

CHRISTIAN-ALBRECHTS-UNIVERSITY KIEL
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Diploma Thesis

Dynamic Analysis of .NET Applications for Architecture-Based Model Extraction and Test Generation

Felix Magedanz (fem@informatik.uni-kiel.de)

October 15, 2011

Advised by: Prof. Dr. Wilhelm Hasselbring
Dipl.-Inform. André van Hoorn

Abstract

Extending the Kieker framework [Software Engineering Group, University of Kiel 2011] to allow comprehensive dynamic analysis of applications developed with Microsoft's .NET framework is the fundamental approach taken by this thesis. Furthermore, the practicality and robustness of the presented extension—which we call *Kieker.NET*—had to be proved not only by controlled experiments in the lab, but also by an in-depth case study on a real-world software system.

We achieved the first goal by developing a .NET-based Kieker framework that does not just simply co-exist with its Java-counterpart, but rather cooperates with it. We employed a .NET/Java bridging technology to make Kieker's core available to .NET, and ported only as few Java classes as possible. Our monitoring probes are based on Postsharp, a powerful AOP implementation for the .NET framework.

With running examples throughout the development stages of our framework, as well as a comprehensive overhead analysis conducted with a micro-benchmark, we showed that Kieker.NET is capable of performing almost all tasks in the .NET programming environment that can be done with Kieker's original monitoring components for Java-based applications.

An extensive case study on a complex real-world software system provided by a major German finance institute, including multiple extracted architecture- and trace-based models and a macro-benchmark, substantiated our findings from the lab studies.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Goals	2
1.3. Document Structure	3
2. Foundations	5
2.1. Dynamic Analysis	5
2.2. Kieker Framework	6
2.3. Microsoft .NET Framework	8
2.4. Bookstore Sample Application	9
2.5. Related Work	10
3. .NET Integration of Kieker	17
3.1. .NET Integration Solutions for Kieker	17
3.2. Bridging Solutions	21
3.3. .NET Integration with JNBridge	24
4. Dynamic Analysis With Kieker.NET	31
4.1. Monitoring Configuration	31
4.2. Manual Instrumentation	33
4.3. Monitoring of the Bookstore Sample Application	36
5. AOP-Based Monitoring With Kieker.NET	39
5.1. Aspect-Oriented Programming in .NET	39
5.2. Kieker.NET Implementation with Postsharp	42
5.3. Aspect Application	48

6. Overhead Evaluation	55
6.1. Causes of Overhead	55
6.2. Experiment Design	57
6.3. Experiment Results	62
7. Case Study	67
7.1. Nordic Analytics	67
7.2. Code Instrumentation	69
7.3. Dynamic Analysis of Nordic Analytics	72
7.4. Architecture-Based Model Extraction	72
7.5. Overhead Analysis	82
8. Conclusion	87
8.1. Summary	87
8.2. Discussion	89
8.3. Future Work	90
A. Kieker.NET Visual Studio Solution	93
A.1. The Bookstore Projects	93
A.2. The Kieker Project	94
B. JNBridge Related Configuration Files and Scripts	109
B.1. Kieker Proxy Generation with JNBProxy	109
B.2. Monitoring Configuration Files	110
Bibliography	113
Acknowledgments	117
Declaration	119

List of Figures

2.1. Overview of the Kieker framework components.	7
2.2. Microsoft .NET framework components.	9
2.3. Structure and dynamics of the Bookstore sample application.	11
2.4. DynaMod work packages.	12
2.5. Basic principles of MDSD.	13
2.6. Architectural domains and modernization drivers.	14
2.7. ADM horseshoe model.	14
3.1. Schematic diagram of a re-implementation of Kieker in .NET.	18
3.2. Schematic diagram of a Kieker client/server architecture.	19
3.3. Schematic diagram of Kieker for .NET employing a bridging solution.	20
3.4. JNBBridge communication features.	23
3.5. Kieker.NET as a new framework component of Kieker.	25
3.6. JNBridge installation directory structure.	26
3.7. GUI version of JNBProxy.	27
4.1. Class diagram of the instrumented Bookstore sample application.	33
4.2. Monitored execution of instrumented Bookstore sample application.	36
4.3. Kieker JVM console output.	37
4.4. Kieker monitoring log.	38
5.1. Postsharp's post-compile-time aspect weaver.	40
5.2. Class diagram of all Kieker.NET classes.	43
5.3. Multithreading problems with bridged ControlFlowRegistry Java class.	47
6.1. Stages I and II of the experiment design.	59

6.2.	Stage III. Quantification of monitoring overhead induced by JNBridge and bridged Java-side Kieker monitoring controller (with Kieker monitoring disabled).	61
6.3.	Stage IV. Quantification of monitoring overhead induced by bridged Java-side Kieker monitoring classes.	62
6.4.	Micro-benchmark overall performance analysis results.	63
7.1.	Nordic Analytics component dependency graph.	70
7.2.	Manually created Nordic Analytics sequence diagram.	75
7.3.	Nordic Analytics assembly component dependency graph (Job 7).	76
7.4.	Nordic Analytics assembly operation dependency graph (Job 7).	77
7.5.	Nordic Analytics aggregated assembly call tree (Job 7).	78
7.6.	Nordic Analytics assembly component dependency graph (Job 29).	79
7.7.	Nordic Analytics assembly operation dependency graph (Job 5).	80
7.8.	Nordic Analytics assembly component dependency graph (Jobs 1 to 29).	81
7.9.	Nordic Analytics macro-benchmark overall performance analysis results.	83
7.10.	Detailed Nordic Analytics performance analysis results.	84
A.1.	Bookstore Visual Studio project structure.	93
A.2.	Visual Studio project structure of the .NET-based Kieker.	94
A.3.	Kieker.NET Visual Studio configuration.	95

Listings

4.1. Instrumented Bookstore class.	35
5.1. Transformed method after aspect application.	41
5.2. Aspect Compile-time initialization method.	44
5.3. Aspect runtime initialization method.	44
5.4. OperationExecutionAspect's OnEntry() advice.	45
5.5. OperationExecutionAspect's OnExit() advice.	45
5.6. Method-level aspect application.	48
5.7. Monitoring log entry for instrumented Catalog.GetBook() method.	49
5.8. Class-level aspect application.	49
5.9. Monitoring log entry for class-level attributed Bookstore sample application.	50
5.10. Assembly attribute for assembly-level aspect application.	50
5.11. Assembly attributes for managed assembly-level aspect application.	51
5.12. Batch file to apply a given aspect to an assembly.	52
5.13. Configuring the aspect targets programmatically.	52
6.1. Performance analysis benchmark (excerpt).	57
6.2. The monitored class containing the single monitored method with parameterized (operation) response time.	58
6.3. The "empty" dummy aspect.	60
7.1. Aspect multicast attributes and filters, leading to 2.836 instrumented methods of Nordic Analytics.	70
A.1. OperationExecutionAspect class.	96
A.2. OperationExecutionAspectProvider class.	101
A.3. MonitoringControllerWrapper class.	102
A.4. ControlFlowRegistry class.	103

B.1. Proxy generation BATCH script.	109
B.2. Kieker proxy class list text file that defines which Java classes to expose.	109
B.3. Batch file to start Kieker JVM.	110
B.4. jnbcore properties file for JNBridge (TCP).	110
B.5. jnbcore properties file for JNBridge (HTTP).	110
B.6. jnbcore properties file for JNBridge (shared memory, only needed for Java/.NET directed communication).	111
B.7. .NET application configuration file for TCP/Binary communication with JNBridge.	111
B.8. .NET application configuration file entry for shared memory communica- tion with JNBridge.	112

List of Tables

3.1. Comparison of bridging solutions.	24
3.2. Exposed Kieker classes.	28
6.1. Stage I experiment results.	63
6.2. Micro-benchmark overall performance analysis results.	63
6.3. Micro-benchmark overhead results.	64
7.1. Code statistics of Nordic Analytics.	68
7.2. Detailed monitored Job statistics.	73
7.3. Nordic Analytics macro-benchmark overall performance analysis results.	83
7.4. Nordic Analytics macro-benchmark overhead results.	83

1. Introduction

1.1. Motivation

In the world of large business applications, software systems often have a lifespan not only of a few years, but a few decades. They become *legacy systems* when they significantly resist modification and evolution [Brodie and Stonebraker 1995]. However, legacy systems are typically the backbone of an organization’s information flow and the main vehicle for consolidating business information [Bisbal et al. 1999]. Given these characteristics, in addition to the fact that the acquisition costs or development expenses— together with the costs that would emerge in case of a transition to another software system—can be extremely high, legacy systems are mission critical [Bennett 1995].

Retaining the maintainability of these “critical assets” [Comella-Dorda et al. 2000] by continuous and sustainable modernization is the approach taken by the DynaMod Project (DynaMod) [van Hoorn et al. 2011], an ongoing joint research project of the b+m Informatik AG, the University of Kiel, and two associated companies, Dataport and HSH Nordbank AG. The focus of DynaMod lays on techniques for Model-Driven Modernization (MDM) of software systems. The innovative aspect of DynaMod is that static and dynamic analysis are used for reverse engineering of architectural and usage models [van Hoorn et al. 2011]. These models, together with semantic augmentation, are the foundation for subsequent generative forward engineering steps and tests.

As one of the envisioned case study systems of DynaMod is a function library developed with the Microsoft .NET framework, this diploma thesis will be conducive to and therefore part of the DynaMod project (for a more detailed introduction to DynaMod see Section 2.5.1).

The goals of this thesis are presented in the next Section 1.2. An overview of the document structure is given in Section 1.3.

1.2. Goals

The context of this work is given by the DynaMod project. As indicated by the title, its main purposes are (1) *architecture-based model extraction* and (2) *test generation* for applications developed with .NET, and therefore running on Microsoft Windows operating systems. The following goals will be substantial for accomplishing (1) and (2) and determine the steps of the approach of this work, which are loosely based on some of the *work packages* defined by DynaMod.

1.2.1. Technical Instrumentation of .NET

The first goal is enabling .NET-based applications to be dynamically analyzed with the *Kieker framework* (Kieker) [van Hoorn et al. 2009]. For more details on Kieker see Section 2.2. As of today, Kieker is currently restricted to Java-based systems only.

Performance—or the absence of overhead caused by the presented solutions—will be a main concern of the tasks corresponding to this goal, as time related analysis, e.g., monitoring of the execution time of a method call, would be drastically affected by overhead due to utilized frameworks, bridging solutions or other ways of framework intercommunication. The same holds for monitoring activities in “live” environments, where delays are often not tolerable.

Another important feature of Kieker is instrumentation by employing techniques based on aspect-oriented programming (AOP) [Kiczales et al. 1997]. This allows monitoring (e.g., measuring the response times of certain method calls) without actually modifying the source code. This feature must be provided for .NET applications as well.

1.2.2. Extraction of Architectural and Usage Models

Developing techniques based on DynaMod and other similar approaches for extracting architecture-based models from .NET-based systems and enriching them with user-behavior data is the second goal of this thesis. Selecting appropriate meta-models and generating usage models with data obtained by dynamic analysis will be the essential tasks. As Kieker already supports different types of models to be generated from data obtained by monitoring, accomplishing the first goal (i.e., instrumentation) also (partially) accomplishes this one.

1.2.3. Test Generation

Developing means for test automation for .NET-based systems is the third goal. The tests should be based on extracted models obtained by the previous steps, and must be compatible with a testing framework to support the required test automation to test even large systems. The generated tests should also be generic and therefore easily adoptable to another system based on the same domain model—which is most likely the same system but modernized with techniques of model-driven modernization in this context.

1.2.4. Evaluation

The final goal is a proof of concept with multiple .NET-based applications, which will indicate the practicality and efficiency of the developed methods and hopefully—presumably after several reiterations—contribute to the success of the “mother project” DynaMod. The main evaluation will be based on one of the case study systems of DynaMod, namely *Nordic Analytics*, the C#-based function library mentioned in the motivation to this document. HSH Nordbank AG uses this library internally for assessment and risk control of finance products.

1.3. Document Structure

The structure of this document is described in the following list.

Chapter 2 briefly introduces the foundations of this work and presents some technologies that are crucial for understanding the following chapters. We also hint to related work.

Chapter 3 focuses on the .NET integration of Kieker. Different approaches are presented, followed by our developed solution.

Chapter 4 shows how the newly integrated Kieker can be used in the .NET framework. With the results of the previous chapter, basic dynamic analysis of .NET applications is already possible, and its core functionality will be illustrated with the help of a running example .

Chapter 5 deals with the employment of techniques based on aspect-oriented programming to allow for a more elegant way of integrating monitoring logic into a system under observation.

Chapter 6 introduces a staged micro-benchmark that will be used for a comprehensive evaluation of the overhead that is caused by our monitoring framework.

Chapter 7 presents a case study that hopefully will show the practicality and robustness of our solutions. The case study system will also be used for a comprehensive macro-benchmark.

Chapter 8 draws the conclusions of this thesis and gives an outlook to possible future work that could be done to further improve dynamic analysis of .NET applications with Kieker.

2. Foundations

This chapter gives an overview of some of the prime methods and technologies that will be used and referenced throughout the thesis. After a short motivation to dynamic analysis in Section 2.1, we introduce the Kieker framework for dynamic analysis in Section 2.2.

Section 2.3 presents the basic concepts of Microsoft's .NET framework, and Section 2.4 introduces a sample application that we use throughout this thesis. Finally, a short overview of related work is given in Section 2.5.

2.1. Dynamic Analysis

Dynamic software analysis is performed by executing the software and analyzing it at runtime, therefore providing information about the runtime behavior of software systems. Advantages of this method compared to static analysis—with the latter being basically source code analysis—are usage profiling as well as the ability to obtain performance measurements. The significance of the findings may depend on the test input the software is executed with.

In their recently published article, Cornelissen et al. [2009] reported on a systematic literature survey with a research body of 4,795 articles that had been published between 1999 and 2008. They systematically selected 176 articles for their comprehensive study. After analyzing and categorizing their selection, their main findings were

1. that a significant technical progress had been made in the past decade by putting great effort into comparing and combining earlier techniques;
2. that the studied literature may overemphasize standard object-oriented systems at the cost of more modern web applications, distributed software systems, and multithreaded systems. They argue that dynamic analysis is especially suitable for those systems;

3. that comparisons and benchmarking should be considered more often as evaluation methods than they currently are. Moreover, conducting controlled experiments is their recommended evaluation procedure.

They also encourage researchers to make their tools publicly available, so that evaluation and verification is possible.

2.2. Kieker Framework

Dynamic analysis in the context of this thesis will be done with the Kieker framework [Software Engineering Group, University of Kiel 2011] by van Hoorn et al. [2009]. Kieker is a monitoring framework that allows dynamic analysis of (Java-based) applications by providing monitoring probes employing AspectJ, Java EE Servlet, Spring, and Apache CXF technology.

Kieker is actively developed and maintained by the Software Engineering Group at the University of Kiel and, according to the description on the project website¹, has already been used for monitoring and profiling in several distributed industry systems of companies in the telecommunications and business sectors. Kieker is licensed under the Apache License, Version 2.0, and located for download at SourceForge².

2.2.1. Kieker Components

Kieker is composed of two main components, `Kieker.Monitoring` and `Kieker.Analysis` (Figure 2.1). By dynamically analyzing an instrumented application, the monitoring component creates monitoring records. These records (bundled in a monitoring log) can later be read by the analysis component. The `Kieker.TraceAnalysis` plugin can be used to extract several different types of models and trace data visualization from the monitoring logs.

2.2.2. Concept

The starting point of all monitoring with Kieker is the concept of *probes*. Monitoring probes are in some way integrated into a system under observance—Kieker provides

¹<http://se.informatik.uni-kiel.de/kieker/>

²<http://sourceforge.net/projects/kieker/files/kieker/>

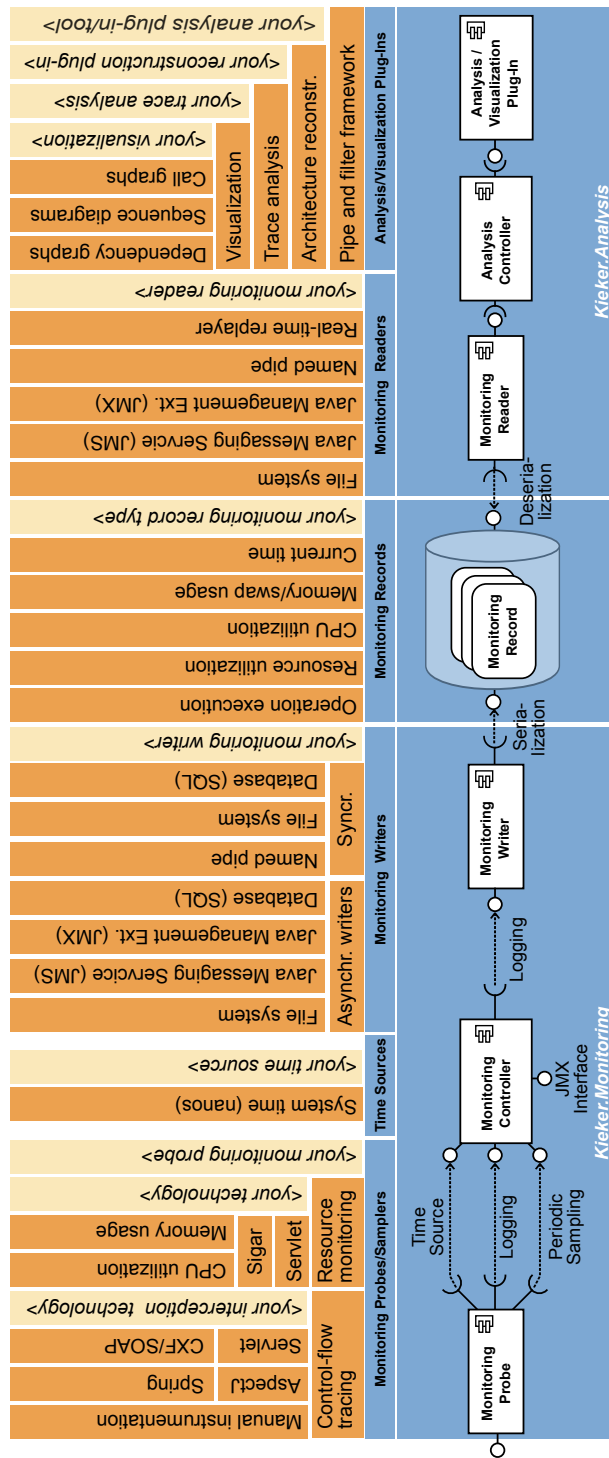


Figure 2.1.: Overview of the Kieker framework components [Software Engineering Group, University of Kiel 2011].

several techniques for that—and collect and possibly preprocess monitoring data when the analyzed application is executed.

The next element in Kieker’s “chain of command” is the monitoring controller. When the probes are triggered, they interact closely with the monitoring controller. For example, the controller provides a *time source* that can be used for temporal measurements by the probes. The data collected by the probes is passed to the controller as *records* (with `OperationExecutionRecord` being a prominent example in this work).

To store the monitoring data, a *monitoring log writer* is employed by the monitoring controller. The elected writer implementation (e.g., filesystem writer, database writer, or writers employing Java Management Extensions (JMX) and Java Messaging Service (JMS)) is responsible for serializing the data as so-called *monitoring logs*. As mentioned before, these logs are the foundation of all further analysis with the Kieker analysis components.

For more details on Kieker and how it can be used for continuous monitoring of software services and comprehensive dynamic analysis, see the Kieker project website, van Hoorn et al. [2009], and the Kieker user guide by Ehmke et al. [2011].

2.3. Microsoft .NET Framework

Microsoft .NET³ is a programming framework that supports a wide range of programming languages. Applications developed with (and for) .NET target primarily computers with Microsoft Windows operating systems, but other implementations exist as well.

One special aspect of .NET is its capability to support language interoperability. A function written in one of the supported languages can be used in any other supported language as well. The most common .NET languages are C++, C#, and VB.NET, but many other compatible languages (for example J#, an implementation of Java) exist.

Figure 2.2 shows the main components of the .NET framework. The core is the *Common Language Infrastructure* (CLI) with its *Common Intermediate Language* (CIL) and its (software-based) runtime environment system *Common Language Runtime* (CLR).

Every application written in a supported language will be compiled into bytecode with the CIL syntax. This bytecode can be executed in the software environment of the CLR.

