot 6 (Xint)	JRockit 6
Normal	9506.9	201081.1	5236.5
Refactored	9533.1	200273.3	5205.7
Benefit	-0.28%	0.40%	0.59%
Statistically significant	no	yes	no

Table 5.5: The mean runtimes (in ms) of Eishay benchmark for the normal and refactored version of Kryo. Negative benefit percentages mean that the refactoring causes a performance degradation.

Table 5.6 shows that the difference in runtime between the refactored and the non–refactored version is only statistically significant for the interpreted–mode HotSpot 6 VM. Because the confidence intervals for HotSpot 6 VM in client–mode and JRockit 6 VM include zero, we cannot rule out that the difference in runtime for those virtual machines is caused by random fluctuations in performance.

VM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	-26.2	15.2	6.7	-59.1
HotSpot 6 (Xint)	807.8	74.1	968.6	647.0
JRockit	30.8	21.9	78.4	-16.8

Table 5.6: The confidence intervals for the sets of measurements of Kryo.

PrettyTime

In PrettyTime, the Jefactor inlines 12 method invocations. Table 5.7 on the facing page shows that the inlining of these methods only has a positive influence on the interpreted–mode HotSpot 6 VM. The mean runtime of the PrettyTimeBench is reduced by 523.1 ms (about 1.7 percent). On the contrary, the mean runtimes of the benchmark rises on the client-mode HotSpot 6 VM by 102.7 ms and on the Oracle 6 VM by 37.7 ms. Possibly, the manual method inlining prevents other optimizations performed by the JIT compilers of these virtual machines.

Table 5.8 on the next page exhibits that the differences in measurements are statistically significant for client–mode and interpreted–mode HotSpot 6 VM. Consequently, we can assume that the improvement and, respectively, the degradation of the runtime are caused by the refactoring *Inline Method*. By contrast, the confidence interval for the JRockit VM includes zero. This means that the differences in runtimes of PrettyTimeBench are not statistically significant on this VM.

	HotSpot 6 (client)	HotSpot 6 (Xint)	JRockit 6
Normal	2249.8	30301.8	1518.4
Refactored	2352.5	29778.7	1556.1
Benefit	-4.56%	1.73%	-2.48%
Statistically significant	yes	yes	no

Table 5.7: The PrettyTimeBench results in ms before and after applying method inlining. Negative benefit percentages mean that the refactoring causes a performance degradation.

VM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	-102.7	21.8	-55.5	-149.9
HotSpot 6 (Xint)	523.1	16.8	559.6	486.6
JRockit	-37.7	95.1	168.7	-244.0

Table 5.8: The confidence intervals for the sets of measurements of PrettyTime.

5.2.3 Summary

In five of twelve cases, applying the *Inline Method* refactoring could statistically significantly decrease the execution time of the benchmarks. The refactoring has particularly a positive effect on the interpreted–mode HotSpot 6 VM. On this virtual machine, the refactoring could improve the runtime of all tested benchmarks. Nonetheless, also for the HotSpot 6 VMIM, the performance benefit is only slight. It is between 0.4 and 1.1 percent. A more aggressive inlining heuristic could arguably gain the performance benefit. However, too large methods can degrade the performance, because they cause cache misses [DC94].

On the client–mode HotSpot 6 VM, the refactoring improves statistically significantly only the result of the Bahrain benchmark. In contrast, the result of the PrettyTimeBench is statistically significantly worse after applying the refactoring on this VM. All other benchmarks did not show any statistically significant differences before and after refactoring. Accordingly, the *Inline Method* refactoring seems to have no effect on the JRockit 6 VM. Presumably, the reason is that both HotSpot 6 VMCM and JRockit 6 VM have a JIT compiler that can automatically inline methods.

The benchmarks disprove the general claim that *Inline Method* improves the runtime of a Java program. As a consequence, we have to accept the null hypothesis of Thesis A. Nevertheless, the converse argument that *Inline Method* is generally futile is wrong as well—the benchmarks showed clearly that method inlining causes a performance gain on an interpreted–mode HotSpot 6 VM.