³<http://msdn.microsoft.com/netframework>

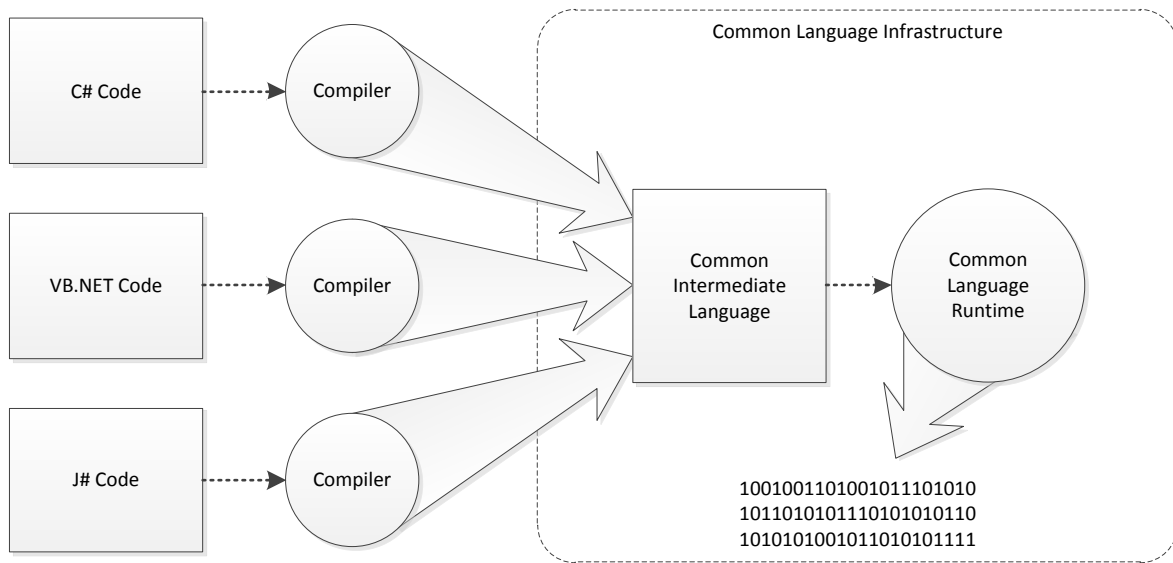


Figure 2.2.: Microsoft .NET framework components.

Microsoft also provides their *Visual Studio*, an *Integrated Development Environment* (IDE) that targets mainly .NET developers, currently available in the 2010 version. Visual Studio is a commercial product, but there is also an *Express* version available for download on the MSDN website for free. The initial release of .NET was in 2002, the current release version is .NET 4.0.

2.4. Bookstore Sample Application

For having a convenient sample application for this work, we ported the Bookstore sample application introduced by Ehmke et al. [2011] to the .NET environment by re-implementing the classes in C#. The class diagram in Figure 2.3(a) shows the classes of the .NET-based Bookstore and lists the members of each class.

Representing a customer facility to search for books (**Bookstore**), combined with a customer relationship management system (**CRM**), the Bookstore sample application provides a basic test setting for the development of our Kieker port to .NET. Most of the methods are just stubs, as there is no need to actually *do* something there for our purposes.

The sequence diagram in Figure 2.3(b) and the operation dependency graph in Figure 2.3(c) illustrate the dynamics of the application. The sequence diagram shows a

typical *trace* (i.e., the order in which a given set of methods calls each other) through the different classes. The operation dependency graph is augmented by the number of calls that are made when executing the sample application. On most of the edges this is the number 10, because the main method of the `BookstoreStarter` class actually starts 10 threads that execute the `Request()` method in parallel. This element has been added to the Bookstore sample application to allow concurrency tests as well.

2.5. Related Work

2.5.1. The DynaMod Project: Dynamic Analysis for Model-Driven Software Modernization

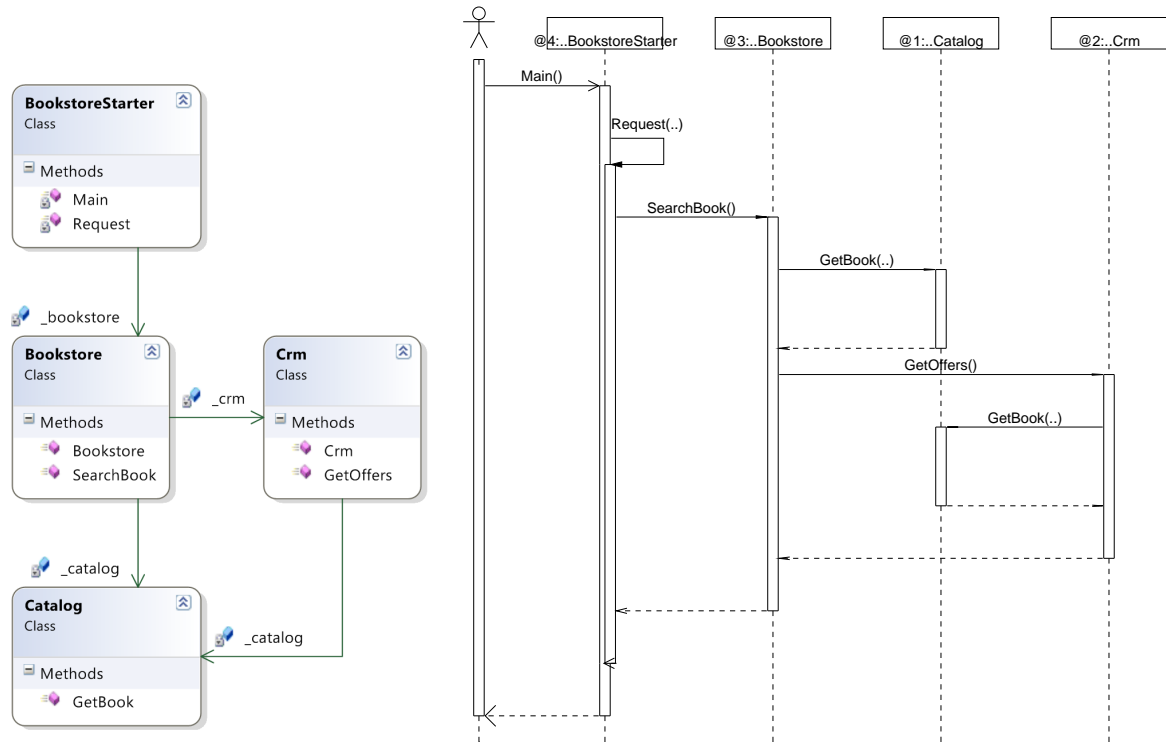
DynaMod⁴ is currently an active research project, which addresses model-driven modernization of software systems. The project started with the beginning of 2011 and has a two-year funding from the German Federal Ministry of Education and Research (BMBF). The Consortium that initiated the project consists of four members: the b+m Informatik AG as development partner and consortium leader, the University of Kiel as scientific partner, and two associated companies, Dataport and HSH Nordbank AG. Dataport and HSH Nordbank AG provide the case study systems that are going to be (partly) modernized and therefore be used to evaluate the techniques developed by DynaMod.

The following key characteristics of the envisioned approach of DynaMod are listed in the paper by van Hoorn et al. [2011]:

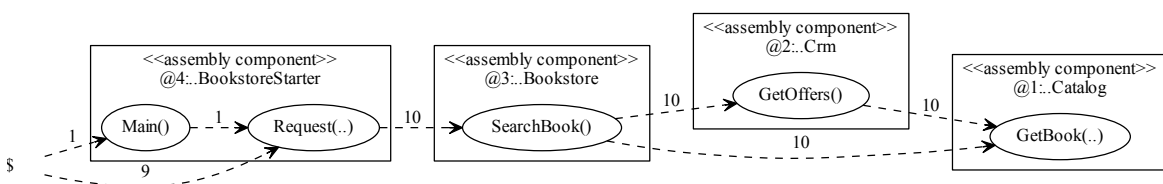
1. Combining static and dynamic analysis for extracting models of a legacy system's architecture and usage profile.
2. Augmenting these models with information that is relevant to the subsequent architecture-based modernization steps.
3. Automatically generating implementation artifacts and test cases based on the information captured in the models.

Figure 2.4 shows a finer grained view on the work packages of DynaMod—aligned with the horseshoe model for re-engineering. Based on static analysis, the first DynaMod work

⁴<http://kosse-sh.de/dynamod/> (in German).



(a) Class diagram of the Bookstore sample application. (b) Sequence diagram of the Bookstore sample application.



(c) Assembly operation dependency graph of the Bookstore sample application.

Figure 2.3.: Structure and dynamics of the Bookstore sample application, following Ehmke et al. [2011].

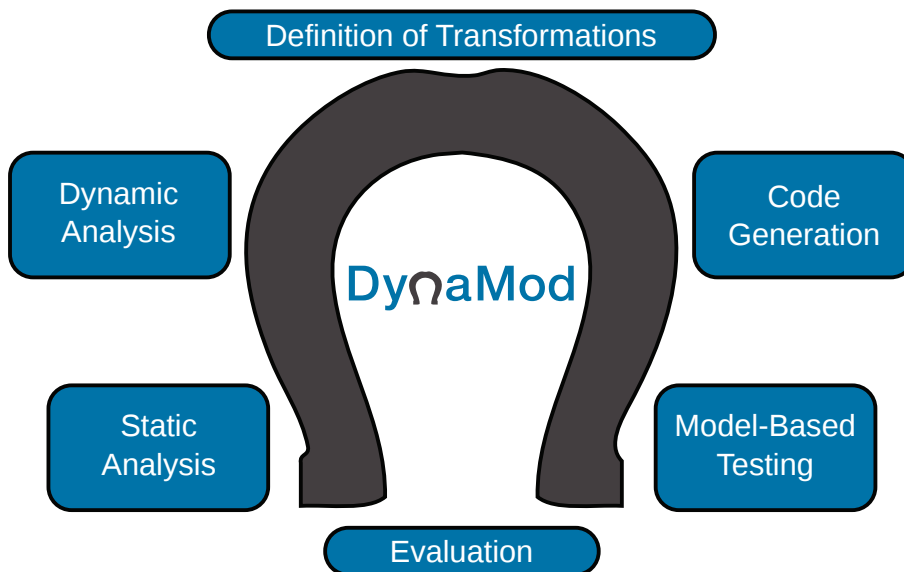


Figure 2.4.: DynaMod work packages [van Hoorn et al. 2011].

package (WP1) focuses on the extraction of architectural models. Appropriate meta-models will be developed by employing and extending standards defined by the Object Management Group for Architecture-Driven Modernization (ADM, see Section 2.5.2). Supported programming platforms will be limited by those emerging from the evaluation case studies provided by Dataport and HSH Nordbank AG.

The legacy system’s internal behavior and external usage profile are the concerns of the dynamic analysis (WP2). For more detail on dynamic analysis and the employed Kieker framework see Section 2.2.

Definition of transformations (WP3) means developing transformation rules for the translation of enriched architecture-level models (i.e., models that describe the existing systems) towards possible target architectures. ADM meta-models may serve for the source models as well as the target models.

Code generation tasks (WP4) employ model-to-code transformation techniques introduced by Stahl and Völter [2006] with the Model Driven Software Development (MDSD) approach (see Figure 2.5 for the basic principles of MDSD). The modeling infrastructure will be based on the Eclipse Modeling Platform (EMP)—which includes the former

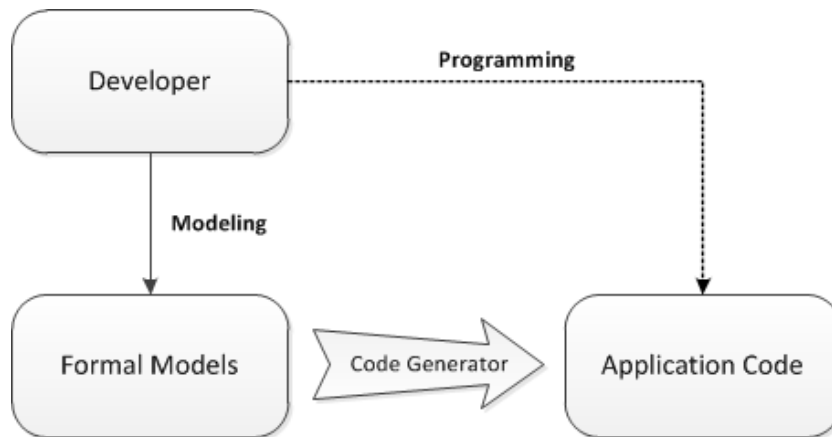


Figure 2.5.: Basic principles of MDS: different methods of code “generation”.

oAW framework that has been co-developed by the DynaMod consortium member b+m Informatik AG.

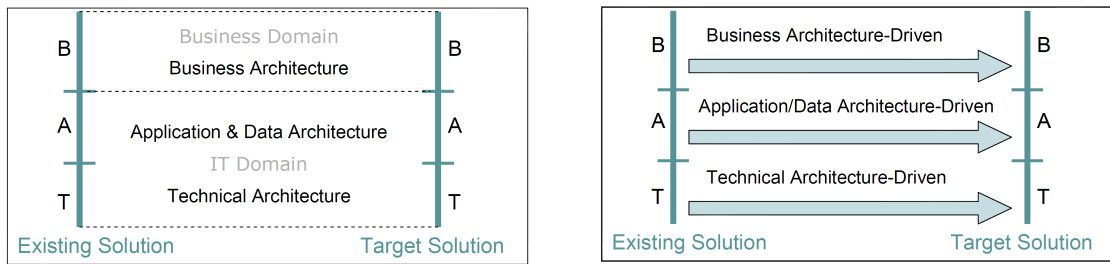
Based on usage models of the legacy system obtained by dynamic analysis, model-based testing (WP5) utilizes appropriate testing tools—most probably the load test tool Apache JMeter—for workload generation and testing of both the modernized and the outdated system. This will allow comparison of quality properties like performance and reliability.

The evaluation (WP6) of the developed methodology and tooling infrastructure will be based on three case study systems from Dataport and HSH Nordbank AG. They are considered to be benchmark examples and representatives of modernization projects pending in practice.

2.5.2. Architecture-Driven Modernization

Architecture-Driven Modernization (ADM) is a collection of standards defined by the Object Management Group (OMG). Two White Papers [Ulrich and Khusidman 2007; Khusidman 2008] are provided at the OMG’s website that present a basic insight into the purposes of ADM.

Basically, ADM is the migration of an existing software system to a target (i.e., modernized) system by applying project-based incremental transformations to the underlying architecture of that system. These projects can be located in the business domain, driven by business related purposes (i.e., changes to business semantics or business processes),



(a) Business & IT architecture domains.

(b) Modernization drivers & trajectories.

Figure 2.6.: Architectural domains (a) and modernization drivers (b) [Ulrich and Khusidman 2007].

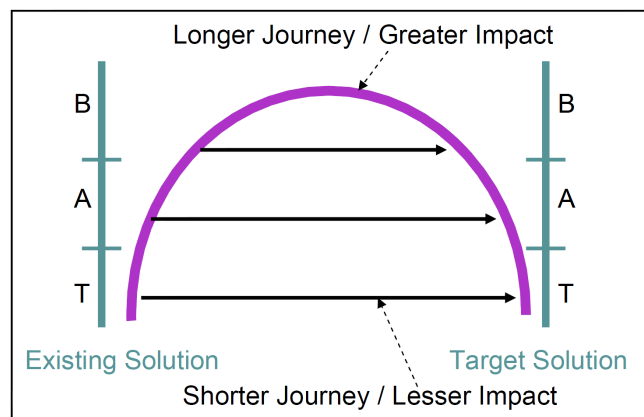


Figure 2.7.: ADM horseshoe model [Ulrich and Khusidman 2007].

as well as in the IT domain, driven by modifications to the technical architectures. (See Figure 2.6).

Historically, modernization projects focused on the transformation of technical architectures. The approach proposed by OMG is referred to as the ADM horseshoe model, as the knowledge curve of the transformational path resembles an upside down horseshoe (see Figure 2.7).

Business architecture-driven modernization is by far the most complex transformation in this model because it incorporates all three architectures. As synchronization of the transformational paths must be both vertically and horizontally (i.e., in the business-to-physical implementation as well as existing-to-target), very little modernization work in the business architecture-driven modernization has been effectively deployed according

to Ulrich and Khusidman [2007]. This is supposedly because the mapping paradigms between the architectures lack standardization. Among others, the Knowledge Discovery Meta-Model (KDM) [OMG 2010] and Software Metrics Meta-Model (SMM) [OMG 2009] are those standards defined by the OMG's KDM Task Force to facilitate complex business architecture-driven modernization.

3. .NET Integration of Kieker

Dynamic analysis in the context of this thesis will be done with the Kieker framework. As Kieker is a Java framework, monitoring is currently restricted to Java-based systems. Enabling .NET-based applications to be dynamically analyzed with Kieker is the main goal of this chapter.

Section 3.1 deals with the assessment of different solutions to the problem of .NET integration of Kieker. In Section 3.2 we present three (commercial) .NET/Java bridging solutions. The one most feasible for our implementation is then presented in detail in Section 3.3.

3.1. .NET Integration Solutions for Kieker

The first step of the technical instrumentation of .NET is the determination of the best way to enable the monitoring logic of Kieker to be called out of the .NET programming environment. Due to the design of the .NET framework, this would allow monitoring of applications written in all .NET programming languages (with C#, C++, and VB.NET being the popular ones). There are several possible solutions, with three being described in more detail in this section. Finally we present our approach based on a short evaluation of the alternatives.

3.1.1. Alternative Approaches

The following paragraphs provide an overview of the diverse possible implementations, ordered by descending assumptive complexity of implementation.

Re-implementing Kieker in .NET

Copying all functionality by re-implementing Kieker in a .NET programming language like C# is one solution to the task of bringing the functional range of Kieker to .NET.

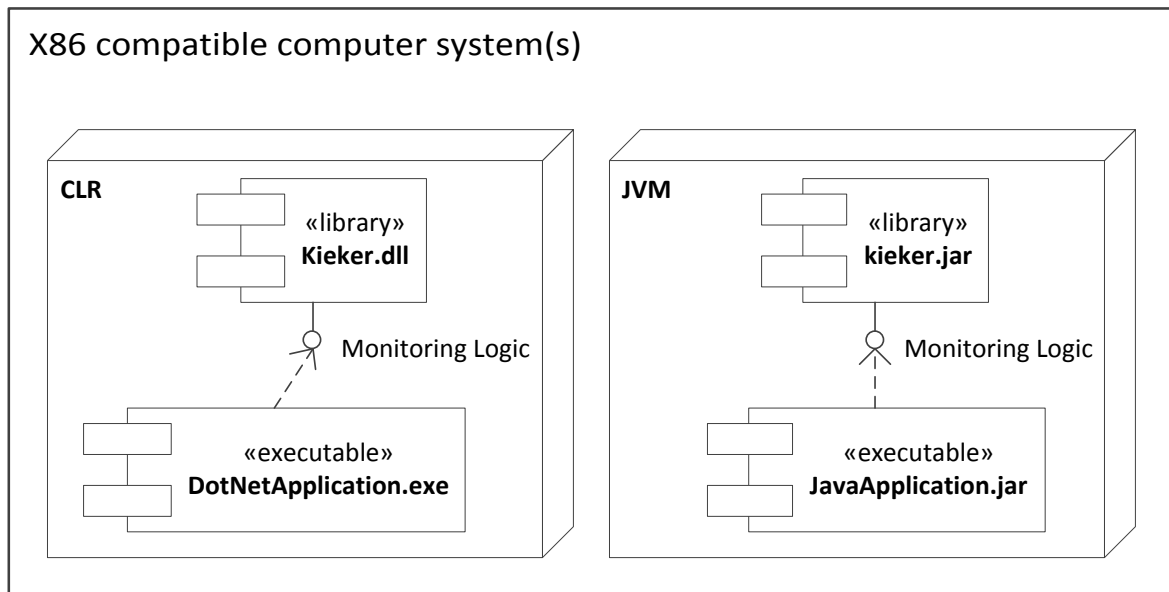


Figure 3.1.: Schematic diagram of a re-implementation of Kieker in .NET.

As C# is syntactically quite similar to Java, this approach would result in a more or less direct copy of Kieker’s classes and interfaces (Figure 3.1). Existing .NET frameworks could be utilized to achieve the same range of application that is provided by the Java platform.

.NET applications execute in the Common Language Runtime (CLR) (see Section 2.3), which is based on the same concept as Java’s Virtual Machine, so the expected runtime characteristics of a Kieker port for .NET could be expected to be much similar to those of its Java archetype.

Building a Client/Server Architecture

Another option is the employment of a communication method like web services for interaction between a Kieker server (Java-based) and a rather lightweight—yet to be developed—Kieker client based on a .NET programming language.

Existing Kieker components could be extended to enable Kieker to act as a server that “listens” to network ports and reacts to messages sent by the client, i.e., providing a web service. Possible messages could be the creation of a collection of monitoring data (called a *record*), a method execution time measurement, etc.

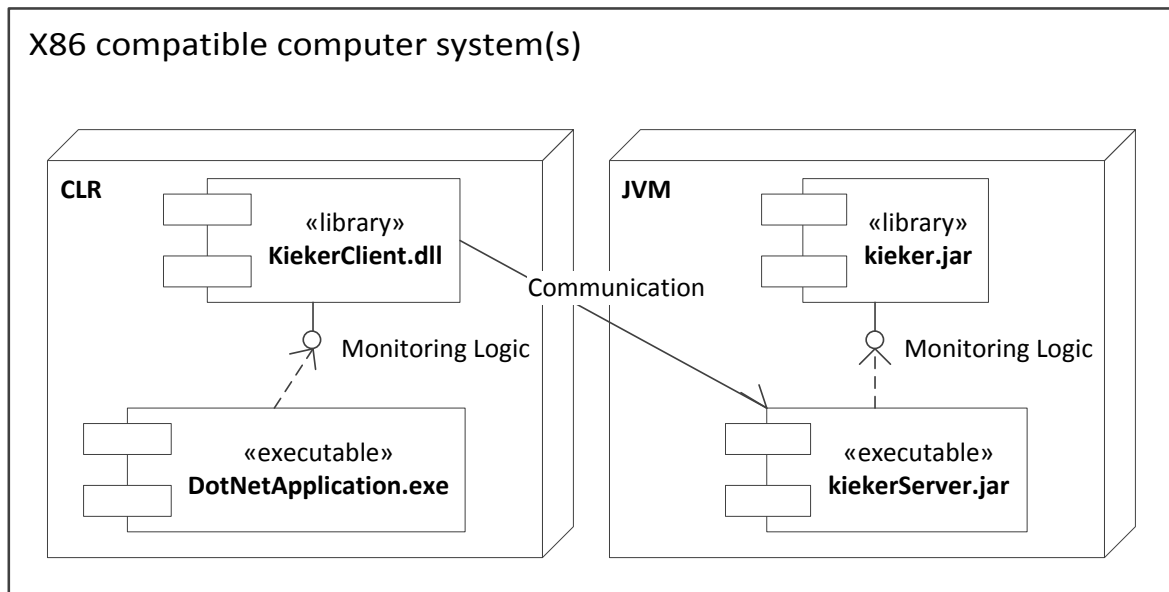


Figure 3.2.: Schematic diagram of a Kieker client/server architecture.

The communication interface could be defined by a machine-readable description of the operations provided by the web service, which is most commonly done with a *Web Service Description Language (WSDL)*.

The client would basically provide an interface that is compatible with all applications written in a programming language compatible with the target platform .NET (see Figure 3.2 for a schematic diagram) and send relevant information to the server via messages defined by the WSDL. In this case, the client would be more than just a Kieker probe. At least all communication with the monitoring controller—which should probably stay on the Java-side—must be handled here as well (see Section 2.2 for the basic principles of Kieker’s components).

Employing a Bridging Solution

Several (mostly commercial) bridging solutions enable interoperability between .NET and Java by providing tools to wrap Java classes into .NET-compatible *Dynamic-Link Libraries (dll)*, which are the *Java-archive (jar)* equivalent of Windows Programming. The other direction (Java/.NET) is also possible in some cases. The wrapped classes, so called *proxies*, can be utilized by the target architecture as if they were native.

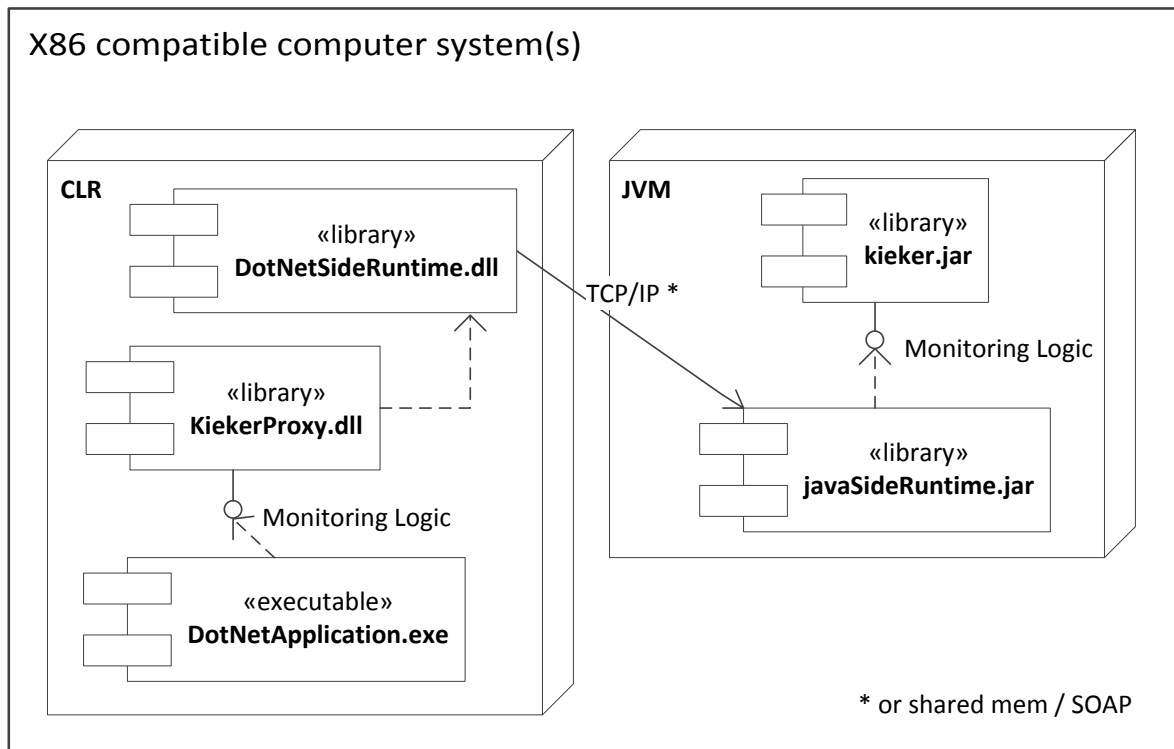


Figure 3.3.: Schematic diagram of Kieker for .NET employing a bridging solution.

Figure 3.3 gives an overview of the involved components when employing a bridge. In the case of our Kieker port, the **KiekerProxy.dll** library would be generated from the Kieker Java classes with the proxy generation tool provided by the bridge. The **DotNetSideRuntime.dll** and **javaSideRuntime.jar** are runtime libraries that ship with the bridge and need to be accessible for the respective execution environments. Thus, all inter-process communication is managed by the bridge and mostly hidden from the user/programmer. Most of the solution’s communication is based on TCP/IP, but some also support shared memory or even messaging based on web service protocols like *Simple Object Access Protocol (SOAP)*.

3.1.2. Evaluation

Re-implementing Kieker for .NET would probably be the “cleanest” solution, as no additional overhead due to—in this case unnecessary—framework inclusion or communication would influence or even falsify monitoring results. As of today, the Java-based

source code of Kieker consists of about 50 Java classes that use different Java technologies and frameworks. Mirroring all this functionality in .NET would be a very time-consuming task. Another severe drawback of this solution would be the redundancy of the two code bases for Kieker (one for the Java platform and one for .NET) with basically the same functionality. This is the reason why we decided against the approach of re-implementing Kieker.

Even though extending Kieker to a client/server architecture looks like an appealing solution, it is questionable whether (1) because of the performance requirements, the “client” to some extent would become a complete re-implementation of Kieker after all (at least of the monitoring part), and (2) considering the third option, the implementation of a .NET-based client, alongside with the modifications to the original Java-based Kieker, would be worth the effort. As pointed out before, all communication would have to be implemented and optimized from scratch to allow efficient client/server interaction. Bridging solutions claim to have all that “out-of-the-box”. One advantage of the development of a Kieker server and the corresponding communication interface would be the re-usability of such a solution.

The employment of a commercial bridging solution seems to be efficient and relatively easy to implement, but its performance impact, i.e., the overhead that comes with bridging technologies, must be evaluated properly. As most of the existing bridges are commercial, license costs must also be taken into consideration. Because of the rather efficient impression of this approach, we decided to choose the employment of a bridging solution for the technical instrumentation of .NET with Kieker. Structurally, the resulting .NET integration of Kieker can also be considered a “Kieker client”, as we implement the probes on the target architecture, and all communication is taken care of by the bridging solution.

3.2. Bridging Solutions

In this section we present three solutions to the .NET/Java interoperability problem. After giving a short overview of the considered solutions in Section 3.2.1, namely *EZ JCom* [Desiderata Software 2011], *J-Integra.NET* (J-Integra) [Intrinsyc Software International, Inc. 2011], and *JNBridgePro* (JNBridge) [JNBridge LLC. 2011a], we discuss our evaluation in Section 3.2.2.

3.2.1. Overview of Commercial Bridging Solutions

In general, .NET/Java bridging solutions can be divided into two classes, distinguished by whether they target the *Component Object Model* COM interface (classic windows programming, which is now largely superseded by .NET) or .NET directly.

EZ JCom

EZ JCom is a bridging solution that allows COM components to be called from Java, and likewise allows Java to be called from COM-aware languages such as the .NET languages. It provides a user interface for proxy generation and also has 64-bit support, but not in the evaluation version. There was no intrinsic support for .NET, but as COM components are accessible in .NET, this would not have been critical.

Aside from its relatively high license costs, the major drawback of EZ JCom was its lack of documentation and its restricted evaluation version that would randomly terminate the established bridge connection when utilized.

J-Integra

With J-Integra.NET and J-Integra COM, J-Integra provides two products that seem to fulfill our requirements. However, we evaluated only J-Integra.NET, as we are interested in calling Java from the .NET environment.

Like the evaluation version of EZ JCOM, the trial license of J-Integra.NET had the limitation to perform a shutdown after two hours of continuous use. That was unacceptable for us, as we expected the monitoring sessions of our case study system to be much longer (which they certainly were).

JNBridge

JNBridge seemed to provide all functionality we needed. It aims directly at .NET/Java intercommunication, can be integrated into a Kieker build process, and ships with a complete reference documentation. The proxy generation tool of JNBridge has a user interface version as well as a command line tool. JNBridge allows different ways of .NET/Java intercommunication. Figure 3.4 gives an overview of all configurable options. For more information about .NET/Java intercommunication with JNBridge see Section 3.3.

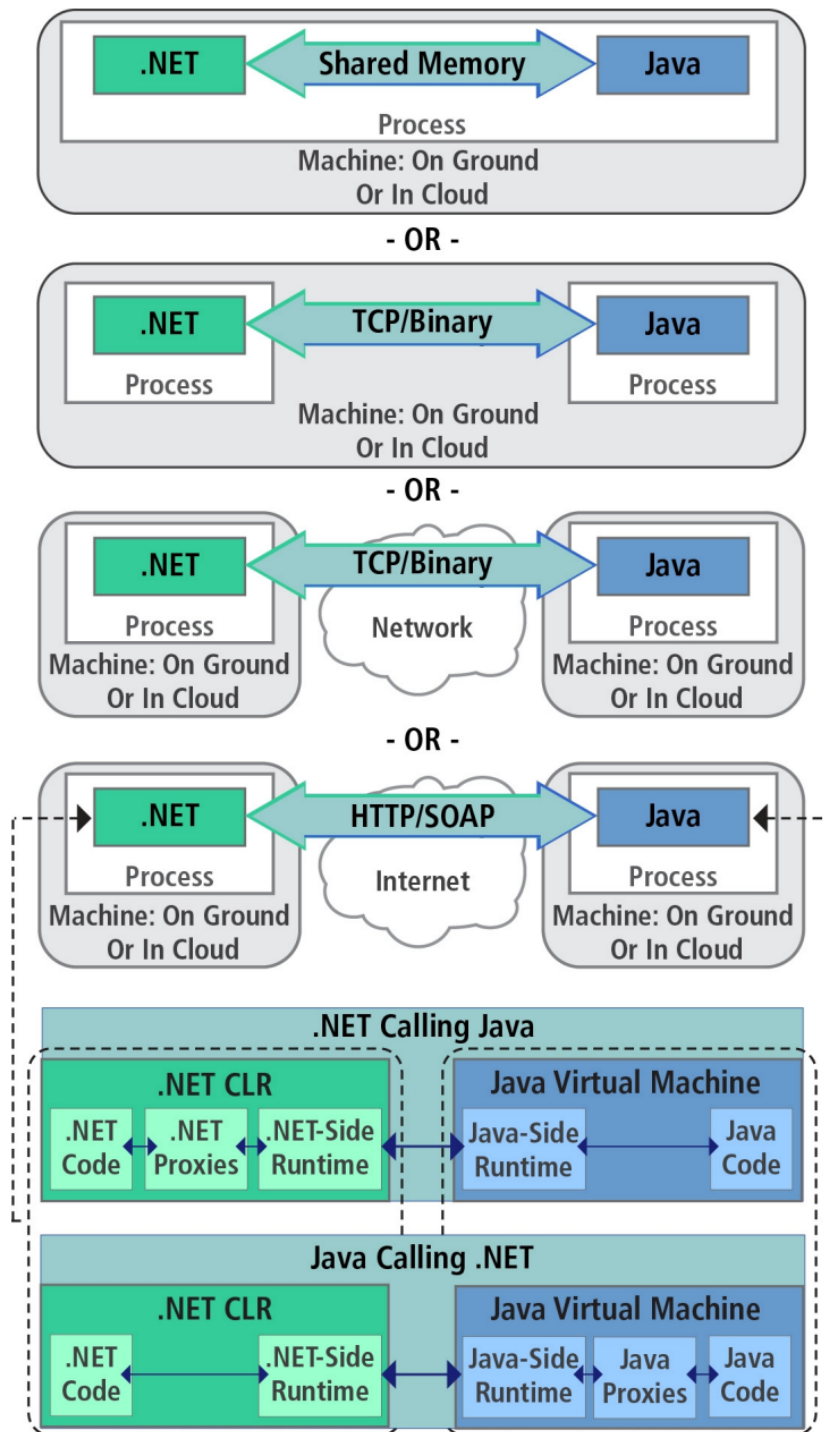


Figure 3.4.: JNBBridge communication features [JNBBridge LLC. 2011a].

CRITERIA	EZ JCOM	J-INTEGRA	JNBRIDGE
Documentation	-	+/-	+
Actuality	-	-	+
Ease of use	+/-	+/-	+
GUI version	+	+	+
Build integration	-	+	+
Shared memory comm.	+	-	+
Unrestricted evaluation	-	-	+
License costs	-	-	-

Table 3.1.: Comparison of bridging solutions.

3.2.2. Evaluation

Table 3.1 clearly identifies JNBridge as our “weapon of choice” for .NET/Java interoperability. After evaluating JNBridge along with the alternatives, the comprehensive and up-to-date documentation, the comparatively long history of regular software updates, and the ability to support allegedly fast shared memory communication lead to the decision to employ JNBridge for .NET/Java intercommunication in the context of our Kieker port.

3.3. .NET Integration with JNBridge

Enabling .NET applications to communicate with a Java system by simply “calling” Java methods can be done with JNBridge in a straightforward manner. As pointed out before, the documentation [JNBridge LLC. 2011b] and demos that come with the distributed installation file give a good overview on how to start, but also have enough depth to allow more advanced application.

We follow the introduction given in the Evaluation and Quick Start Guide [JNBridge LLC. 2011c] to present the necessary steps for installation, configuration, and basic use of JNBridge. The following information should be sufficient for understanding our implementation of Kieker.NET employing JNBridge. In the remainder of this section, all given examples cover .NET → Java directed communication (previously referred to as .NET/Java), i.e., .NET code that calls Java code through the JNBridge communication

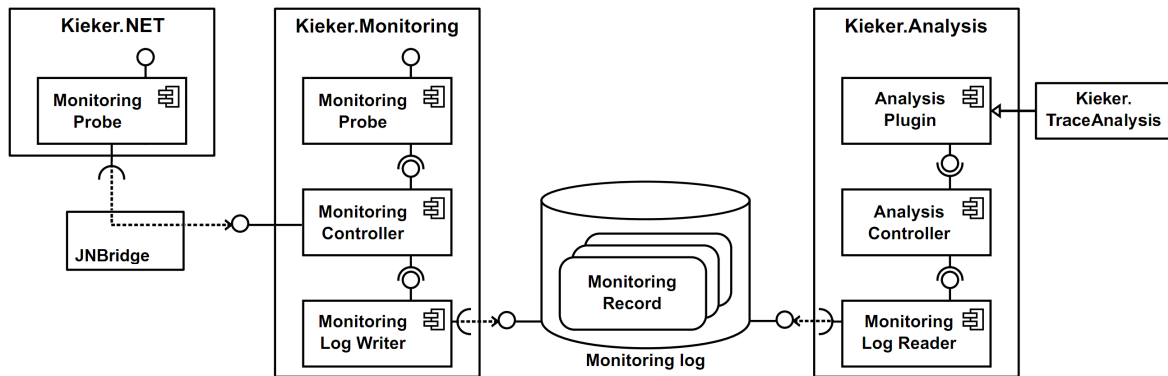


Figure 3.5.: Kieker.NET as a new framework component of Kieker.

mechanism, as this is the only way we employ JNBridge for Kieker.NET. The other direction (Java \rightarrow .NET) is also possible, but is not shown here.

Our targeted result is an extension of the existing Kieker architecture, as shown in Figure 3.5

3.3.1. Download and Installation

To download the latest version of JNBridge, one has to register with a valid email address at the JNBridge web page¹ to obtain a fully functional 30-day trial license and the download link (more detail on JNBridge licensing, especially for open source projects such as Kieker, follows in Section 3.3.2).

There are two different usage scenarios when installing JNBridge on a computer: *proxy generation* and *proxy use*. Hence, as for JNBridge version 6.0 (Sept. 2011), either *Development Configuration* for proxy generation or *Deployment Configuration* for proxy use has to be selected after launching the JNBridge installation file.

Development Configuration is the mode for proxy generation. That means, that the proxy generation tool *JNBProxy* will be installed along with the JNBridge runtime libraries. With this tool, the developer is able to generate the proxy classes that will be used to “call” methods written in the other programming language(s) and compiled to the respective bytecode. We used this tool to generate the Kieker-

¹<http://www.jnbridge.com/downloads.htm>

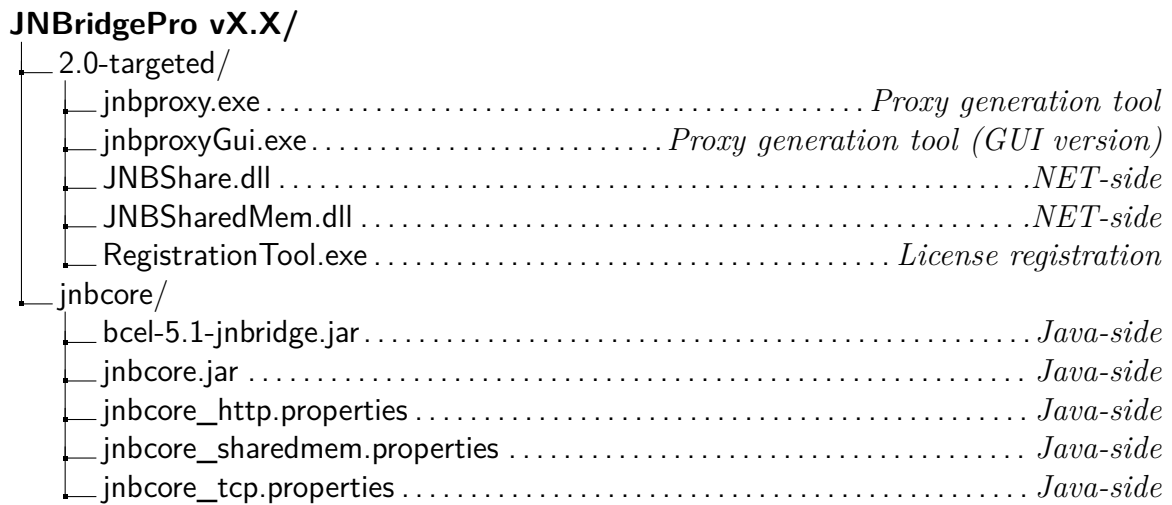


Figure 3.6.: JNBridge installation directory structure (only files we refer to are shown here).