5.3 Inline Class

There are numerous obstacles to the inlining of a class in Java. The small obstacles are, for example, that a class is not allowed to have multiple fields with the same name

or methods with the same signature. Thus, Jefactor has to rename the affected fields and methods of the class to inline. If the absorbing class and the class to inline inherit both from other classes than `java.lang.Object`, the inlining is extremely complex or not possible at all. Java does not support multiple inheritance. In such cases, we have to end the inheritance relationship of the class to inline. That is, the Jefactor tool must copy the used methods and fields of its superclass(es) into the class to inline. Accordingly, if only the class to inline inherits from another class than `java.lang.Object`, the refactoring can be not possible if its superclass is located in another package than the absorbing class and is **package-private**. Moreover, we did not inline a class if the absorbing class gives other classes (its *clients*) access to a reference to the class to inline. An inlining of such a class would enforce also the change of the *clients*. Another obstacle for inlining a class is the use of reflective methods. For example, if a class named `Test1` that uses a reflective method invocation like **`this`**`.getClass().getName()` is inlined into an class called `Test2` the outcome of the method invocation would change from `Test1` to `Test2`. This change can break a whole application. During the development of the Jefacator, we found multitudinous similar traps that can break the code.

As a consequence, we applied the *Inline Class* refactoring in two steps. First, the Jefactor tried to apply the refactorings supplied by Janalyzer. Afterwards, a bytecode verifier (the `CheckClassAdapter` of the ASM framework) tested whether the change was successful. Second, we tried to manually apply the failed refactorings in the source code of the applications.

The analysis using Janalyzer showed that classes that are only used by one class and are no inner classes of the absorbing class do not exist in the programs of our case study. Consequently, we inlined all classes that are used by at most three other classes. If a method of the class to inline was used by only one method in the absorbing class, we also inlined the method of the class to inline into the respective method of the absorbing class.

5.3.1 Hypothesis

The focus of this section is to determine whether *Inline Class* improves the performance of a Java program. To focus our study, we set up the following hypothesis A. H_0 represents the null hypothesis and H_1 represents the alternative hypothesis of the null hypothesis:

- Hypothesis B

  H_0: The *Inline Class* refactoring does not improve the performance of a Java program.

  H_1: The *Inline Class* refactoring does improve the performance of a Java program.

5.3.2 Results

HyperSQL

Jefactor inlined 16 classes in HyperSQL. These class inlinings made possible the inlining of 27 method invocations. Table 5.9 on the facing page shows that the Melbourne benchmark results after applying *Inline Cass* are better on all virtual machines. For the

Melbourne benchmark, the performance benefit is between 4.4 ms on a JRockit 6 VM and 27.6 ms on a HotSpot 6 VMCM. In the case of the Bahrain benchmark, *Inline Class* improves the measured performance on a client–mode HotSpot 6 VM by 27.6 ms and on an interpreted–mode HotSpot 6 VM by 114.8 ms. By contrast, the performance on a JRockit VM degrades by 11.6 ms after applying the class inlinings.

VM:	HotSpot 6 (client)		HotSpot 6 (Xint)		JRockit 6	
Benchmark:	Melbourne	Bahrain	Melbourne	Bahrain	Melbourne	Bahrain
Normal	2152.9	6486.5	2156.4	6397.2	2145.6	6393.9
Refactored	2125.3	6373.1	2130.6	6342.7	2141.1	6405.5
Benefit	1.28%	1.75%	1.19%	0.85%	0.21%	-0.18%
Statistically significant	yes	yes	yes	yes	no	no

Table 5.9: The results for PolePosition (in ms) before and after applying Inline Class. Negative benefit percentages mean that the refactoring causes a performance degradation.

Table 5.10 shows the confidence intervals for the Melbourne benchmarks. The significance test proves at a confidence level of 0.97 that the differences in the measurements on the client–mode and on the interpreted–mode HotSpot VM is caused by the *Inline Class* refactoring. The improvement in the case of the JRockit 6 VM is not statistically significant. For the Bahrain benchmark, the results are accordant. The decreases in runtime are statistically significant for both tested HotSpot VMs, but not for the JRockit VM (see Table 5.11 on the following page).

VM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	27.6	5.3	39.0	16.2
HotSpot 6 (Xint)	25.8	3.0	32.2	19.4
JRockit 6	4.4	3.4	11.8	-2.9

Table 5.10: The confidence intervals for the Melbourne benchmark results.

To sum up, the Melbourne as well as the Bahrain benchmark perform statistically significantly better after applying *Inline Class* under our constraints on the tested HotSpot VMs. However, the performance improvement for both benchmarks together is only 141.0 ms (about 1.6 percent) on a HotSpot 6 VM in client–mode and 80.3 ms (about 0.9 percent) on an interpreted–mode HotSpot 6 VM. On the JRockit VM, the differences in measurement are statistically insignificant for the Melbourne and the Bahrain benchmark. That is, the performed inlinings have no measurable influence on the performance on this virtual machine.