Proxy.dll proxy library for use in a .NET environment. More on proxy generation in Section 3.3.3.

Deployment Configuration has to be selected when JNBridge is installed on a computer where .NET or Java applications will be run that make use of the JNBridge interoperability capabilities. This might be either a client machine for production use, e.g., a terminal in a supermarket that runs a .NET front-end that communicates with a server located Java back-end, or a workstation where the .NET side and the Java-side are hosted in parallel on the same computer. In our case, this is the convenient configuration for monitoring.

Figure 3.6 lists all files referred to in this chapter with their location inside the JNBridge installation directory.

3.3.2. License Activation

After installing JNBridge, *JNBridgePro registration tool* (**RegistrationTool.exe**) has to be launched to activate a valid license. For evaluation purposes, the 30-day trial-license-key received via email after registration may be sufficient.

When evaluating JNBridge for an open source project like Kieker, or for use in temporary project or work such as a thesis, a trial-extension-key for up to 6 months or even

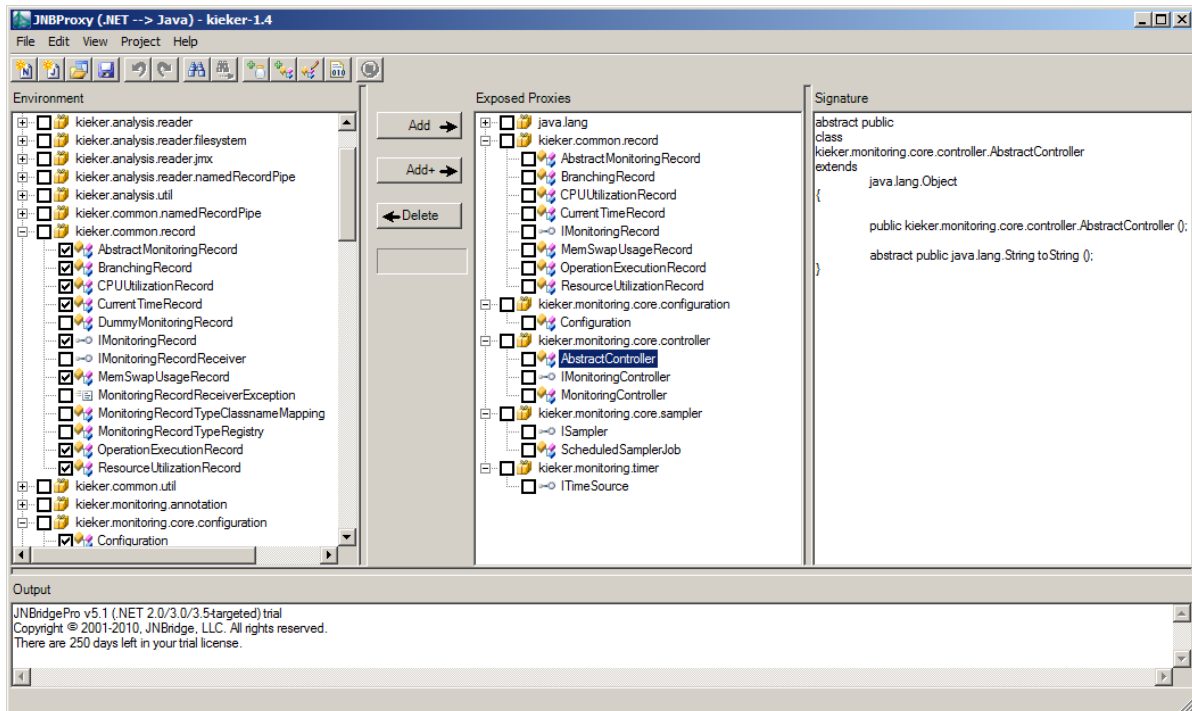


Figure 3.7.: GUI version of JNBProxy.

more can be obtained through contacting the JNBridge sales team. This is how we managed to work with JNBridge for the past 8 months.

3.3.3. Proxy Generation

JNBProxy, the JNBridge proxy generation tool, is installed with JNBridge when selecting *Development Configuration* during installation. To generate proxies, the developer may either launch the GUI version (`jnbproxygui.exe`) or the command-line version (`jnbproxy.exe`) of the tool. Figure 3.7 shows the main view of the GUI version.

- The *Environment* pane on the left side shows the loaded Java classes. For our project, the contents of the `kieker-1.4.jar` Java archive that ships with the Kieker distribution had been loaded.
- In the middle, the *Exposed Proxies* pane shows the classes that will be exposed via the proxy file that will be generated. The exposed classes correspond with the checkboxes in the Environment pane on the left side. This is how the developer

3. .NET Integration of Kieker

NAMESPACE	EXPOSED CLASSES
kieker.common.record	AbstractMonitoringRecord BranchingRecord CPUUtilizationRecord CurrentTimeRecord IMonitoringRecord MemSwapUsageRecord OperationExecutionRecord* ResourceUtilizationRecord
kieker.monitoring.core.configuration	Configuration*
kieker.monitoring.core.controller	AbstractController IMonitoringController* MonitoringController*
kieker.monitoring.core.sampler	ISampler ScheduledSamplerJob
kieker.monitoring.timer	ITimeSource*

Table 3.2.: Exposed Kieker classes.

selects the classes to expose. We only exposed classes that are reasonable to use on the .NET-side (Table 3.2).

- The right pane shows the *Signature* of a selected class. This provides all necessary information when selecting the classes to expose.

In Table 3.2 we list the exposed Kieker classes. All classes marked with (*) are actually used for manual instrumentation or by our Kieker.NET implementation. The other exposed classes are candidates for future extension of the work of porting the functional range of Kieker to .NET, e.g., by supporting other types of `kieker.common.record` for different monitoring purposes.

After defining what classes to expose, the library with the proxies can be generated and is then ready to be referenced and used by the target framework, which will be .NET in our case. As depicted in Figure 3.5, the resulting proxy classes embody a new Kieker component. Throughout the remainder of this document, we will refer to our generated proxy library that contains the exposed Kieker classes as `KiekerProxy.dll`.

The GUI version of JNBProxy also allows to generate a command-line script (BATCH file) for integration of proxy generation into a build process. See Listing B.1 for a script that can be used to generate the Kieker proxy library. For more detail on configuration and operation of the proxy generation tool see the JNBridge Users' Guide [JNBridge LLC. 2011b].

4. Dynamic Analysis With Kieker.NET

In this chapter, we present the first part of an extension of the original Kieker framework, which we call *Kieker.NET*, for dynamic analysis of .NET applications. In Section 4.1, we give a detailed introduction on how to configure the monitoring environment for dynamic analysis with Kieker.NET. Section 4.2 provides instructions on how to fulfill the minimum requirements for monitoring in .NET (i.e., monitoring probes) are provided that allow manual instrumentation by mixing monitoring logic with business logic.

Section 4.3 concludes this chapter with the results of a complete monitored run of the Bookstore sample application employing the techniques we developed so far.

4.1. Monitoring Configuration

4.1.1. Java-Side Configuration

Before using the generated proxy library `KiekerProxy.dll` in our Bookstore sample application, the Java-side, i.e., the computer that hosts the JVM with the exposed Kieker Java classes loaded, must be properly configured. Starting of the JVM can be configured to be triggered automatically. Then, whenever a first use of a proxy is made, a JVM with the corresponding Java classes available is started within the same process the .NET application is running. This can be configured in a configuration file on the .NET-Side as shown in Section 4.1.2. Another option is starting the Java-side manually. This adds another level of control, but is actually necessary if the Java-side resides on a different machine than the .NET-side.

In any case, the JVM with the exposed classes is started by executing a Java program (`com.jnbridge.jnbc core.JNBMain`) that ships with JNBridge as part of the library `jnbc core.jar`. Manual starting of the JVM can be done efficiently by using a batch file. Listing B.3 shows our `start-kieker-jvm.bat` batch file that can be used to start the Kieker

JVM for JNBridge operation. The JVM can also be started programmatically from within an existing JVM running another Java program.

Configuration of the JVMs communication parameters (essentially the IP address) must be done with a properties file. This can be either `jnbcore_sharedmem.properties` for shared memory communication, `jnbcore_http.properties` for HTTP/SOAP communication, or `jnbcore_tcp.properties` for communication via TCP/Binary as in our case. Listing B.4 shows our configuration for the Kieker JVM (note that these are the default files that ship with JNBridge).

Figure 3.6 helps locating the JNBridge-related files when editing the batch file. Analog to starting of the JVM, its configuration can also be done programmatically.

When the communication settings are properly configured, Kieker itself also has a `kieker.monitoring.properties` file that allows different monitoring configurations. As Kieker.NET basically provides only the .NET-compatible monitoring probes, all configuration entries in that file are still valid and necessary. Aside from basic settings like the name of the Kieker instance and the hostname (both to appear in the monitoring log file names), the actual monitoring log writer and different other monitoring parameters are set up here.

4.1.2. .NET-Side Configuration

On the .NET-side, all that has to be done to access the exposed Java classes is referencing the generated proxy library, in our case `KiekerProxy.dll`, together with the appropriate JNBridge runtime library, `JNBShare.dll` for TCP/IP communication or `JNBSharedMem.dll` for shared memory communication. The .NET application that uses the proxy classes needs some JNBridge-specific configuration entries to be added to its `app.config` file. This tells the .NET-side which communication mechanism should be used, the ip or hostname of the server (i.e., the Java-side), and whether the secure communication via *Secure Sockets Library* (SSL) should be used. These settings have to match those configured in the Java-side's `jnbcore_tcp.properties` or `jnbcore_http.properties` configuration file mentioned before when using TCP/Binary or HTTP/SOAP communication, respectively. Listing B.7 shows a sample `app.config` file—consisting solely of the JNBridge-related entries—configured for TCP/Binary communication. Listing B.8 shows the necessary entries when shared memory communication is used. For this, the correct locations of Java's `jvm.dll` (installed with any version of the Java Runtime Environment (JRE) or

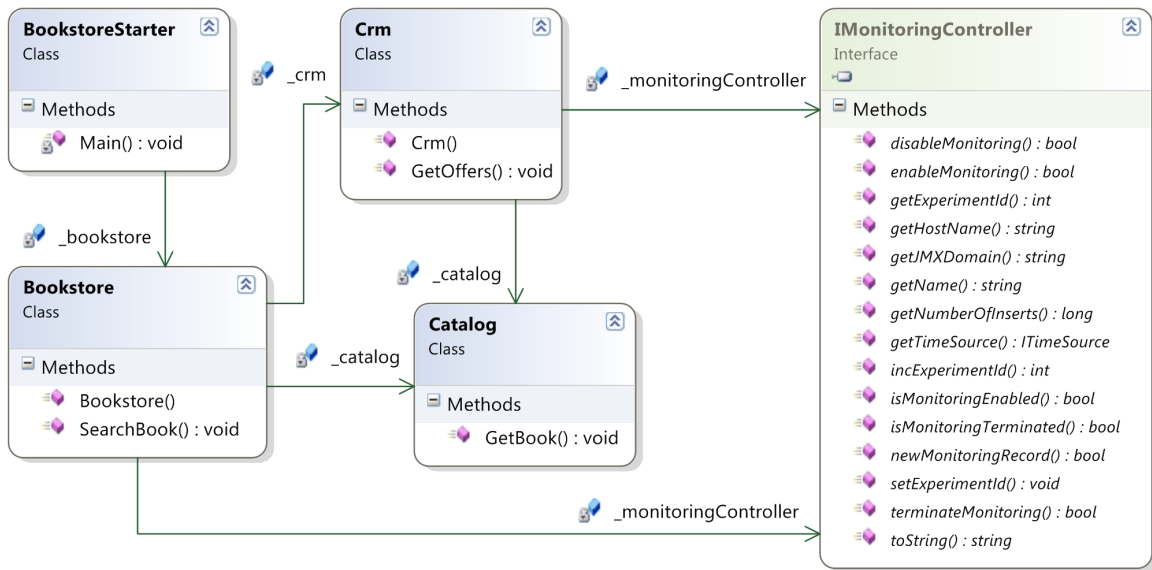


Figure 4.1.: Class diagram of the instrumented Bookstore sample application.

Java Development Kit (JDK)), as well as the location of JNBridge's own libraries have to be configured in the `App.config` file before execution. If the application does not yet have a configuration file, the naming scheme is `Application1.exe.config` for a compiled application `Application1.exe`.

Now that our JNBridge .NET/Java interoperability environment is set up, the next step is the implementation of the manual instrumentation on the .NET-side to allow monitoring with Kieker.NET.

4.2. Manual Instrumentation

Manual instrumentation with Kieker.NET can be implemented in a straightforward manner and is not much different to the way Kieker allows manual instrumentation of Java code as shown in Ehmke et al. [2011].

To demonstrate manual instrumentation, we use the C# Bookstore sample application, as introduced in Section 2.4. The instrumented classes can be found in the `BookstoreInstrumented` sub-folder of our Kieker.NET Visual Studio solution. Figure 4.1 shows how the bridged `MonitoringController` is integrated when instrumenting the classes

Bookstore and CRM. Note that all `MonitoringController` class members are actually available on the .NET-side.

4.2.1. Monitoring Probes

Listing 4.1 shows the instrumented `Bookstore` class. Specifically, we want to monitor the execution of method calls to `Catalog.GetBook()`. The parameter of type `bool` is not relevant in this context. Instrumentation of CRM is identical.

1. To monitor a method call in this class, first we need a reference to an instance of the Kieker monitoring controller class `MonitoringController`. This reference is obtained in Line 22. The call to the static `getInstance()` method is actually our first application of `JNBridge`. Note that the `using` directives (Lines 1, 2) also reference actual Java classes.
2. To monitor the execution time of calls to `Catalog.GetBook()`, we measure time before and after the call (Lines 28 and 34, directly surrounding the actual call in Line 31). Time measurement is also done by calling Kieker `getTime()` Java methods through `JNBridge`.
3. After that, we create a new `OperationExecutionRecord` with our time measurements and the name of the monitored method (Lines 37, 38) and hand it over to our `MonitoringController` (Line 42).

Validating and all further handling and storing of monitored data is done on the Java-side by Kieker and does not need to be changed.

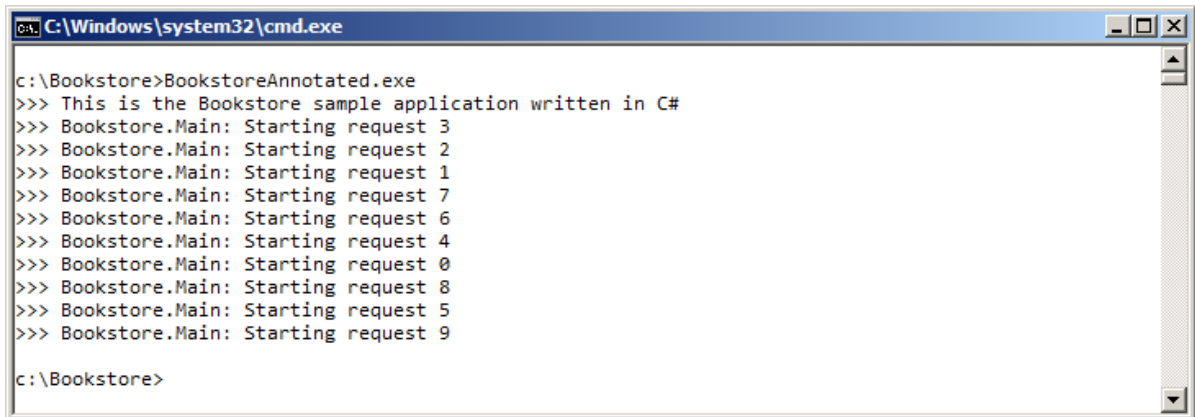
4.2.2. Monitoring Preliminaries

For runtime monitoring of the Bookstore sample application, we just need to build the Bookstore project with the instrumented `Bookstore` class shown above, and run its executable `BookstoreInstrumented.exe`.

It is important that the Kieker JVM is running on the Java-side before executing the instrumented application if TCP/Binary or HTTP/SOAP communication is chosen. As the .NET-side runtime library of `JNBridge` needs access to the JVM as soon as any

```
1 using kieker.common.record; // Bridged Java class.
2 using kieker.monitoring.core.controller; // Bridged Java class.
3
4 public class Bookstore
5 {
6     private readonly Catalog _catalog;
7     private readonly Crm _crm;
8
9     // Private field that holds a reference to Java-based
10    // MonitoringController.
11    private readonly IMonitoringController _monitoringController;
12
13    public Bookstore()
14    {
15        _catalog = new Catalog();
16        _crm = new Crm(_catalog);
17
18        // Retrieving a reference to Java-based MonitoringController
19        // by calling a static Java method.
20        _monitoringController = MonitoringController.getInstance();
21    }
22
23    public void SearchBook()
24    {
25        // Measure time before method call.
26        var tin = _monitoringController.getTimeSource().getTime();
27
28        // Actual method call to be monitored.
29        _catalog.GetBook(true);
30
31        // Measure time when method returns.
32        var tout = _monitoringController.getTimeSource().getTime();
33
34        // Create a new OperationExecutionRecord with monitored data.
35        var e = new OperationExecutionRecord(
36            typeof(Catalog).FullName, "GetBook()", tin, tout);
37
38        // Pass the new record to MonitoringController for further
39        // processing.
40        _monitoringController.newMonitoringRecord(e);
41        _crm.GetOffers();
42    }
43 }
44 }
```

Listing 4.1: Instrumented Bookstore class.



```
c:\Bookstore>BookstoreAnnotated.exe
>>> This is the Bookstore sample application written in C#
>>> Bookstore.Main: Starting request 3
>>> Bookstore.Main: Starting request 2
>>> Bookstore.Main: Starting request 1
>>> Bookstore.Main: Starting request 7
>>> Bookstore.Main: Starting request 6
>>> Bookstore.Main: Starting request 4
>>> Bookstore.Main: Starting request 0
>>> Bookstore.Main: Starting request 8
>>> Bookstore.Main: Starting request 5
>>> Bookstore.Main: Starting request 9
c:\Bookstore>
```

Figure 4.2.: Monitored execution of instrumented Bookstore sample application.

bridged type is accessed, the monitored application will throw a `TypeInitializationException` caused by JNBridge on startup otherwise. When using shared memory communication, the Kieker JVM is started by the monitored process itself.

4.3. Monitoring of the Bookstore Sample Application

The following Sections show the console output of both a monitored execution of our instrumented Bookstore sample application as well as the corresponding Kieker JVM.

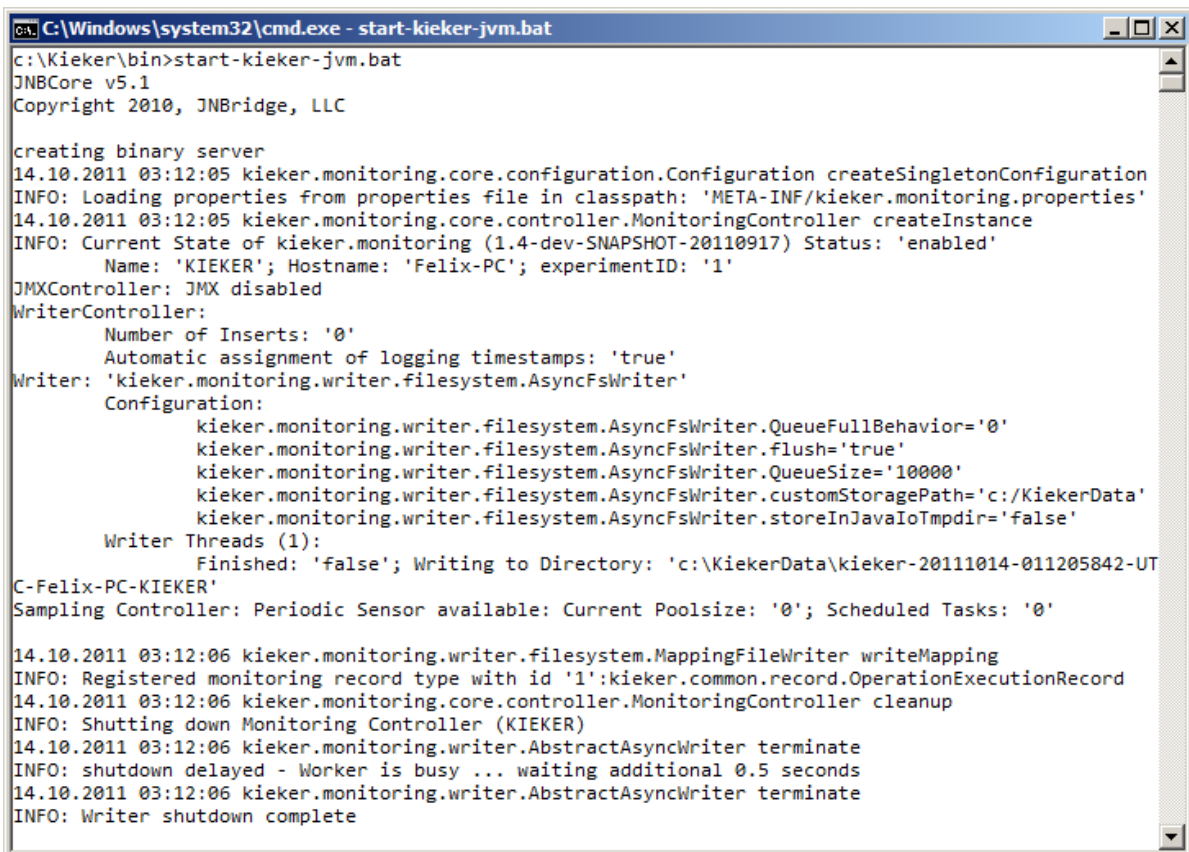
4.3.1. .NET-Side

On .NET-side, there is nothing notably different when executing the instrumented version of Bookstore. Figure 4.2 shows the console output of one execution of the Bookstore.

4.3.2. Java-Side

As pointed out before, the Kieker JVM has to be started before executing an instrumented application (when using TCP/binary communication as in our example). The state of the running JVM before monitoring is the output in the first four lines of Figure 4.3. After that, there is only output generated by Kieker itself. Most of it is printed before monitoring and reflects the actual configuration of the started `MonitoringController`.

4.3. Monitoring of the Bookstore Sample Application

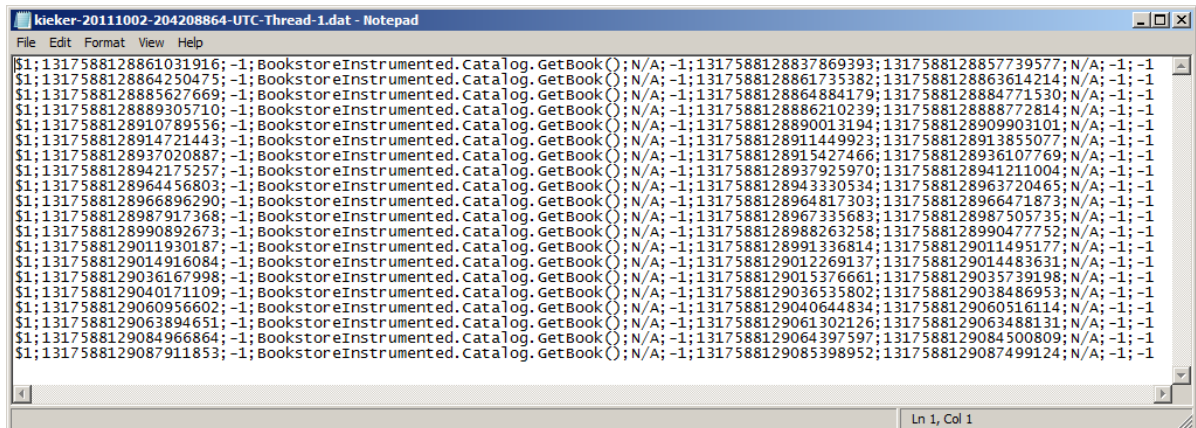


```
C:\Windows\system32\cmd.exe - start-kieker-jvm.bat
C:\Kieker\bin>start-kieker-jvm.bat
JNBCore v5.1
Copyright 2010, JNBridge, LLC

creating binary server
14.10.2011 03:12:05 kieker.monitoring.core.configuration.Configuration createSingletonConfiguration
INFO: Loading properties from properties file in classpath: 'META-INF/kieker.monitoring.properties'
14.10.2011 03:12:05 kieker.monitoring.core.controller.MonitoringController createInstance
INFO: Current State of kieker.monitoring (1.4-dev-SNAPSHOT-20110917) Status: 'enabled'
      Name: 'KIEKER'; Hostname: 'Felix-PC'; experimentID: '1'
JMXController: JMX disabled
WriterController:
  Number of Inserts: '0'
  Automatic assignment of logging timestamps: 'true'
Writer: 'kieker.monitoring.writer.filesystem.AsyncFsWriter'
  Configuration:
    kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior='0'
    kieker.monitoring.writer.filesystem.AsyncFsWriter.flush='true'
    kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize='10000'
    kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath='c:/KiekerData'
    kieker.monitoring.writer.filesystem.AsyncFsWriter.storeInJavaIoTmpdir='false'
  Writer Threads (1):
    Finished: 'false'; Writing to Directory: 'c:\KiekerData\kieker-20111014-011205842-UT
C-Felix-PC-KIEKER'
Sampling Controller: Periodic Sensor available: Current Poolsize: '0'; Scheduled Tasks: '0'

14.10.2011 03:12:06 kieker.monitoring.writer.filesystem.MappingFileWriter writeMapping
INFO: Registered monitoring record type with id '1':kieker.common.record.OperationExecutionRecord
14.10.2011 03:12:06 kieker.monitoring.core.controller.MonitoringController cleanup
INFO: Shutting down Monitoring Controller (KIEKER)
14.10.2011 03:12:06 kieker.monitoring.writer.AbstractAsyncWriter terminate
INFO: shutdown delayed - Worker is busy ... waiting additional 0.5 seconds
14.10.2011 03:12:06 kieker.monitoring.writer.AbstractAsyncWriter terminate
INFO: Writer shutdown complete
```

Figure 4.3.: Kieker JVM console output after instrumented Bookstore sample application has been run once.



```

kieker-20111002-204208864-UTC-Thread-1.dat - Notepad
File Edit Format View Help
$1;1317588128861031916;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128837869393;1317588128857739577;N/A;-1;-1
$1;1317588128864250475;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128861735382;1317588128863614214;N/A;-1;-1
$1;1317588128885627669;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128864884179;1317588128884771530;N/A;-1;-1
$1;1317588128889305710;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128886210239;131758812888872814;N/A;-1;-1
$1;1317588128910789556;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128890013194;1317588128909903101;N/A;-1;-1
$1;1317588128914721443;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128911449923;1317588128913855077;N/A;-1;-1
$1;1317588128937020887;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128915427466;1317588128936107769;N/A;-1;-1
$1;1317588128942175257;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128937925970;1317588128941211004;N/A;-1;-1
$1;1317588128964456803;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128943330534;1317588128963720465;N/A;-1;-1
$1;1317588128966896290;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128964817303;1317588128966471873;N/A;-1;-1
$1;1317588128987917368;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128967335683;1317588128987505735;N/A;-1;-1
$1;1317588128990892673;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128988263258;1317588128990477752;N/A;-1;-1
$1;1317588129011930187;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588128991336814;1317588129011495177;N/A;-1;-1
$1;1317588129014916084;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588129012269137;1317588129014483631;N/A;-1;-1
$1;1317588129036167998;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588129015376661;1317588129035739198;N/A;-1;-1
$1;1317588129040171109;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588129036335802;1317588129038486953;N/A;-1;-1
$1;1317588129060956602;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588129040644834;1317588129060516114;N/A;-1;-1
$1;1317588129063894651;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588129061302126;1317588129063488131;N/A;-1;-1
$1;1317588129084966864;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588129064397597;1317588129084500809;N/A;-1;-1
$1;1317588129087911833;-1;BookstoreInstrumented.Catalog.GetBook();N/A;-1;1317588129085398952;1317588129087499124;N/A;-1;-1
Ln 1, Col 1

```

Figure 4.4.: Resulting Kieker monitoring log for the instrumented Bookstore sample application.

4.3.3. Kieker Monitoring Log

After a successful run of the monitored application, in this case the execution of the compiled `BookstoreInstrumented.exe`, the configured Kieker's `AsyncFsWriter` wrote a monitoring log file to the hard disk of the machine that hosts the Java-side. Figure 4.4 shows the file that is generated through manual instrumentation as shown in the previous sections. As expected, Kieker monitored 20 calls to `Catalog.GetBook()`, where 10 calls are produced by `Bookstore` and the other 10 by `Crm`.

Note that a trace analysis is not possible with this data, as no entries (other than -1) are there for `eoi` and `ess` values [van Hoorn et al. 2009]. This is because we did not keep track of execution order or execution stack size with our basic manual instrumentation. An analysis of the involved components or component's operations is also not promising, as we kept track of only one single method. A more efficient way of instrumenting the code is required to allow a deeper analysis of complex applications. The following Chapter introduces much more powerful instruments that will cope with the instrumentation of even very complex systems easily.

5. AOP-Based Monitoring With Kieker.NET

To allow non-intrusive and efficient instrumentation of .NET applications, Kieker.NET supports monitoring based on the concepts of aspect-oriented programming (AOP). In Section 5.1, we give a basic overview of different types of AOP in .NET and present three different technologies that support AOP for .NET-based applications. The solution we chose for Kieker.NET is presented in detail in Section 5.2. One challenge when dealing with AOP is often the way the aspects are actually applied to the code. In Section 5.3, we show different ways of how Kieker.NET's main aspect can be applied.

5.1. Aspect-Oriented Programming in .NET

The principles of aspect-oriented programming (AOP) [Kiczales et al. 1997] can be applied to the .NET languages just as well as to any other object-oriented language. The core concept of cross-cutting concerns (i.e., aspects) is not language-restricted.

Several AOP frameworks for .NET exist, but they differ significantly in *how*, *when*, and *to which* targets aspects can be applied (or *woven*). Kiczales et al. [1997] defined two different approaches of *when* the aspects are applied to the code, *runtime weaving* and *compile-time weaving*. For .NET implementations of AOP, the same classification can be used.

5.1.1. Postsharp

Postsharp [SharpCrafters s.r.o. 2011a] calls itself a *static aspect weaver* because aspects developed with Postsharp can only be woven to the targeted code at compile-time (post-compile-time is also possible). The aspect weaver of Postsharp does this by altering the application's CIL code after it had been compiled (Figure 5.1).

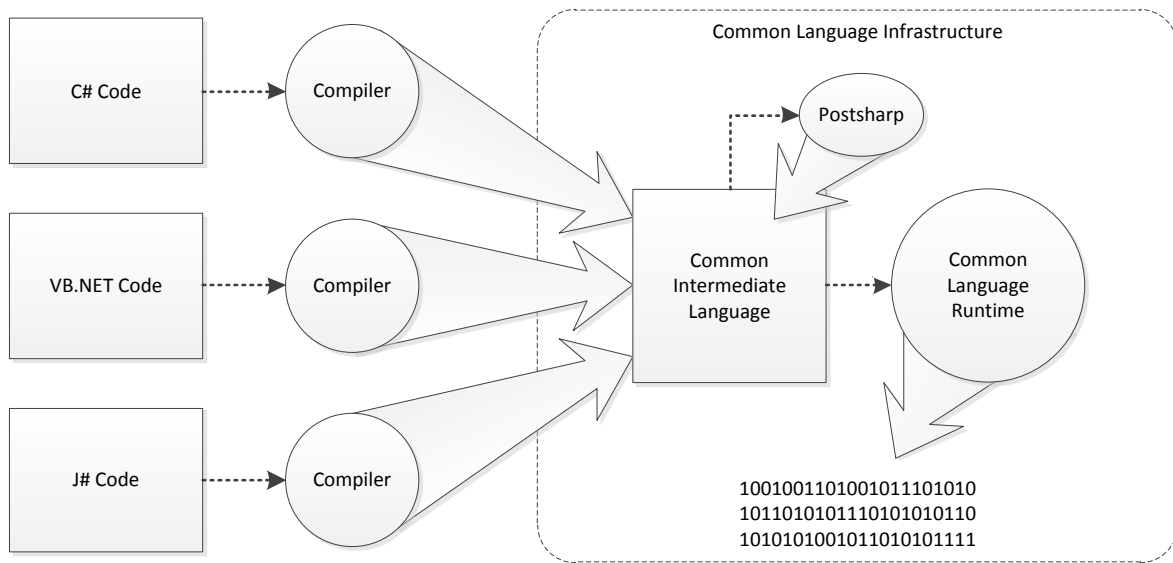


Figure 5.1.: Postsharp's post-compile-time aspect weaver.

Postsharp provides aspect parent classes and interfaces for most of the common cross-cutting concerns. Developing an aspect is done by deriving from such a class or implementing an interface and override/implement the aspect advices (i.e., the notifications when a method is entered and exited for the most common example, Postsharp's `OnMethodBoundaryAspect`).

Programmatically, aspect application can be done by utilizing .NET attributes (the .NET equivalent of Java annotations). When an aspect is applied to a method, schematically, the aspect transforms the method at compile-time as shown in Listing 5.1.

For more information on aspect development and application see Section 5.2 and the Postsharp reference documentation [SharpCrafters s.r.o. 2011b].

5.1.2. Spring.NET

Spring.NET¹ is an open source application framework led and sustained by SpringSource², the developers of the popular Spring development framework for Java.

¹<http://www.springframework.net/>

²<http://www.springsource.com/>

```
1 void InstrumentedMethod(...)
2 {
3     OnEntry();
4     try
5     {
6         // Original method body.
7         OnSuccess();
8         return returnValue;
9     }
10    catch ( Exception e )
11    {
12        OnException();
13    }
14    finally
15    {
16        OnExit();
17    }
18 }
```

Listing 5.1: Transformed method after aspect application.

Spring.NET supports AOP through dependency injection for applications that have been developed with the framework. As opposed to Postsharp, aspect weaving in Spring.NET is called *dynamic* because it will be done at runtime.

This always has the advantage that the application the aspect is applied to does not have to be altered at all, because the aspect can be woven into its bytecode at runtime.

There is one huge disadvantage coming with the AOP implementation of Spring.NET (aside from the fact that, first of all, a potential monitored application must be a Spring.NET application), namely the fact that only *virtual* methods can be targeted by aspects.

5.1.3. Castle DynamicProxy

Castle DynamicProxy³ is another dynamic AOP implementation for .NET. Compared to Spring.NET it has the “advantage” that it is not required for an application to be developed with a certain application framework (other than .NET itself). But it shares Spring.NET’s drawback of allowing only virtual methods to be targeted by the runtime aspect weaver. In C#, all methods are per default non-virtual (unlike Java methods).

³<http://docs.castleproject.org/>

5.2. Kieker.NET Implementation with Postsharp

The AOP implementation of the Java-based Kieker is based on AspectJ, which supports both compile-time and runtime weaving. Unfortunately, no equivalent framework exists for .NET, so we had to decide which way to go with our implementation.

We chose Postsharp for our work, as it is the de facto standard when it comes to compile-time aspect weaving in a .NET-based environment. Moreover, the limited applicability of both of the other presented solutions was not acceptable, especially with the case study system in mind.

Our Postsharp-based implementation of Kieker.NET consists of the C# classes shown in the class diagram in Figure 5.2. We adopted the namespace from the Java-based Kieker framework, ported all the code that has to be executed platform-specific, and finally added new classes to the corresponding contexts. As we wanted to avoid code redundancies, we only ported code where it was necessary. In the following sections, we present our implementation in detail and will also comment on the platform-specific changes that had to be done to the Java-based classes. The complete source code for the presented classes can be found in Appendix A.2.3.

5.2.1. The OperationExecutionAspect Class

As introduced in Section 5.1.1, Postsharp basically is a post-compiler that weaves aspects into applications or function libraries by altering their Common Intermediate Language (CIL) bytecode (see Section 2.3 for an introduction to the concepts of .NET). Our *cross-cutting concern* for monitoring is the creation of `OperationExecutionRecords` at runtime. We want every instrumented method to have monitoring code—the probes—executed *before* and *after* the method call.

The aspect that contains this code is our class `OperationExecutionAspect`. Figure 5.2 shows the class diagram with the aspect itself and its base class that is provided by Postsharp, alongside with other supporting classes that will be introduced in the following sections. See Chapter A.2.3 for the complete source code of these classes. We explain our implementation by describing the (`public`) methods of our aspect. Note that the four implemented methods are all inherited or implemented from Postsharp aspect base classes or interfaces (also shown in Figure 5.2).

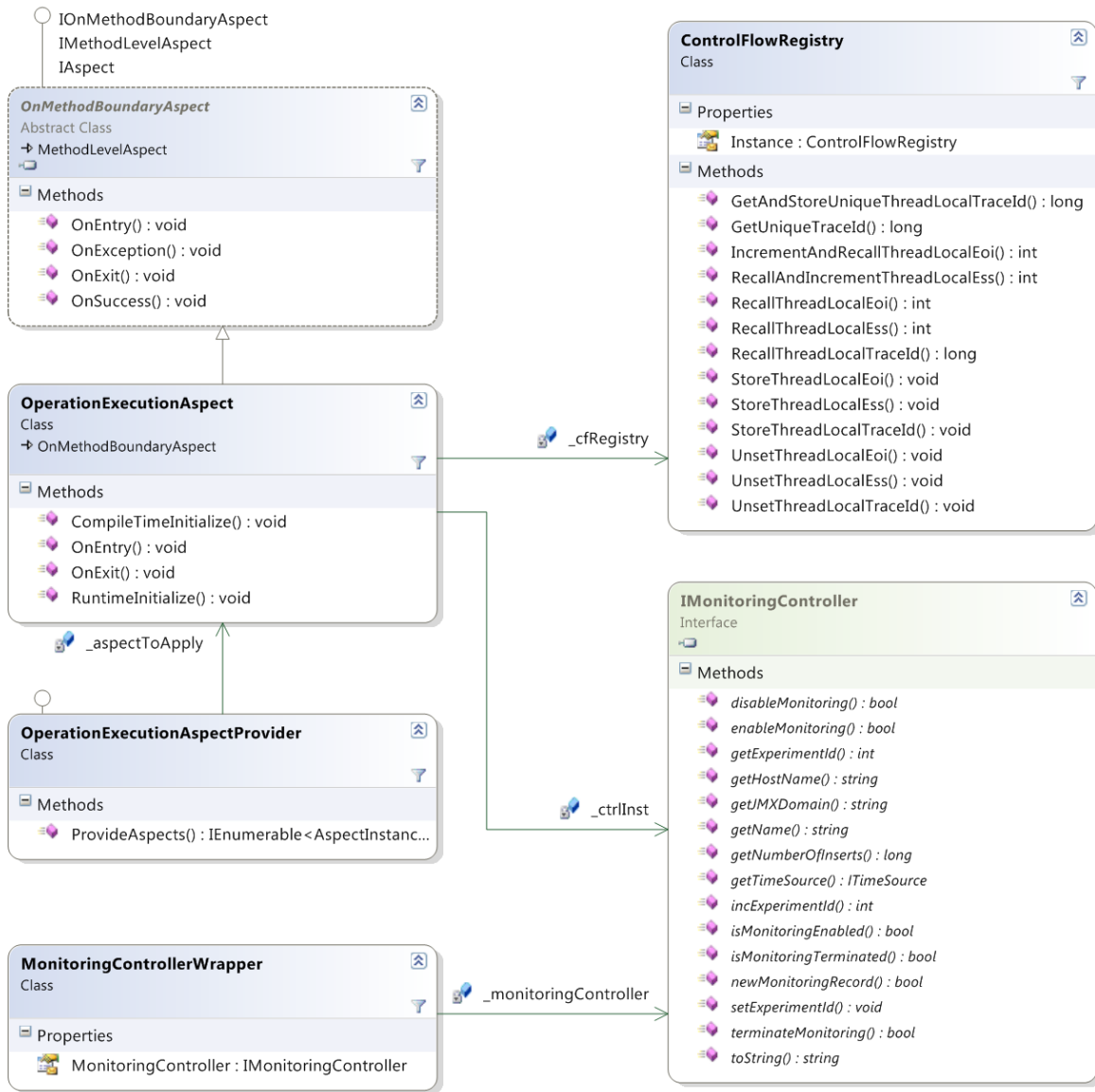


Figure 5.2.: Class diagram of all Kieker.NET classes.