Kryo

In Kryo, Jefactor could inline seven classes. These class inlinings made possible the inlining of six method invocations. After applying the refactoring, the Eishay JVM

VM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	113.4	40.3	200.8	26.0
HotSpot 6 (Xint)	54.5	16.3	89.9	19.1
JRockit 6	-11.6	8.1	6.0	-29.1

Table 5.11: The confidence intervals for the Bahrain benchmark results.

	HotSpot 6 (client)	HotSpot 6 (Xint)	JRockit 6
Normal	9506.9	201081.1	5236.5
Refactored	9513.0	196115.3	5190.8
Benefit	-0.06%	2.47%	0.15%
Statistically significant	no	yes	no

Table 5.12: The mean runtimes (in ns) of Eishay benchmark for the normal and refactored version of Kryo. Negative benefit percentages mean that the refactoring causes a performance degradation.

serializer benchmark runs faster on the interpreted–mode HotSpot 6 VM and on the JRockit 6 VM (see Table 5.12). The decreases in runtime are 4965.9 ms (about 2.5 percent) on the HotSpot 6 VMIM and, respectively, 7.6 ms (about 0.2 percent) on the JRockit VM. In contrast, the benchmark takes 6.1 ms longer after applying the refactoring on a client–mode HotSpot 6 VM.

Table 5.13 depicts that the improvement of the performance is statistically significant only for the interpreted–mode HotSpot 6 VM. Thus, we can assume at a confidence level of 0.97 that the applied class inlinings cause the performance gain on this VM. On the client–mode HotSpot 6 VM and on the JRockit 6 VM, the differences in measurements are statistically insignificant. That is, random fluctuations in performance cause possibly the slight deviations after applying the refactoring on these virtual machines.

VM	$\bar{x}$	s_x	c_1	c_2
HotSpot 6 (client)	-6.1	15.3	27.0	-39.2
HotSpot 6 (Xint)	4965.8	63.5	5103.7	4827.9
JRockit	7.6	18.2	47.0	-31.9

Table 5.13: The confidence intervals for the sets of measurements of Kryo.

PrettyTime

In PrettyTime, there are no classes that meet the requirements to be inlined.

5.3.3 Summary

In five of nine cases, the *Inline Class* refactoring could statistically significantly improve the benchmark results. On the interpreted–mode HotSpot 6 VM, the refactoring improved the results of all tested benchmarks. On this VM, the benefit due to class inlining is 0.9 to 2.5 percent. Accordingly, we observed also a positive effect of the refactoring for the Melbourne and Bahrain benchmark on the client–mode HotSpot 6 VM. In contrast, all benchmarks on the JRockit 6 VM and the Eishay benchmark on the HotSpot 6 VMCM showed no statistically significant change after applying *Inline Class*.

On the HotSpot 6 VMs, a more aggressive inlining heuristic could possibly gain the performance benefit. For instance, we could inline all classes that are used by at most six other classes. However, a class that is inlined into several classes can increase the memory footprint (see Section 4.3 on page 63). Moreover, large classes can cause cache misses [TS05]. Class inlining on spec can thus even have a negative effect on performance.

Based on the benchmark results, we cannot draw a clear conclusion. Although the claim that class inlining improves the performance of a Java program is true for an interpreted–mode HotSpot 6 VM, it is wrong for a JRockit 6 VM. Consequently, we have to accept the null hypothesis of Thesis B. That is, the general claim that *Inline Class* improves the performance of a Java program is not true.

5.4 Replace Inheritance with Composition

The application of the *Replace Inheritance with Composition* refactoring involves many obstacles. For instance, **protected** members of the former superclass (now the *delegatee*) become invisible if the refactored class (the *delegator*) is not in the same package like the former superclass. Also, a class may break if it uses synchronization methods such as `wait()` and `notify()` in both the *delegator* and the *delegatee*. Consequently, [KS08] shows that the refactoring *Replace Inheritance with Composition* is only applicable to 26 percent of all subclasses. Moreover, in 37 percent of all possible applications the refactoring exposes so much implementation overhead that the benefit of the refactoring must be questioned.

Our benchmarks in Section 3.3 on page 37 show that the refactorings is only suitable for classes with an inheritance depth greater than four. But even in HyperSQL with more than 200,000 lines of code, there is only one class that meets this requirement (namely `org.hsqldb.ParserDDL`). This fact shows that the approach of improving the object creation and method invocation speed by *Replace Inheritance with Composition* lacks practical relevance.