CompileTimeInitialize() is used for gathering information about the monitored method (i.e., its name and parameter types) as well as the declaring type. This can be done at compile-time (or *build-time*), so that no use of `System.Reflection` has to be made at runtime. As reflection in .NET is done by examination of an assembly's meta-data, avoiding such operations at runtime is preferable whenever possible. Especially when dealing with costly operations like string formatting as in our case (the methods for that are omitted here), taking advantage of compile-time initialization is most certainly beneficial for the performance at runtime.

```
1 public override void CompileTimeInitialize(MethodBase method, ...)
2 {
3     _componentName = FormatType(method.DeclaringType.FullName);
4     _method = FormatMethodName(method.Name)
5         + FormatParameters(method.GetParameters());
6 }
```

Listing 5.2: Aspect Compile-time initialization method.

RuntimeInitialize() deals with initialization of the aspect's references to the `MonitoringController` and the `ControlFlowRegistry`. This method is executed on aspect load-time, i.e., when the class that contains the implementation of monitored methods is loaded for the first time. This is due to the fact that our bridged access to the Java-side Kieker JVM can only be established at runtime. Trying to initialize bridged types at compile-time results in a `TypeInitializationException` in the build process—a very uncommon behavior. In this case, the type of our field `_ctrlInst` is such a bridged Java type, accessed through the `MonitoringControllerWrapper`.

```
1 public override void RuntimeInitialize(MethodBase method)
2 {
3     _ctrlInst = MonitoringControllerWrapper.MonitoringController;
4     _cfRegistry = ControlFlowRegistry.Instance;
5 }
```

Listing 5.3: Aspect runtime initialization method.

OnEntry() contains the code that will be executed before the body of the monitored method is entered. In our case, this is time measurement, the initialization of a `OperationExecutionRecord`, and to support tracing, we also manage counter for

execution order and call stack size at this point. To allow concurrency, we have to initialize the `OperationExecutionRecord` locally, thus avoiding interference from possible concurrent calls to the same monitored method. This is only possible because we can store the collected data in the aspect context for later use.

```

1 public override void OnEntry(MethodExecutionArgs args)
2 {
3     if (!_ctrlInst.isMonitoringEnabled()) { return; }
4     OperationExecutionRecord execData = InitExecutionData();
5     int eoi = 0; // execution order index
6     int ess = 0; // execution stack size
7     if (execData.isEntryPoint)
8     {
9         _cfRegistry.StoreThreadLocalEoi(0);
10        _cfRegistry.StoreThreadLocalEss(1);
11    }
12    else
13    {
14        eoi = _cfRegistry.IncrementAndRecallThreadLocalEoi();
15        ess = _cfRegistry.RecallAndIncrementThreadLocalEss();
16    }
17    if ((eoi == -1) || (ess == -1))
18    {
19        _ctrlInst.terminateMonitoring();
20    }
21    execData.eoi = eoi;
22    execData.ess = ess;
23    // Time when monitored method begins execution.
24    execData.tin = _ctrlInst.getTimeSource().getTime();
25    // Store execData for further use in OnExit().
26    args.MethodExecutionTag = execData;
27 }

```

Listing 5.4: `OperationExecutionAspect`'s `OnEntry()` advice.

OnExit() is executed when the monitored method returns. Here, we measure the time again, and hand over all data to the (Java-based) `MonitoringController`.

```

1 public override void OnExit(MethodExecutionArgs args)
2 {
3     if (!_ctrlInst.isMonitoringEnabled()) { return; }
4     // Restore execData.
5     OperationExecutionRecord execData = (OperationExecutionRecord)
6         args.MethodExecutionTag;
7     // Time the monitored method is finished.
8     execData.tout = _ctrlInst.getTimeSource().getTime();

```

```
8     if (execData.isEntryPoint)
9     {
10        _cfRegistry.UnsetThreadLocalTraceId();
11    }
12    // Create a new monitoring record with the measured data.
13    _ctrlInst.newMonitoringRecord(execData);
14    if (execData.isEntryPoint)
15    {
16        _cfRegistry.UnsetThreadLocalEoi();
17        _cfRegistry.UnsetThreadLocalEss();
18    }
19    else
20    {
21        _cfRegistry.StoreThreadLocalEss(execData.ess);
22    }
23 }
```

Listing 5.5: OperationExecutionAspect's OnExit() advice.

Of the four advices of the base aspect `OnMethodBoundaryAspect`, we only implement `OnEntry()` and `OnExit()`, as we are not interested in exception logging in this context.

5.2.2. The ControlFlowRegistry Class

To support (thread-safe) call tracing, we needed to re-implement the `ControlFlowRegistry` of Java-based Kieker in C#. The sequence diagram in Figure 5.3 shows the problem that occurs when using the bridged `ControlFlowRegistry`. There was no way to ensure that the `ThreadLocal` fields of the registry on Java-side are accessed by the exact threads that initiated the aspect in the first place. Instead, we observed that aspects were not able to recall their (supposed-to-be thread-local) values they stored in the registry in previous executions.

The functionality of the methods of our implementation of `ControlFlowRegistry`—which are basically accessor and mutator methods for thread-local fields `traceId`, execution order index (`eoi`), and execution stack size (`ess`)—is mostly unchanged from the Java version.

Listing A.4 shows the source code of this class. Analog to the Java-based class, we used the Singleton pattern [Gamma et al. 1995] to ensure that only one instance of the `ControlFlowRegistry` exists at a time and all aspects have the same reference to it. This is important, because we want to store global call traces, and identifying the connection between data generated by single aspects would be impossible otherwise.

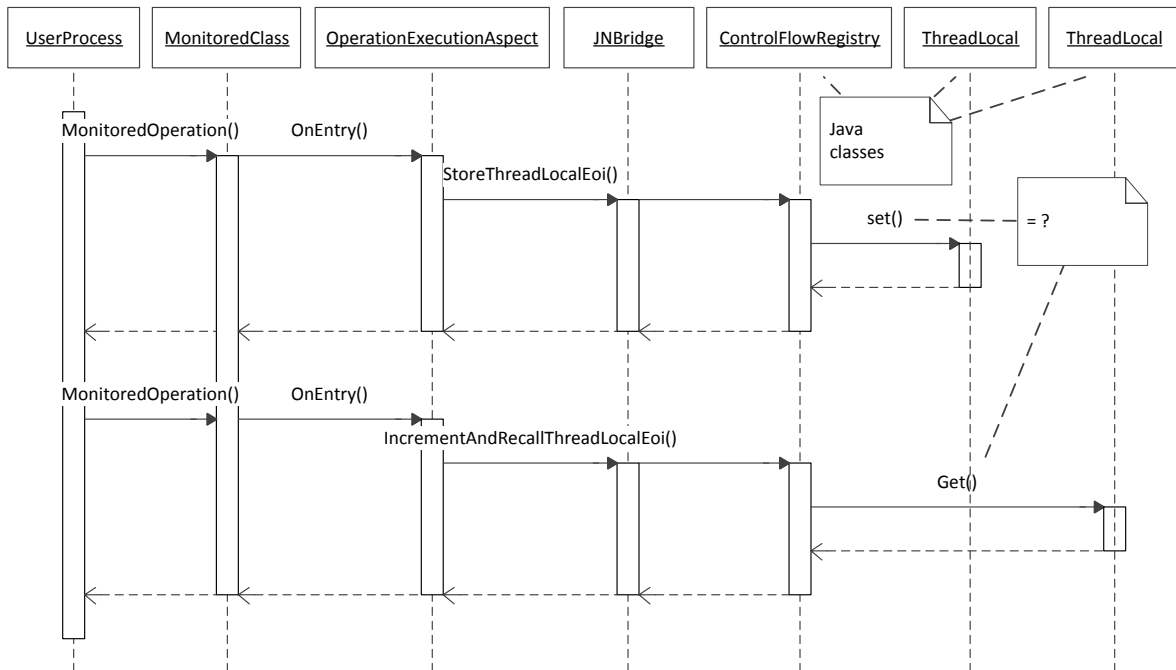


Figure 5.3.: Multithreading problems with bridged ControlFlowRegistry Java class.

For the same reasons, the `SessionRegistry` class of Java-based Kieker should also be re-implemented when the functional range of Kieker.NET is extended to support session-based monitoring.

5.2.3. The MonitoringControllerWrapper Class

As the name suggests, this class is a wrapper for the Java-based `MonitoringController` instance which we access via `JNBridge`. Basically, this would not be necessary, but we wanted to hook into the `ProcessExit` event of the monitored application to send a `terminateMonitoring()` signal to the `MonitoringController`. Aside from providing a reference to the singleton `MonitoringController` to our .NET-side, this is all we do here. Structurally, inheriting from the Java-based `MonitoringController` and overriding the shutdown logic would be possible, but the class is `final` on Java-side, and this also applies to the .NET-side. See Listing A.3 for the source code of this class.

5.2.4. The `OperationExecutionAspectProvider` Class

This class, an implementation of Postsharp's `IAспектProvider`, can be used to apply aspects to compiled function libraries (dll) or applications (exe) where no alterations of source code and no new build is needed. More details on this in Section 5.3.4.

5.3. Aspect Application

Our Postsharp-based implementation of Kieker.NET supports several levels to apply the aspect, i.e., the Kieker monitoring probe, to an application. Sections 5.3.1, 5.3.2, and 5.3.3 deal with applications techniques that involve different degrees of changes to the targeted source code and new build of the targeted application. Additionally, Section 5.3.4 presents an approach for aspect application to compiled assemblies that cannot be altered by editing their source code.

5.3.1. Method Level

Aspect application on method level can be considered the common case. This simply allows instrumentation of a method without mixing monitoring and business logic. Listing 5.6 shows how our `OperationExecutionAspect` can be applied to the Bookstore sample application's `Catalog.GetBook()` method by simply adding the `[OperationExecutionAspect]` attribute to it.

```
1 public class Catalog
2 {
3     [OperationExecutionAspect]
4     public void GetBook(bool complexQuery)
5     {
6         Thread.Sleep(complexQuery ? 20 : 2);
7     }
8 }
```

Listing 5.6: Method-level aspect application.

All calls to `Catalog.GetBook()` are now monitored. Listing 5.7 shows one entry of the monitoring log that is generated by Kieker for this instrumentation when running the Bookstore sample application once.

```

1 $! ;1317074261977377840;1;BookstoreAnnotated.Catalog.GetBook(System.Boolean);N/A
  ;3242591731706757121;1317074261974127505;1317074261976446825;Felix-PC;0;0

```

Listing 5.7: Monitoring log entry for instrumented `Catalog.GetBook()` method.

For the 10 iterations of `BookstoreStarter` calling `Bookstore.SearchBook()`, Kieker created exactly 20 monitoring log entries, all of them much similar to the one shown in Listing 5.7, with zero values on logged execution order index and execution stack size (the last two values). This is because the instrumentation of one single method cannot produce traces as long as no recursion is involved.

5.3.2. Class Level

Class-level aspect application is much similar to method-level application. Here, the `OperationExecutionAspect` attribute is added to classes directly. This automatically applies the aspect to all methods of a class, regardless of their access level modifier. In our `Bookstore` sample application, these classes could be `Bookstore` and `Catalog`, which now also should allow us to actually monitor traces (note that this is also possible with method-level aspect application, even if the effort of applying the aspect would be higher).

```

1 [OperationExecutionAspect]
2 public class Bookstore
3 {
4     private readonly Catalog _catalog;
5     private readonly Crm _crm;
6
7     public Bookstore()
8     {
9         _catalog = new Catalog();
10        _crm = new Crm(_catalog);
11    }
12
13    public void SearchBook()
14    {
15        _catalog.GetBook(false);
16        _crm.GetOffers();
17    }
18 }

```

Listing 5.8: Class-level aspect application.

Listing 5.8 exemplifies class-level aspect application on the `Bookstore` class. Adding the attribute to the class constructor (`Bookstore()`) and the `SearchBook()` method instead of the class would have the exact same effect. For `Catalog`, the attribute just can be moved from the `GetBook()` method to the class declaration.

```

1 $!;1317137637484078573;1;BookstoreAnnotated.Catalog.ctor();N/A
  ;1837468647967162369;1317137637482419620;1317137637483238505;Felix-PC;1;1
2 $!;1317137637485977494;1;BookstoreAnnotated.Bookstore.ctor();N/A
  ;1837468647967162369;1317137637479510425;1317137637485432545;Felix-PC;0;0
3 $!;1317137637494477895;1;BookstoreAnnotated.Catalog.GetBook(System.Boolean);N/A
  ;1837468647967162370;1317137637492050826;1317137637493944268;Felix-PC;1;1
4 $!;1317137637499171323;1;BookstoreAnnotated.Catalog.GetBook(System.Boolean);N/A
  ;1837468647967162370;1317137637496517801;1317137637498814476;Felix-PC;2;1
5 $!;1317137637500195841;1;BookstoreAnnotated.Bookstore.SearchBook();N/A
  ;1837468647967162370;1317137637489299418;1317137637499837534;Felix-PC;0;0
6 $!;1317137637507210781;1;BookstoreAnnotated.Catalog.GetBook(System.Boolean);N/A
  ;1837468647967162371;1317137637504758511;1317137637506838594;Felix-PC;1;1
7 $!;1317137637511214257;1;BookstoreAnnotated.Catalog.GetBook(System.Boolean);N/A
  ;1837468647967162371;1317137637509239364;1317137637510848644;Felix-PC;2;1
8 $!;1317137637513432038;1;BookstoreAnnotated.Bookstore.SearchBook();N/A
  ;1837468647967162371;1317137637502548764;1317137637513064965;Felix-PC;0;0
9 ...

```

Listing 5.9: Monitoring log entry for class-level attributed `Bookstore` sample application.

Monitoring the execution of the class-level attributed classes produces the monitoring log shown in Listing 5.9. This time, the tracing values are zero and non-zero, which would allow us to consider a first Kieker trace analysis (although the resulting diagrams would be incomplete, as not all classes (i.e., `Crm`) are instrumented and therefore not under observation).

5.3.3. Assembly Level

Attributing all classes of an assembly produces the same result as the one single instruction shown in Listing 5.10. With this, we apply the aspect to all methods of all classes contained in the assembly.

```

1 [assembly: OperationExecutionAspect]

```

Listing 5.10: Assembly attribute for assembly-level aspect application.

In fact, the `[assembly: ...]` attribute can be anywhere inside any class of the whole assembly, but it can be considered good practice to have a dedicated “class” file (in our case named `AspectInfo.cs`) added to the other sources that contain the classes that will be monitored.

It may be necessary to control to what methods the aspect will actually be applied. The above example is an unlimited *multicast* that makes no restrictions, resulting in the instrumentation of all classes' methods. Listing 5.11 shows three example assembly attributes that can be used to manage aspect application.

```

1 [assembly: OperationExecutionAspect(
2     AttributeTargetTypes = "Bookstore.*")]
3
4 [assembly: OperationExecutionAspect(
5     AttributeExclude = true,
6     AttributeTargetTypes = "Bookstore.Crm.*")]
7
8 [assembly: OperationExecutionAspect(
9     AttributeExclude = true,
10    AttributeTargetMembers = "regex:ctor|get_.*|set_.*")]

```

Listing 5.11: Assembly attributes for managed assembly-level aspect application.

By specifying the `AttributeTargetTypes` property of the first attribute, we instruct Postsharp to apply our aspect to all classes in the `Bookstore` namespace (for our Bookstore example application, this is analog to the attribute in Listing 5.10). With the second attribute, we exclude the classes in the `Bookstore.Crm` namespace—that have been included before—by specifying the `AttributeExclude=true` property. The third attribute removes constructors and accessor and mutator methods from our observation.

Especially the third attribute can be quite valuable when monitoring complex software systems where a too coarse grained positioning of monitoring points can lead to unmanageable amounts of monitoring output (logs), as well as poor performance [van Hoorn et al. 2009]. For more detail on how to specify aspect target elements, see the Postsharp reference documentation [SharpCrafters s.r.o. 2011b].

We now finally have a convenient tool that allows comprehensive dynamic analysis of .NET applications. With assembly-level aspect application, we can target all classes of an application at once, without having to modify any of the original classes. For the Bookstore sample application, the operation dependency graph and the sequence diagram that are shown in the Foundations chapter in Section 2.4 are actually generated by monitoring the .NET-based Bookstore sample application with assembly-level aspect application.

5.3.4. Build-Independent Aspect Application

With the help of a new feature in the newest version of Postsharp (version 2.1) and our `OperationExecutionAspectProvider` introduced in Section 5.1.1, we can apply the `OperationExecutionAspect` also to compiled assemblies, regardless whether targeting pure function library assemblies (dll) or executables (exe).

```
1 @ECHO OFF
2 SETLOCAL
3
4 SET POSTSHARP=C:\Program Files (x86)\PostSharp 2.1
5 SET .NET_VERSION=2.0
6 SET ARCHITECTURE=x86
7 SET TARGET_ASSEMBLY=Bookstore.exe
8
9 @"%POSTSHARP%\Release\postsharp.%.NET_VERSION%-%ARCHITECTURE%-cil.exe" "
   %TARGET_ASSEMBLY%" /p:AspectProviders=Kieker.Monitoring.Probe.Postsharp
   .OperationExecution.OperationExecutionAspectProvider,Kieker /p:Output=
   output\Bookstore.exe
10
11 @COPY /y *.dll .\Output
12 @PAUSE
```

Listing 5.12: Batch file to apply a given aspect to an assembly.

Listing 5.12 shows how Postsharp can be utilized for this type of aspect application. Here, we apply the `OperationExecutionAspect` to our non-instrumented `Bookstore.exe` compiled executable. Afterwards, the now instrumented file will be in a subfolder called `Output`, together with all needed libraries.

```
1 foreach (MethodInfo targetMethod in type.GetMethods(BindingFlags.Instance
   | BindingFlags.Public | BindingFlags.DeclaredOnly))
2 {
3   ...
4 }
```

Listing 5.13: Configuring the aspect targets programmatically.

The aspect targets can be defined in the `OperationExecutionAspectProvider` class. Listing 5.13 shows the line of this class where this can be done programmatically. It would probably be better to extend the aspect provider, so that the aspect targets can be defined by an external configuration file, e.g., in XML format.

Creating a `Bookstore.exe.config` file as described in Section 4.1.2 with the entries for `JNBridge` is all that is needed to monitor execution of `Bookstore.exe` with `Kieker.NET` without the need to have access to the source code.

Any usage of the interface `IAспектProvider`—that includes our `OperationExecutionAspectProvider`—requires a valid *Professional* license of Postsharp registered on the computer where the aspect application as described above will be performed.

6. Overhead Evaluation

Evaluation of the performance of Kieker.NET is the main concern of this chapter. For this, we have to quantify the overhead that is induced by Kieker.NET to an application under observance. We present monitoring overhead analysis with a micro-benchmark, following van Hoorn et al. [2009], and concentrate on quantitative assessment, more precisely on quantifying the monitoring overhead that is induced by Kieker.NET.

That Kieker.NET works for small example applications (i.e., that the logged data is valid and can be analyzed further with Kieker's analysis tools), has already been shown in our running examples throughout Chapters 4 and 5.

By measuring and assessing the monitoring overhead and comparing the results with reference values obtained by runtime analysis of non-instrumented applications, we present information that may allow to weigh whether monitoring of certain components is adequate (e.g., complex computation of a mathematical problem that involves a large sum of iterative steps).

Because of the number of involved frameworks and utilities (Postsharp, JNBridge, Java), we expect several different causes of overhead with their diverse performance (i.e, response time) impacts. We try to break down and identify these causes in Section 6.1. A convenient experiment design that allows us to distinguish and quantify all involved causes for overhead is introduced in Section 6.2. Finally, we present the results of our analysis in Section 6.3.

6.1. Causes of Overhead

When the decisions to employ the different frameworks for Kieker.NET were made, their impact on the pure performance was not the main deciding factor. As we described in Sections 3.1 and 5.1, it was more the practicality and technical maturity of the solutions that abetted their employment.

The overhead caused by the original Java-based Kieker was thoroughly evaluated and analyzed by van Hoorn et al. [2009]. Here, we need to focus on the .NET-related technologies, namely Postsharp and JNBridge. Especially JNBridge is expected to have a significant impact on performance.

However, as we evaluate Kieker.NET as a whole, we have to be aware of the different causes for the expected overhead. With this in mind, the more fine-grained our observations can be, the better. Hence, the goal for our experimental setting(s) must be finding a way to distinguish each source of overhead individually. Possible causes of overhead are:

.NET Framework itself is supposedly a cause for overhead, given its application virtual machine approach as opposed to native compiled code (e.g., compiled C or COBOL applications). For two reasons we are not going to incorporate this into our performance analysis at all. First, the native Java Kieker runs in a comparable Java environment as well, so there is not point in considering .NET as a cause of overhead. And second, all applications that can potentially be monitored with Kieker.NET are affected by a possible overhead themselves.

Postsharp is potentially the first cause of overhead that is directly related to our technological choices when implementing Kieker.NET. Here, we have to assess the overhead that comes with the employment of AOP. The aspect class of Postsharp we chose for Kieker.NET places code (i.e., advices) on a methods boundary. To what extent the performance impact of this is linear to the amount of instructions added is to be analyzed.

JNBridge with its .NET/Java intercommunication features is expected to account for the biggest part of Kieker.NET's monitoring overhead. According to the JNBridge user guide, it is also crucial which communication mechanism is used, as shared memory is supposed to be much faster than the other two alternatives TCP/binary and HTTP/SOAP. We will try to verify this by using both in our experiments.

Java-side Kieker is somewhat directly related to JNBridge. Without the class loader of JNBridge, there would be no active Kieker instance. And analyzing the (communication) overhead of JNBridge without the respective Java classes to communicate with would not make any sense. There is, of course, a certain amount of monitoring to be expected from the Java classes, because—after all—their methods are

called via JNBridge. Quantifying this very amount is most probably outside of the focus of this analysis, as it would require additional alterations to the Java classes, which is not envisioned.

Our Experiment design to be introduced in the next section attempts to take all different causes of monitoring overhead into account by introducing a staged design.

6.2. Experiment Design

All experiments presented in this chapter are performed on an up-to-date workstation, powered by an Intel Xeon W3530 quad-core processor running at 2.8 GHz, with 12 GB DDR3 memory, and an Intel 510 series solid-state drive. The operating system is Microsoft Windows 7 Professional 64-bit. The micro-benchmark is a Microsoft .NET 3.5 project, using Postsharp version 2.1.3.3 and JNBridgePro version 5.1. The Java-side (Kieker 1.4-dev-SNAPSHOT) is hosted by a version 1.6.0_24-b07 JVM.

6.2.1. Benchmark

The benchmark is a .NET 3.5 console application. An excerpt with the part where the actual method call of the monitored method is made is shown in Listing 6.1.

```
1     #region Benchmark
2
3     var monitoredClass = new MonitoredClass();
4     var stopWatchSingle = new Stopwatch();
5
6     for (var i = 0; i < TotalOperationCalls; i++)
7     {
8         stopWatchSingle.Start();
9         monitoredClass.MonitoredOperation(MethodTime);
10        stopWatchSingle.Stop();
11        if (i >= TotalOperationCalls - MonitoredOperationCalls)
12        {
13            ResponseTimes.Add(stopWatchSingle.Elapsed.TotalMilliseconds);
14        }
15        stopWatchSingle.Reset();
16    }
17
18    #endregion
```

Listing 6.1: Performance analysis benchmark (excerpt).

Listing 6.2 shows the `MonitoredClass` with its single `MonitoredOperation()`. We tried to keep the non-instrumented execution time of the monitored method as close as possible to the reference value given by the parameter `methodTime` (we chose 100 microseconds). To achieve this, we also utilized the .NET `Stopwatch` class, the same class used for the actual response time measurement as shown before.

```
1 class MonitoredClass
2 {
3     readonly Stopwatch _stopWatch = new Stopwatch();
4
5     [OperationExecutionAspect]
6     public void MonitoredOperation(double methodTime)
7     {
8         _stopWatch.Start();
9         while (_stopWatch.Elapsed.TotalMilliseconds < methodTime)
10        {
11        }
12        _stopWatch.Reset();
13    }
14 }
```

Listing 6.2: The monitored class containing the single monitored method with parameterized (operation) response time.

In one benchmark run, we call `MonitoredOperation()` exactly 100,000 times and store each response time in an array after a call returns. To avoid scattered results due to including the “warm-up phase” in our analysis, we only record the last 10,000 calls made. Later, we calculate mean, median, and standard deviation values for these 10,000 recorded calls. To verify the results, we complete one “initialization” run and five “live” runs subsequently for each stage of the benchmark.

6.2.2. Staged Benchmark Execution

The following staged benchmark design is the core of our analysis. We try to incorporate and distinguish most causes of monitoring overhead by successively enabling the different technologies and frameworks in the five stages of our experiment design. It follows a description of the stages and justification for why we think which exact overhead is triggered and therefore identified in each single stage.

Stage I: Reference

The first stage is the non-instrumented benchmark application. The results of this stage are the reference values for all further comparisons. Every additional microsecond needed by any of the following stages can be considered overhead. The sequence diagram in Figure 6.1(a) shows this. The reference time for the monitored operation will be denoted as Δ_I in the remainder of this chapter.

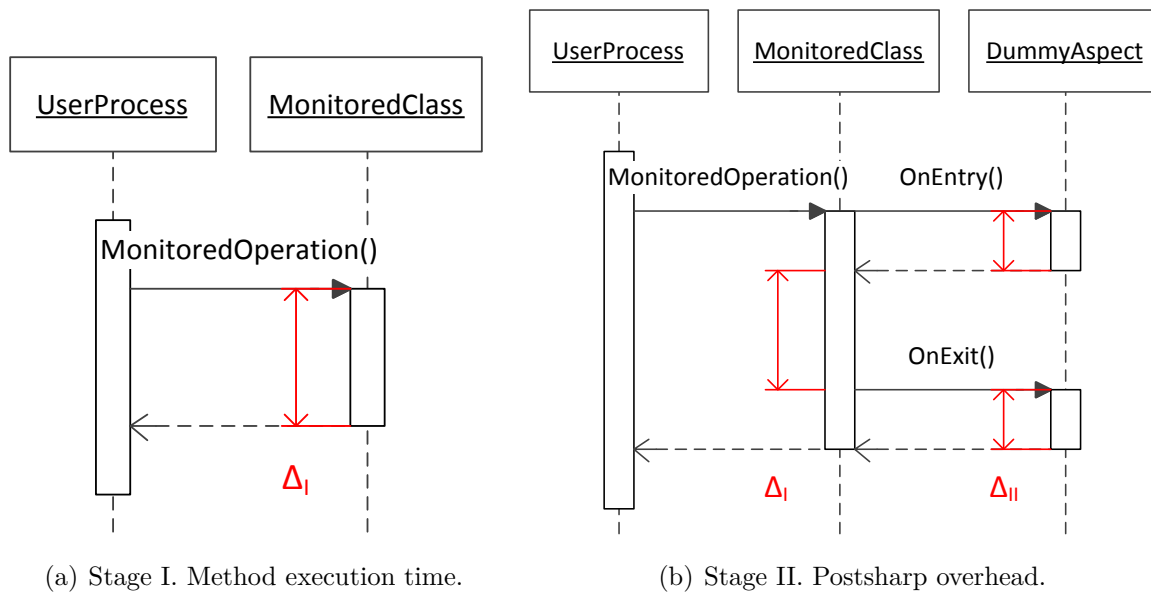


Figure 6.1.: Stages I and II of the experiment design.

Stage II: Postsharp

In the second stage, the overhead that comes with the employment of Postsharp is analyzed. For that, the `MonitoredOperation()` will be “instrumented” with the quasi empty `DummyAspect` shown below in Listing 6.3. The `OnEntry()` and `OnExit()` methods are not complete stubs to avoid possible compiler optimizations, but the added instructions can be considered as non-demanding.

The resulting sequence diagram in Figure 6.1(b) shows the expected overhead Δ_{II} . As Postsharp is a compile-time aspect weaver, we expect the performance impact of this stage to be relatively insignificant.

```
1 [Serializable]
2 class DummyAspect : OnMethodBoundaryAspect
3 {
4     private int _x;
5
6     public override void OnEntry(MethodExecutionArgs args)
7     {
8         _x = 1;
9     }
10
11    public override void OnExit(MethodExecutionArgs args)
12    {
13        _x = _x == 1 ? 2 : 1;
14    }
15 }
```

Listing 6.3: The “empty” dummy aspect.

Stage III: JNBridge/Kieker #1

In this Stage, the `OperationExecutionAspect` of Kieker.NET is applied to the `MonitoredOperation()`, but for now, Kieker monitoring will be disabled.

A closer look at the `OnEntry()` and `OnExit()` methods of the aspect (see Section 5.2) reveals that in this configuration, the aspect makes exactly one call to the Java-based `MonitoringController` before and after execution of `MonitoredOperation()`, to check whether Kieker monitoring is enabled. If not, the aspect methods return immediately. The resulting overhead as depicted in Figure 6.2 is the sum of Δ_{IIIa} and Δ_{IIIb} . As mentioned before, we will not distinguish between “pure” JNBridge overhead (Δ_{IIIa}) and the resulting Java overhead (Δ_{IIIb}), because they clearly belong together.

As JNBridge supports multiple communication mechanisms, it may also be of interest to show how switching from the allegedly fast shared memory mode to TCP/binary affects the overall performance of Kieker.NET. For that, we include an additional Stage III_{TCP} in our analysis.

Stage IV: Kieker #2

Now with Kieker monitoring enabled, the full aspect will be executed on each call to `MonitoredOperation()` in this stage. Figure 6.3 shows the combined monitoring overhead as Δ_{IV} . There is no need for a finer grained classification in this stage, as we

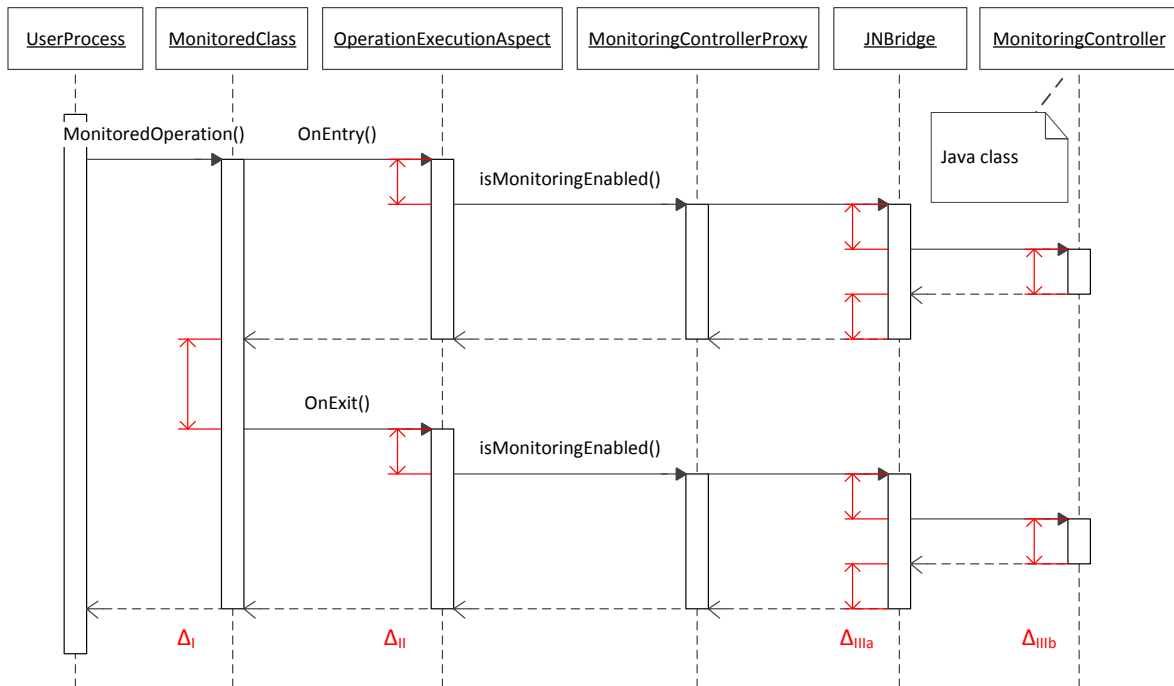


Figure 6.2.: Stage III. Quantification of monitoring overhead induced by JNBridge and bridged Java-side Kieker monitoring controller (with Kieker monitoring disabled).

incrementally got to this point and can subtract the previous results to get Δ_{IV} without Δ_{II} and Δ_{III} included if needed.

We set Kieker’s monitoring writer to the `DummyWriter` class to exclude all possible hard disk access delay from the observations of this stage. The `DummyWriter` receives all data just as, for example, the “regular” `AsyncFsWriter`, but does nothing further with it, so no overhead coming from storing the monitoring logs can be expected.

Stage V: HDD Access

All that is left for the final stage is enabling Kieker’s actual monitoring log writer, in our case the `AsyncFsWriter`, and see how flushing the monitored data to the Java-side’s file system impacts the performance of a monitored application. We omit another sequence diagram for this stage, as it would look exactly like the one of Stage IV shown in Figure 6.3 without going deeper into the class structure of Java-side Kieker.

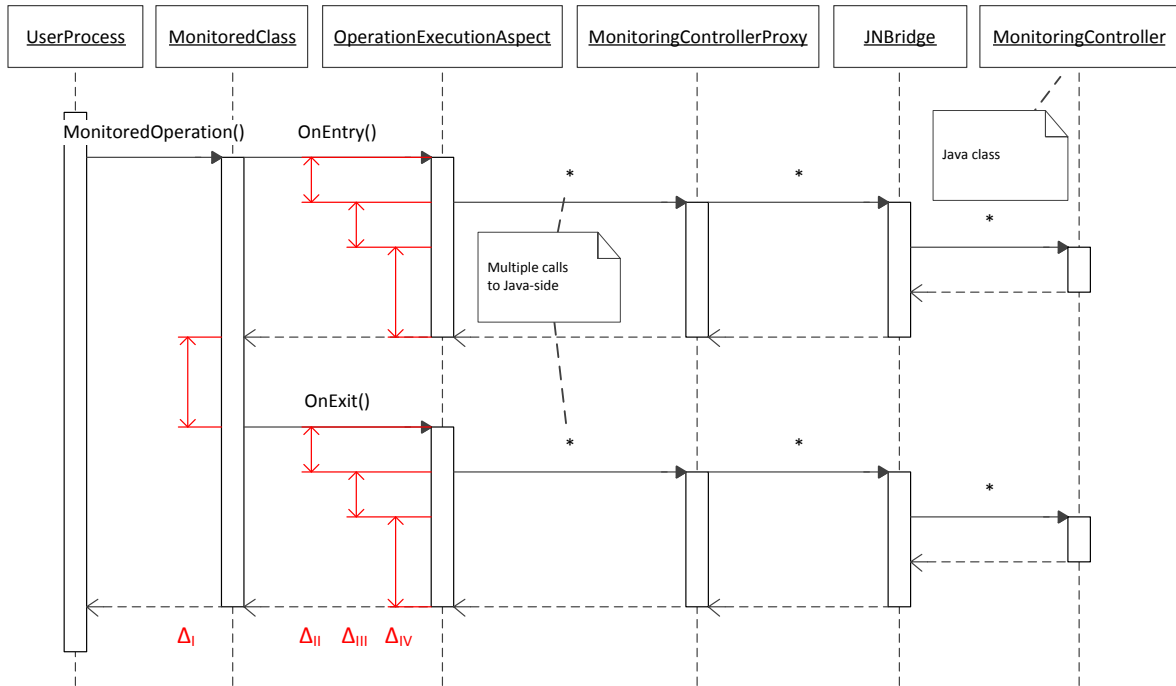


Figure 6.3.: Stage IV. Quantification of monitoring overhead induced by bridged Java-side Kieker monitoring classes.

6.3. Experiment Results

After executing one “initialization” and five “live” runs for each stage, we collected many time measurements concerning the overhead caused by Kieker.NET to a monitored application.

In the first stage, as expected, the execution time of `MonitoredOperation()` (Δ_I) is about 100 microseconds. Table 6.1 shows the results of the total of six runs with 100,000 (10,000 actually recorded) executions each for Stage I. The measured execution time is close to the value of 100 microseconds. We suspect the `Stopwatch` class to be accountable for the (admittedly very small) difference.

During the following benchmark executions for the Stages II to V, it can be observed that the results of the initialization and live runs do not significantly deviate from each other. Therefore, we combine the results of all stages in Table 6.2 and Figure 6.4. We choose the results of each stage’s third live run, and use those for the calculation of the Δ_{II} to Δ_V values, too.

Run	INIT	1ST	2ND	3RD	4TH	5TH
Mean	100.1547	100.1686	100.0719	100.1165	100.1086	100.0997
Median	100.0000	100.0000	100.0000	100.0000	100.0000	100.0000
Std. dev.	0.0026	0.0035	0.0015	0.0022	0.0021	0.0023

Table 6.1.: Stage I experiment results. Response times (in microseconds) of a single operation.

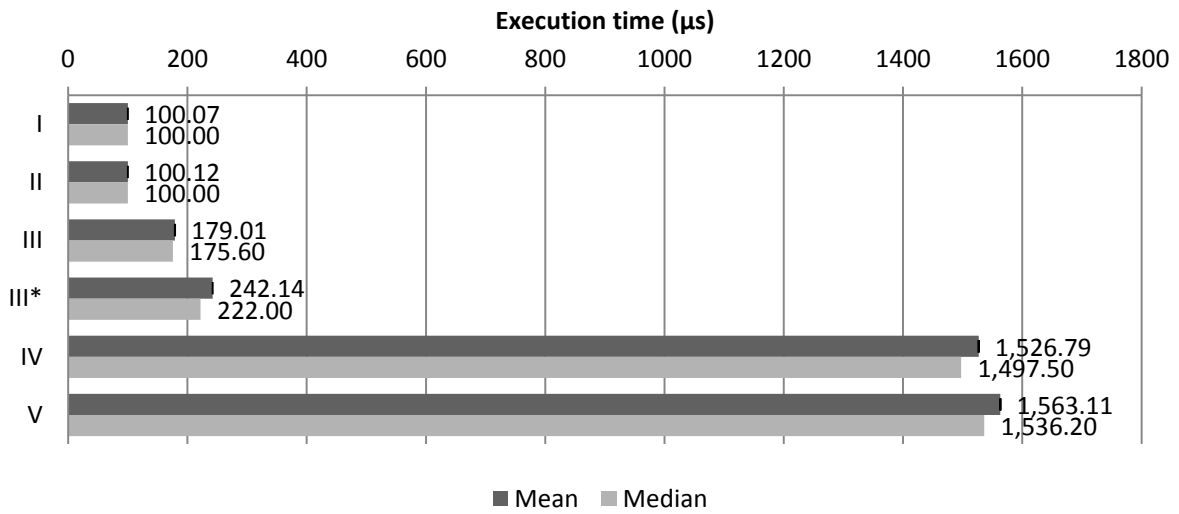


Figure 6.4.: Micro-benchmark overall performance analysis results (the row marked with (*) is Stage III_{TCP}).

STAGE	I	II	III	III _{TCP}	IV	V
MEAN	100.0719	100.1212	179.0138	242.1381	1,526.7927	1,563.1077
MEDIAN	100.0000	100.0000	175.6000	222.0000	1,497.5000	1,536.2000
STD. DEV.	0.0015	0.0019	0.0190	0.0328	0.5134	0.5237
RATIO	1.0000	1.0004	1.7888	2.4196	15.2569	15.6198

Table 6.2.: Micro-benchmark overall performance analysis results. Results taken from each stage's third run. All values in microseconds, ratio values are with respect to Stage I for each stage.

6. Overhead Evaluation

Δ	II	III	III _{TCP}	IV	V	Σ
MEAN	0.0493	78.8926	(142.0169)	1,347.7789	36.3150	1,463.0358
MEDIAN	0.0000	75.6000	(122.0000)	1,321.9000	38.7000	1,436.2000

Table 6.3.: Micro-benchmark overhead results. All values in microseconds.

As expected, the overhead induced by Postsharp (Δ_{II}) can be considered nonsignificant. As shown in Table 6.3, the application of the `DummyAspect` produced a mean execution time increase of only .0493 microseconds. This is most probably the execution time of the two lines of code in each of the aspect’s methods, alongside with the fact that the the actual execution of `MonitoredOperation` is transformed into a try-catch-finally block statement by Postsharp at compile-time, as shown in our Postsharp introduction in Section 5.1.1.

The performance impact of applying our `OperationExecutionAspect` has a different order of magnitude. For Δ_{III} (the sum of Δ_{IIIa} and Δ_{IIIb}), we calculate a mean value of 78.8926 microseconds when using JNBridge’s shared memory communication, and for TCP/binary, the calculated value of $\Delta_{III_{TCP}}$ even doubles that with 142.0169 additional microseconds when compared to the results of Stage II.