Chapter 6

Related Work

Generally, the influence of refactoring on software quality characteristics is a topic of numerous scientific publications. For example, in [Als09], Alshayeb analyzes the influence of refactoring on adaptability, maintainability, understandability, reusability, and testability. His research shows that the claim that refactoring generally improves these software quality characteristics is not true. Accordingly, Du Bois demonstrates, with the help of a formal analysis of the effect of refactorings on coupling and cohesion, that refactoring can both improve and impair software design [DB06]. [WKK07] shows that refactoring does not necessarily cause an increased maintainability and modifiability of a program. By contrast, the results even indicate that more effort can be required when new functions are added to the refactored program.

6.1 Performance

In [Dem02], Demeyer analyzes the influence of the *Replace Conditional with Polymorphism* refactoring. He shows that the performance gets better after applying the refactoring and reasons this improvement with the efficient way in which current compilers optimizes polymorphic methods. In [GR04] refactoring is used to refactor legacy protocol implementations in C++. The refactoring speeds up the execution time in six of eight cases and on an average of 12 percent.

[SKAP10] suggests a refactoring tool to meet the non–functional requirements of the user. Using the tool SPL Conqueror[1], the user should determine his requirements in terms of performance, memory footprint, and so on. Next, the planned tool searches—if provided, with a user–defined metric—for refactoring candidates. Subsequently, SPL Conqueror automatically applies refactorings to meet the user's requirements. Consequently, the tool must balance the use of refactorings with conflicting attributes. In a proof of concept, the authors show that their tool can already improve the runtime of a program by applying the *Inline Method* refactoring. Unlike the *Jefactor*, the tool works on source code level. Therefore, it could also be used to refactor source–code related properties such as coding style.

[TD99] presents a tool that performs source–to–source transformations. The authors concentrate on removing the overheads resulting from dynamic dispatch in common de-

[1] www.fosd.de/SPLConqueror

sign patterns such as *Visitor*. The approach uses logic–based transformation and declaration annotation languages to parameterize a code generator that transforms higher-level design patterns into more efficient representations. That is, their tool applies composite refactorings (i.e., refactorings that are composed of several refactorings) and works on a higher level than *Jefactor*.

A tool that is very similar to *Jefactor* is *BARBER* presented in [TS05]. The tool works also on Java bytecode level and can perform five refactorings. In the example benchmarks, the refactorings of *BARBER* cause a performance benefit of up to 293 percent. Unfortunately, the tool is no longer under development.

Other publications about refactoring and performance suggest the refactoring of sequential applications to concurrency. For instance, [DTR+09] introduces a tool called *ReLooper*[2]. The tool leverages the `ParallelArray` data structure to parallelize loops. It speeds up the execution of the tested programs by 1.32 to 1.96 times on dual–core machine. However, the authors also show that the parallelization of the loops causes a slight performance penalty (about 2 percent) on a single–core machine. [DME09] demonstrates that the refactoring of sequential code to concurrency is error–prone and tedious. Consequently, the authors developed a tool based on the Eclipse IDE called *Concurrencer*[3]. It can automatically perform three refactorings: converting **int** fields to `AtomicInteger`, converting `HashMap` to `ConcurrentHashMap`, and parallelizing divide–and–conquer algorithms. Especially, the speedup of the parallelized divide–and–conquer algorithms is promising. After refactoring, they run 1.18 to 1.94 times faster on a two core machine and 1.6 to 3.82 times faster on a quad core machine. Furthermore, the authors suggest that we should use the classes of the `java.util.concurrent` package to minimize the risk of introducing an error and leverage the performance of its non–blocking algorithms. Similarly, in [VTD06], Vaziri and colleagues present a *data–centric* approach to making a Java class thread-safe. They make use of annotations that denote atomic sets (i.e., sets of class fields that should be updated atomically) and units-of-work (i.e., methods operating on atomic sets that should executed without interleaving from other threads). Their system automatically generates one lock for each atomic set and uses the lock to protect field accesses in the corresponding units-of-work. Accordingly, Javar, a compiler–based, source–to–source tool make use of annotations to parallelize loops and recursive algorithms [BVG97]. Javar rewrites the annotated code to run in parallel using multiple threads.

[SDS+10] investigates the refactoring of already concurrent applications. The paper demonstrates that basic refactorings can break concurrent programs, even if they work correctly for sequential code. However, these problems can be solved in most cases by usage of concurrency–related language constructs such as the **synchronized** keyword in Java.

Another approach to improve the performance by refactoring is described in [BD09]. The authors suggest that refactoring should be used to improve the data locality. This approach is based on the observation that in many applications, cache misses cause the CPU to stall during more than half of the execution time. Consequently, the main bottleneck in applications is not the computation time, but the time to access the memory.

[2] https://netfiles.uiuc.edu/dig/RefactoringInfo/tools/ReLooper/
[3] https://netfiles.uiuc.edu/dig/RefactoringInfo/tools/Concurrencer/

To avoid this bottleneck, they developed an analyzer[4] that detects candidates that possibly cause cache misses and suggests how to prevent them. Due to reducing the stall time of the CPU, the refactoring for data locality speeds up the execution time of several SPEC200 benchmark programs from 1.09 to 4.11. Because of obstacles such as long reuse paths, dynamic data structures, and complex control flow, the analyzer needs a human programmer to verify whether a refactoring can be applied or not. Accordingly, [Pep08] shows that improving the data locality of three nested loops speeds up their execution time by 3.14 times. The positive effect of refactoring on data locality is also presented in [TS05].

Another topic in research that could cause large performance gains is *code perforation* [MSHR10]. *Code perforation* trades off accuracy in return for increased performance of an application. The benchmarks in [ARS+09] show that code perforation can speed up the execution time by two to three times while distorting the output by less than 10 percent. However, it can be difficult for a fully automatic tool to determine whether the code perforation is valid. Therefore, a combination of code perforation and refactoring is promising. A possible scenario is that an analyzer or IDE suggests the developer candidates for code perforation and she decides whether to perforate the regarding code (i.e., apply the refactoring) or not. Thereby, an obstacle is that a refactoring must not change the behavior of an application (see Section 2.1.1 on page 5). However, if an application does not need the provided accuracy, the code perforation does not change the behavior. For example, if an application displays only an rounded off number, the computation of the decimal places of the number is needless.

6.2 Memory Footprint

[AR03] shows how the maximum heap memory consumption of a Java program can be reduced by up to 40 percent. For that purpose, the authors detect unused, constant, or overly-wide fields, and optimize (i.e., delete, change their modifiers, or merge) them. Furthermore, they apply transformations to eliminate fields with common default values or usage patterns.

Shaham and colleagues [SKS01] present a heap-profiling tool for exploring the potential for space savings in Java applications. The output of the tool is used to direct the application source code refactoring in a way that allows more timely garbage collection of objects. By (manually) refactoring the source code of the benchmarks, they report a mean decrease in memory footprint by 18 percent for the applications in the SPECJVM98 benchmark suite. The automatic refactoring of the code (based on profiling data) could be also a use case for the Jefactor tool.

[PHK03] uses a combination of Java reflection, dynamic class loading, code shrinking, code obfuscation, and aspect-oriented programming to decrease the memory footprint of a middleware implementation in Java. The authors reveal that a reduction in memory footprint by more than 95 percent can be achieved. Moreover, the approach needs only a faint intervention by the developer.

Clausen and colleagues [CSCM00] present a tool that uses refactoring on bytecode–level to reduce the memory footprint of programs for low–end embedded systems. The

[4] http://slo.sourceforge.net

authors reveal that their tool can reduce the memory footprint by on average 15 percent. However, the tool has two main disadvantages. First, it introduces a performance penalty of up to 30 percent. Second, the refactored programs cannot be executed by a standard JVM. Unlike Jefactor, the tool does not apply refactorings of Fowler's list [FB99]. Instead, it uses factorization of common instruction sequences to reduce the memory footprint.

[AGGZ07] presents a heap space analysis for Java bytecode. The analysis can—based on the bytecode instructions of a program—estimate the heap consumption of it. Unlike the computation of the heap space computation presented in this thesis, the work takes into account the garbage collection performed by the Java virtual machine.

Hofmann and Jost [HJ06] present a type system for heap analysis at compile–time. Their work uses amortized analysis to assure that the heap memory consumption of a given source code do not excess an upper bound. The approach depends on explicit deallocation of objects at source code level.

Chapter 7

Conclusion

We presented the influence of the refactorings *Replace Inheritance with Delegation*, *Inline Method*, and *Inline Class* on performance and memory footprint. With the help of micro—benchmarking, we showed that these refactorings can improve the performance of a program. We revealed that *Replace Inheritance with Delegation* can speed up the method invocation and object allocation time for classes with an inheritance depth greater than four on HotSpot VMs. The *Inline Class* refactoring also decreases the allocation time of new objects. Additionally, we showed that *Inline Method* is beneficial for improving the method invocation time. However, the reached performance benefits due to the refactorings are only slight. The reason for these small performance gains is the effectiveness of current JVM implementations. They can allocate new objects and invoke methods very fast. Moreover, we showed that the JIT compilers of the HotSpot VM 5 and 6 can significantly optimize the execution time of the code at runtime. That way, they nearly nullify the influence of the analyzed refactorings.

Furthermore, we presented the effect of the refactorings on memory footprint. *Replace Inheritance with Delegation* usually causes an increase in memory footprint. However, it depends on the runtime behavior of the program. If only a small number of class instances exists concurrently, the refactoring may be beneficial in terms of memory consumption. We also observed that *Inline Method* reduces the memory footprint if the method to inline is used by a single method. Moreover, it can be beneficial to inline a method into several other methods if the method's body has a smaller memory footprint than the components (e.g., the *method torso* and the method invocation instructions) that become obsolete by the inlining. Similarly, the inlining of a class decreases the memory footprint if the class to inline is used by a single class or the components (for example, the *class torso* and reference fields) that become obsolete by the inlining have a larger memory footprint than the body of the class to inline. For all presented refactorings, we derived formulas to estimate their memory consumption on an interpreted–mode HotSpot 6 VM. In this vein, it is possible to give an outlook of the effect of a refactoring on memory footprint without measuring.

In the evaluation, we analyzed the effect of the refactorings on performance on a larger scale. For that purpose, we developed two tools. First, the Janalyzer that can identify candidates for refactoring. Second, the Jefactor that applies the refactorings suggested by the Janalyzer on byte–code level. According to the results of the micro–benchmarks, our evaluation showed that the refactorings *Inline Method* and *Inline Class*

are only expedient for the performance of the test programs on an interpreted–mode HotSpot VM, which is a virtual machine without a JIT compiler. In addition, the evaluation demonstrated that the use of *Replace Inheritance with Delegation* to improve the performance is disputable. In hand–written programs, inheritance depths greater than four are rare.

The thesis showed that the effect of a refactoring depends on several conditions. Among others, the used JVM, the given hardware resources, and the code base determine whether a refactoring is beneficial or not. As a consequence, general claims regarding refactoring are not tenable. We have to negate both questions from the beginning of this thesis for the general case:

- Is it beneficial to use refactoring to improve the performance of a software program?
- Is it meaningful to use refactoring to reduce the memory footprint of a software program?

Nevertheless, the thesis also showed that refactoring *can* be useful to improve the performance and to reduce the memory footprint under certain conditions. Accordingly, the presented related work shows that refactoring can cause significant performance benefits and memory footprint reductions. The related work additionally indicates that particularly refactorings that are not included in Fowler's refactoring list, such as refactorings to concurrency and refactorings for data locality, improve the performance.

7.1 Future Work

The analysis of three refactorings only gives an impression of the influence of refactoring on performance–related properties. Therefore, we must cope with several tasks in the future.

First, the influence of the refactorings on automatically generated programs should be examined. In our evaluation, we only analyzed the influence of the refactorings on hand–written and already optimized applications. Consequently, the effect of the refactorings could be more significant for automatically generated programs.

Second, the influence of further refactorings must be examined. This analysis should not only refer to performance and memory footprint, but also include other non–functional properties such as energy consumption.

Third, as we did in the case of memory footprint, future work should focus on the derivation of formulas to estimate the influence of a refactoring for all other non–functional properties. That way, we could use Jefactor to generate tailored software. For instance, the user provides his requirements in terms of energy consumption and memory consumption. Subsequently, Jefactor computes the refactorings that are necessary to meet these requirements and applies the appropriate refactorings.

Fourth, the further development of Jefactor should generally be a main task for the future. Especially, the support of more refactorings is desirable. In addition, [TD99] indicates that refactoring is especially beneficial for performance if it applies transformations that a compiler cannot do. For example, the refactoring of sequential code

to concurrency could significantly increase the performance [DME09][DTR+09]. Consequently, Jefactor could suggest the developer refactoring candidates for those cases where it cannot validate whether a refactoring is correct or not. Such cases occur when the refactoring tool or the compiler has no sufficient information about the intentions of the developer. Due to the information provided by the developer, refactorings could be applied that would otherwise not be possible.

Bibliography

[AFSS00] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, DYNAMO '00, pages 52–64, New York, NY, USA, 2000. ACM.

[AGGZ07] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for java bytecode. In *Proceedings of the 6th international symposium on Memory management*, ISMM '07, pages 105–116, New York, NY, USA, 2007. ACM.

[ALR10] Bill Burke Andrew Lee Rubinger. *Enterprise JavaBeans 3.1*. O'Reilly, 6th edition, September 2010.

[Als09] Mohammad Alshayeb. Empirical investigation of refactoring effect on software quality. *Information & Software Technology*, 51(9):1319–1326, 2009.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison—Wesley, 2nd edition, August 2006.

[AR03] C. Scott Ananian and Martin Rinard. Data size optimizations for java programs. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 59–68, New York, NY, USA, 2003. ACM.

[ARS+09] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, Massachusetts Institute of Technology, 2009.

[Bat06] Don Batory. A tutorial on feature oriented programming and the AHEAD tool suite. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 3–35. Springer, 2006.

[BD09] Kristof Beyls and Erik H. D'Hollander. Refactoring for data locality. *IEEE Computer*, 42(2):62–71, 2009.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[Ben99] Jon Bentley. Algorithm alley: Code tuning in context. *Dr. Dobb's Journal*, 24(5):125–126, 128, May 1999.

[BLC02] Éric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, pages 181–193, Novemeber 2002.

[Blo08] Joshua Bloch. *Effective Java*. The Java Series. Addison—Wesley, Upper Saddle River, NJ, USA, 2nd edition, 2008.

[Bor99] Jürgen Bortz. *Statistik für Human- und Sozialwissenschaftler*. Springer Verlag, Berlin, Germany, 5th edition, 1999.

[BVG97] Aart J. C. Bik, Juan E. Villacis, and Dennis Gannon. javar: A prototype java restructuring compiler. *Concurrency—Practice and Experience*, 9(11):1181–1191, 1997.

[BW08] Sara Boslaugh and Paul Andrew Watters. *Statistics in a Nutshell*. O'Reilly, 2008.

[BW09] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison—Wesley, 4th edition, May 2009.

[CL01] Richard Cardone and Calvin Lin. Comparing frameworks and layered refinement. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 285–294, Washington, DC, USA, 2001. IEEE Computer Society.

[Coo00] James W. Cooper. *Java Design Patterns: A Tutorial*. Addison—Wesley, Reading, MA, USA, 2000.

[CSCM00] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22:471–489, May 2000.

[DB06] Bart Du Bois. *A Study of Quality Improvements by refactoring*. PhD thesis, University of Antwerp, September 2006.

[DC94] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 273–282, 1994.

[Dem02] Serge Demeyer. Maintainability versus performance: What's the effect of introducing polymorphism? Technical report, University of Antwerp, 2002.

[DME09] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, Washington, DC, USA, 2009. IEEE Computer Society.

[DTR+09] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. Relooper: refactoring for loop parallelism in java. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 793–794, New York, NY, USA, 2009. ACM.

[FB99] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. The Addison-Wesley object technology series. Addison—Wesley, Reading, MA, USA, 1999.

[Fla02] David Flanagan. *Java in a Nutshell. A Quick Desktop Reference*. O'Reilly, 3rd edition, 2002.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 57–76, New York, NY, USA, 2007. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison—Wesley, 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison—Wesley Longman, Amsterdam, The Netherlands, 3rd edition, June 2005.

[Goe04] Brian Goetz. Java theory and practice: Dynamic compilation and performance measurement. `http://www.ibm.com/developerworks/java/library/j-jtp12214/`, 2004.

[Goe05] Brian Goetz. Java theory and practice: Anatomy of a flawed microbenchmark. `http://www.ibm.com/developerworks/java/library/j-jtp02225.html`, 2005.

[Goo10] Google. Designing for performance. `http://developer.android.com/guide/practices/design/performance.html`, 2010.

[GP09] Sebastian Götz and Mario Pukall. On performance of delegation in java. In *HotSWUp '09: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, pages 1–6, New York, NY, USA, 2009. ACM.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison—Wesley, Upper Saddle River, NJ, USA, 2006.

[GR04] Birgit Geppert and Frank Rößler. Effects of refactoring legacy protocol implementations: A case study. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium*, pages 14–25, Washington, DC, USA, 2004. IEEE Computer Society.

[Hal09] Stuart Halloway. *Programming Clojure (Pragmatic Programmers)*. Pragmatic Bookshelf, 2009.

[HC08] Cay S. Horstmann and Gary Cornell. *Core Java: Volume I—Fundamentals*. Sun Microsystems Press, Upper Saddle River, NJ, USA, 8th edition, 2008.

[Hei07] Janice J. Heiss. Writing better code. `http://java.sun.com/developer/technicalArticles/Interviews/goetz_qa.html`, March 2007.

[HF01] Jim Highsmith and Martin Fowler. The agile manifesto. *Software Development Magazine*, 9(8):29–30, 2001.

[HJ06] Martin Hofmann and Steffen Jost. Type–based amortised heap-space analysis (for an object–oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[HW10] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Boca Raton, FL, USA, 2010.

[Kab06] Heinz Kabutz. How deep is your hierarchy? `http://www.javaspecialists.eu/archive/Issue121.html`, February 2006.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison—Wesley, Boston, MA, USA, 2004.

[Knu74] Donald E. Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 6:261–301, December 1974.

[KP08] Mehdi Khosrow-Pour. *Encyclopedia of Information Science and Technology*. Information Science Reference, Hershey, PA, USA, 2nd edition, September 2008.

[KS08] Hannes Kegel and Friedrich Steimann. Systematically refactoring inheritance to delegation in java. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 431–440, New York, NY, USA, 2008. ACM.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object–oriented systems. In *OOPLSA '86: Conference Proceedings on Object–Oriented Programming Systems, Languages and Applications*, pages 214–223, New York, NY, USA, 1986. ACM.

[LM06] Michele Lanza and Radu Marinescu. *Object–Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object–Oriented Systems*. Springer Verlag, Berlin, Germany, 2006.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison—Wesley, 2nd edition, 1999.

[McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2004.

[Mey05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison—Wesley Professional, 3rd edition, 2005.

[Mik04] Vitaly Mikeev. Improving swing performance: Jit vs aot compilation. `http://linux.sys-con.com/node/46901`, November 2004.

[MSHR10] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pep08] Kirk Pepperdine. Java performance puzzlers. `http://www.viddler.com/explore/oredev/videos/19/`, 2008.

[PHK03] Mark Panahi, Trevor Harmon, and Raymond Klefstad. Adaptive techniques for minimizing middleware memory footprint for distributed, real-time, embedded systems. In *Proceedings of the IEEE 18th Annual Workshop on Computer Communications*, volume 18, pages 54–58, 2003.

[Pip02] Jens Uwe Pipka. Refactoring in a "test first"—world. In *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 178–181, 2002. Alghero, Sardinia, Italy.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of refactoring. In Kenneth Baclavski and Haim Kilov, editors, *Proceedings of the 10th OOPSLA Workshop on Behavioral Semantics*, pages 18–31. Northeastern University, October 2001. Tampa Bay, Florida, USA.

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Addison—Wesley Professional, September 2003.

[Rob99] Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[Ruo98] Mark Ruolo. Accelerate your java apps! `http://www.javaworld.com/javaworld/jw-09-1998/jw-09-speed.html`, September 1998.

[SDS+10] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct refactoring of concurrent java code. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 225–249. Springer, 2010.

[SFP+08] Roy Singham, Martin Fowler, Rebecca Parsons, Neal Ford, Jeff Bay, Michael Robinson, Tiffany Lentz, Stelios Pantazopoulos, Ian Robinson, Erik Doernenburg, Julian Simpson, Dave Farley, Kristan Vingrys, and James Bull. *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*. Pragmatic Programmers. Pragmatic Bookshelf, March 2008.

[SGCM05] A. K. Samantaray, S. K. Ghoshal, S. Chakraborty, and A. Mukherjee. Improvements to single-fault isolation using estimated parameters. *Simulation*, 81:827–845, December 2005.

[Shi03] Jack Shirazi. *Java Performance Tuning*. O'Reilly, Beijing, China, 2nd edition, 2003.

[SKAP10] Norbert Siegmund, Martin Kuhlemann, Sven Apel, and Mario Pukall. Optimizing non-functional properties of software product lines by means of refactorings. In *Proceedings of Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 115–122, January 2010.

[SKS01] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient java. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 104–113, New York, NY, USA, 2001. ACM.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA, 1986. ACM.

[Sun04] Sun Microsystems. Java platform: Standard edition 5 api specification. `http://download.oracle.com/javase/1.5.0/docs/api/`, 2004.

[Sun05a] Sun Microsystems. The java hotspot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html`, 2005.

[Sun05b] Sun Microsystems. Java tuning white paper. `http://java.sun.com/performance/reference/whitepapers/tuning.html`, 2005.

[TD99] Tom Tourwé and Wolfgang De Meuter. Optimizing object-oriented languages through architectural transformations. In *Proceedings of the 8th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 244–258. Springer, March 1999. Amsterdam, The Netherlands.

[TS05] Eli Tilevich and Yannis Smaragdakis. Binary refactoring: improving code behind the scenes. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 264–273, New York, NY, USA, 2005. ACM.

[vDM02] Arie van Deursen and Leon Moonen. The video store revisited – thoughts on refactoring and testing. In *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76, 2002. Alghero, Sardinia, Italy.

[VTD06] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM.

[WKK07] Dirk Wilking, Umar Farooq Kahn, and Stefan Kowalewski. An Empirical Evaluation of Refactoring. *e-Informatica Software Engineering Journal*, 1(1):27–42, 2007.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Dresden, den 29.01.2011

Stefan Moschinski