This significant increase in execution time is coming from the two calls the aspect makes (one in `OnEntry()` and one in `OnExit()`) to the Java-based `MonitoringController` instance through the JNBridge proxy class. On the Java-side, there is definitely no complex computation involved in checking if Kieker monitoring is enabled (which is not the case in Stage III).

As our `OperationExecutionAspect` makes *several* calls through JNBridge when Kieker monitoring is enabled, the results for Stage IV are not surprising after examining the previous Stage. In Stage IV, we calculated a mean value of 1347.7789 microseconds for Δ_{IV} , which is more than a thousand times the reference execution time of about 100 microseconds.

Compared to the previous stages, the hard disk access “measured” in Stage V is surprisingly not a major cause of overhead. This is probably due to the rather slow communication of JNBridge that gives the asynchronous file system writer of Kieker plenty of time to be started and executed without affecting the performance at all. The deviation of execution times of the Stages IV and V is also almost identical, which is

surprising as well, because we expected a greater variability when any hard disk access is involved.

In summary, the total overhead when monitoring a method's execution with the Post-sharp/JNBridge-based Kieker.NET is about 1.5 milliseconds.

7. Case Study

Aside from the performance impact that our monitoring framework has on the monitored application, testing and analyzing its robustness and also its practicality is equally important. For that, HSH Nordbank AG provided us their *Nordic Analytics* risk assessment system as a case study system. We employed Kieker.NET for comprehensive dynamic analysis of Nordic Analytics. The results are presented in this case study.

Section 7.1 begins with a short introduction to Nordic Analytics and explains how the system was used for the analysis. In Section 7.2, we focus on the challenges of finding the *right* instrumentation (or monitoring points) to find a adequate balance between completeness of monitoring data regarding the models to be extracted and the sheer size and complexity of this data.

In Sections 7.3 and 7.4, we present some of the data we obtained from Nordic Analytics during several monitoring sessions and instrumented evaluations of prepared risk assessment Jobs. This includes multiple extracted component and operation dependency graphs.

To substantiate our results of the overhead analysis with the micro-benchmark in Chapter 6, we also conducted the same experiment in a comprehensive performance (macro-)benchmark of Nordic Analytics, presented in Section 7.5.

7.1. Nordic Analytics

Nordic Analytics is a complex, C#-based function library for assessment and risk control of finance products, developed and actively employed by the German financial institute HSH Nordbank AG.

With Nordic Analytics, the user can create/modify and then evaluate a set of diverse (given) *Jobs* that contain, for example, valuation of interest rate cap and floor options, Swaptions, TARN, Swaps, or SABR-volatility, employing multiple different financial

Classes	1,657
Abstract Classes	136
Interfaces	218
Enums	234
Enums values	1,313
Value types	27
Properties	4,815
Objects	2,272
Methods	23,920
Total lines of code	390,481
Code lines	245,069
Comment lines	111,651
Empty lines	33,761

Table 7.1.: Code statistics of Nordic Analytics.

models such as Blacksholes, Monte Carlo, etc. This involves complex mathematical calculations and tree operations.

Nordic Analytics provides an Application Programming Interface (API) and can also be used either as a function library for Microsoft Excel, where the user can trigger an evaluation of Job data out of Excel, or as a stand-alone console application that has one or more Jobs formatted as text files, so called *Nordic Analytics Object* (NAO) files, as input. These text files are usually generated with Excel as well.

Nordic Analytics was first released in 2004 and was initially developed in cooperation with Prof. Dr. Wolfgang Schmidt from HfB - Frankfurt School of Finance & Management, and is now maintained and further developed by the Financial Engineering department of HSH Nordbank AG.

7.1.1. Code Statistics

HSH Nordbank AG provided us with some code statistics of the current version (2.0.52) of Nordic Analytics. Table 7.1 shows those that are relevant to our case study. The sheer amount of classes and methods is remarkable and—as we will see—introduces some problems when finding the “right” code instrumentation.

7.1.2. Nordic Analytics Monitoring Environment

As Kieker.NET probes can only be applied by re-building the code of an application (this is because of the post-compile-time aspect weaver of Postsharp), it was not possible to continuously monitor Nordic Analytics in its operational “live” environment.

Instead, the application we utilized for our case study is *Nordic Analytics Console Application*, which can be used to evaluate prepared Jobs. The Financial Engineering department of HSH Nordbank AG developed a set of test files (or *Nordic Analytics benchmark*), consisting of about 650 Jobs that cover all functionality of Nordic Analytics. To run this test, one can simply point Nordic Analytics Console Application to the folder that contains all these 650 test files and execute a full evaluation. Our contribution to the DynaMod project (in which HSH Nordbank AG participates) focuses on extraction of architectural models, so the monitoring of “artificial” (but probably more comprehensive) operation was not a drawback for our work at all.

During our first monitoring attempts with the newly developed Kieker.NET, we found that the sheer amount of calculations and repetitive model evaluations of the benchmark with 650 Jobs was probably a case of “too much information”. In addition to finding a convenient instrumentation (i.e., restricting the set of methods that are actually monitored to only the important ones), down-scaling of the benchmark was required.

With the help of Dr. Marcus Steinkamp of HSH Nordbank AG, we managed to narrow the test down to 29 Jobs that cover many of the important model evaluations and should lead to feasible architectural models when analyzed.

7.2. Code Instrumentation

One very challenging task was the process of finding the *right* code instrumentation (or monitoring points) for Nordic Analytics. As shown in Table 7.1, Nordic Analytics is a rather complex system, consisting of many components. The monitoring with the first instrumentation we came up with—after first discussions with the developers of the system—had to be cancelled after eight days of continuous monitoring and producing more than 40 GB of monitoring logs. At that moment, only four of the 650 Jobs had been completed. The resulting assembly component dependency graph we—out of curiosity—created with Kieker is shown in Figure 7.1. Creating an operation dependency graph or

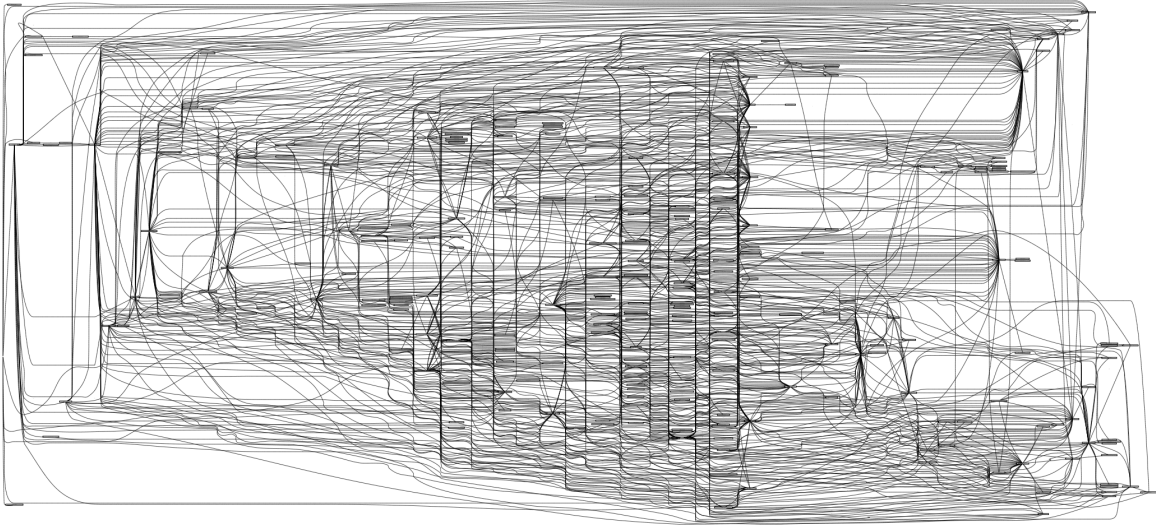


Figure 7.1.: Nordic Analytics component dependency graph with (close-to) full instrumentation, generated with Kieker.TraceAnalysis.

even a call tree was impossible with all tools we tested, because the complexity of the data caused out-of-memory terminations all the time.

With elaborated aspect multicast attributes and filters (see Listing 7.1), we limited our instrumentation of Nordic Analytics to 2,863 of the total 23,920 methods, primarily by trying to avoid instrumentation of accessor and mutator methods, as well as some “low-level” methods used excessively by some complex calculations.

Together with limiting the number of Jobs executed while monitoring to only 29, this rather sophisticated set of aspect multicast attributes and filters allowed us to gather much smaller monitoring logs, and therefore enabled us to finally extract more meaningful system models than the “negative example” shown in Figure 7.1.

```
1 // Includes
2
3 [assembly: OperationExecutionAspect(
4     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Analytics.*")]
5 [assembly: OperationExecutionAspect(
6     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Evaluation.*")]
7 [assembly: OperationExecutionAspect(
8     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Job.*")]
9 [assembly: OperationExecutionAspect(
10    AttributeTargetTypes = "NordicAnalyticsProject.Engine.ObjectFactory.
    cObjectPool")]
```

```

11 [assembly: OperationExecutionAspect(
12     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Result.*")]
13 [assembly: OperationExecutionAspect(
14     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Securities.*")]
15
16 // Excludes
17
18 [assembly: OperationExecutionAspect(
19     AttributePriority = 5,
20     AttributeExclude = true,
21     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Analytics.
    ObjectFunctions.*")]
22 [assembly: OperationExecutionAspect(
23     AttributePriority = 5,
24     AttributeExclude = true,
25     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Analytics.
    cAnalyticsConstructorLog")]
26 [assembly: OperationExecutionAspect(
27     AttributePriority = 5,
28     AttributeExclude = true,
29     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Evaluation.
    PricingModels.AmericanOptionPricingModel.*")]
30 [assembly: OperationExecutionAspect(
31     AttributePriority = 5,
32     AttributeExclude = true,
33     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Evaluation.
    PricingModels.BermudanStyleOptionPricingModel.
    NormalShortRateTreeModel.*")]
34 [assembly: OperationExecutionAspect(
35     AttributePriority = 5,
36     AttributeExclude = true,
37     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Securities.
    cCashFlowDescriptionTableFloatFlowDescription")]
38 [assembly: OperationExecutionAspect(
39     AttributePriority = 5,
40     AttributeExclude = true,
41     AttributeTargetTypes = "NordicAnalyticsProject.Engine.Securities.
    cCashFlowDateGrid")]
42
43 // Removing ctors, getters, setters, and NA-intrinsic tracking methods.
44
45 [assembly: OperationExecutionAspect(
46     AttributePriority = 10,
47     AttributeExclude = true,
48     AttributeTargetMembers = "regex:ctor|get_.*|set_.*|TrackUsage.*")]

```

Listing 7.1: Aspect multicast attributes and filters, leading to 2.836 instrumented methods of Nordic Analytics.

7.3. Dynamic Analysis of Nordic Analytics

After the optimization of the number and exact locations of monitoring points as described before, we were finally able to extract plenty of useful data from Nordic Analytics.

Of the 2,863 methods we instrumented with our `OperationExecutionAspect`, a total of 1,087 methods were actually called during the execution of Nordic Analytics Console Application evaluating the 29 Jobs of the benchmark we prepared.

The full monitored execution resulted in a total of 51 Kieker monitoring log files, with a combined file size of 357 MB. These monitoring log files contain execution data of 1,255,467 operations (i.e., method calls), including trace and timing information for each operation execution.

In Table 7.2, we present some information about each individual Job. This data was collected not by monitoring a sequential execution of all 29 Jobs, but by starting Nordic Analytics Console Application for each Job individually. This is why the sum of all operation executions in Table 7.2 is not exactly equal to 1,255,467 (i.e., the number we determined out of a single sequential execution of all 29 Jobs in a row as described above). Obviously, there was a (very small) “infrastructural overhead” included in the monitoring data when starting each Job individually. On the other hand, the results show that our instrumentation seems to “avoid” almost all methods that are not directly related to Job evaluation, which is good, as we can be sure that the monitoring points we chose cover only the core functionality of Nordic Analytics.

7.4. Architecture-Based Model Extraction

One of the main goals of our work is to provide the means for architecture-based model extraction of .NET applications. With our implementation of Kieker.NET, we were able to generate the monitoring log files that allowed comprehensive models of Nordic Analytics to be created with Kieker’s trace analysis tool `Kieker.TraceAnalysis`.

We start with a sequence diagram that has been created manually prior to our case study by a member of the Financial Engineering department of HSH Nordbank AG (Figure 7.2). It shows the expected internal component dependencies and control flow for an evaluation of a (rather simple) Swaption valuation with Nordic Analytics.

In Figure 7.3, we show an *assembly component dependency graph* created out of monitoring logs obtained by monitoring evaluation of Job 7, which is also a Swaption valuation

JOB	OPERATION EXECUTIONS	MONITORING LOG SIZE (MB))
1	6,212	1.46
2	8,139	1.94
3	254,000	60.70
4	62,292	16.20
5	3,973	0.94
6	1,797	0.42
7	1,802	0.43
8	11,573	2.79
9	5,669	1.35
10	4,301	1.00
11	46,105	12.10
12	84,842	26.70
13	4,498	1.05
14	10,598	3.59
15	9,332	3.00
16	95,785	32.70
17	3,283	0.80
18	230,099	60.60
19	56,966	16.70
20	3,130	0.74
21	3,113	0.73
22	64,053	20.80
23	139,168	45.80
24	44,574	15.70
25	29,930	7.03
26	907	0.29
27	49,540	17.30
28	14,448	3.42
29	5,422	1.27
Σ	1,255,551	357.56

Table 7.2.: Detailed monitored Job statistics.

much like the depicted execution in the manually created sequence diagram. The corresponding *assembly operation dependency graph* and *aggregated assembly call tree*, also created with Kieker.TraceAnalysis, are shown in Figures 7.4 and 7.5.

Figures 7.6 and 7.7 then present the extracted models of the evaluation of two more complex Jobs (5 and 29), involving Constant Maturity Swaps (CMS) and utilization of static replication models by Hagan.

Finally, Figure 7.8 shows the combined result of our refined instrumentation, the assembly component dependency diagram of the evaluation of all 29 Jobs of the benchmark. Opposed to the “negative example” shown in Figure 7.1, this graph actually could (and probably will) be a valuable source of information when used in architectural analysis of Nordic Analytics.

All following models except Figure 7.2 are vector graphics that allow seamless zooming when viewed in digital format.

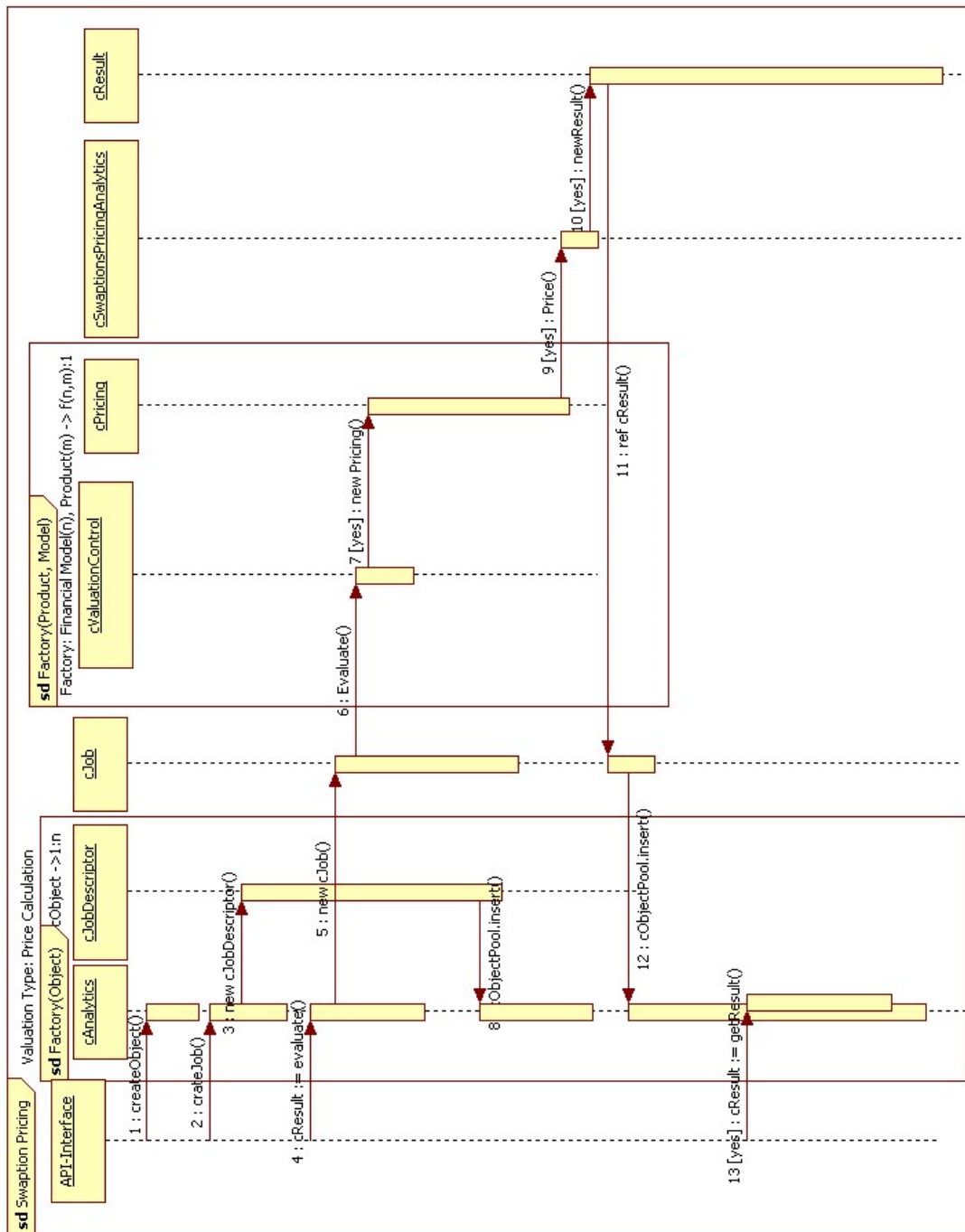


Figure 7.2.: Nordic Analytics components involved in a (rather simple) Swaption valuation. The Sequence diagram was manually created prior to our case study by a member of the Financial Engineering department of HSH Nordbank AG.

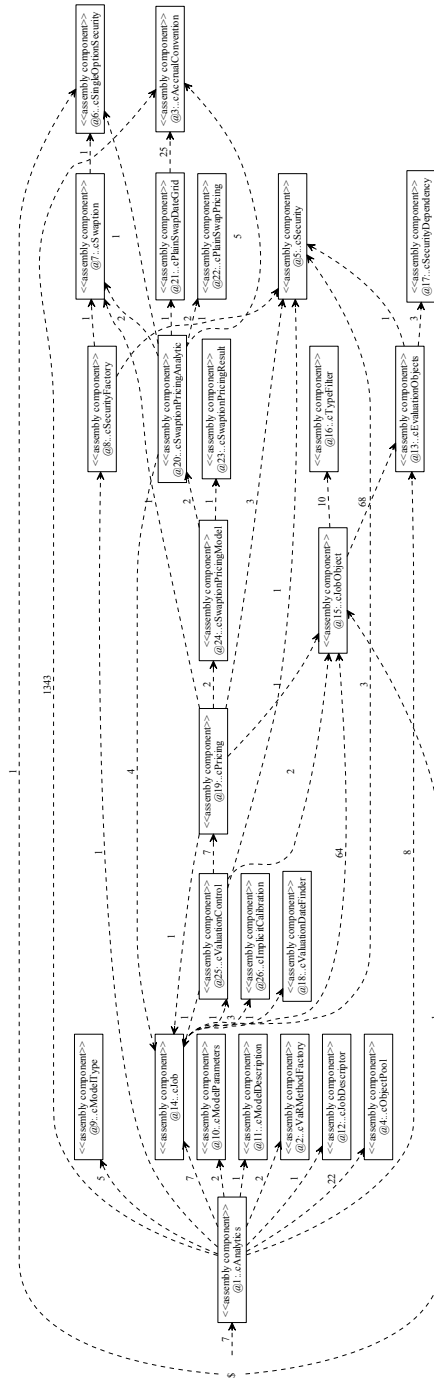


Figure 7.3.: Assembly component dependency graph of the monitored execution of Nordic Analytics evaluating a Swaption valuation using a Blacksholes model (Job 7).

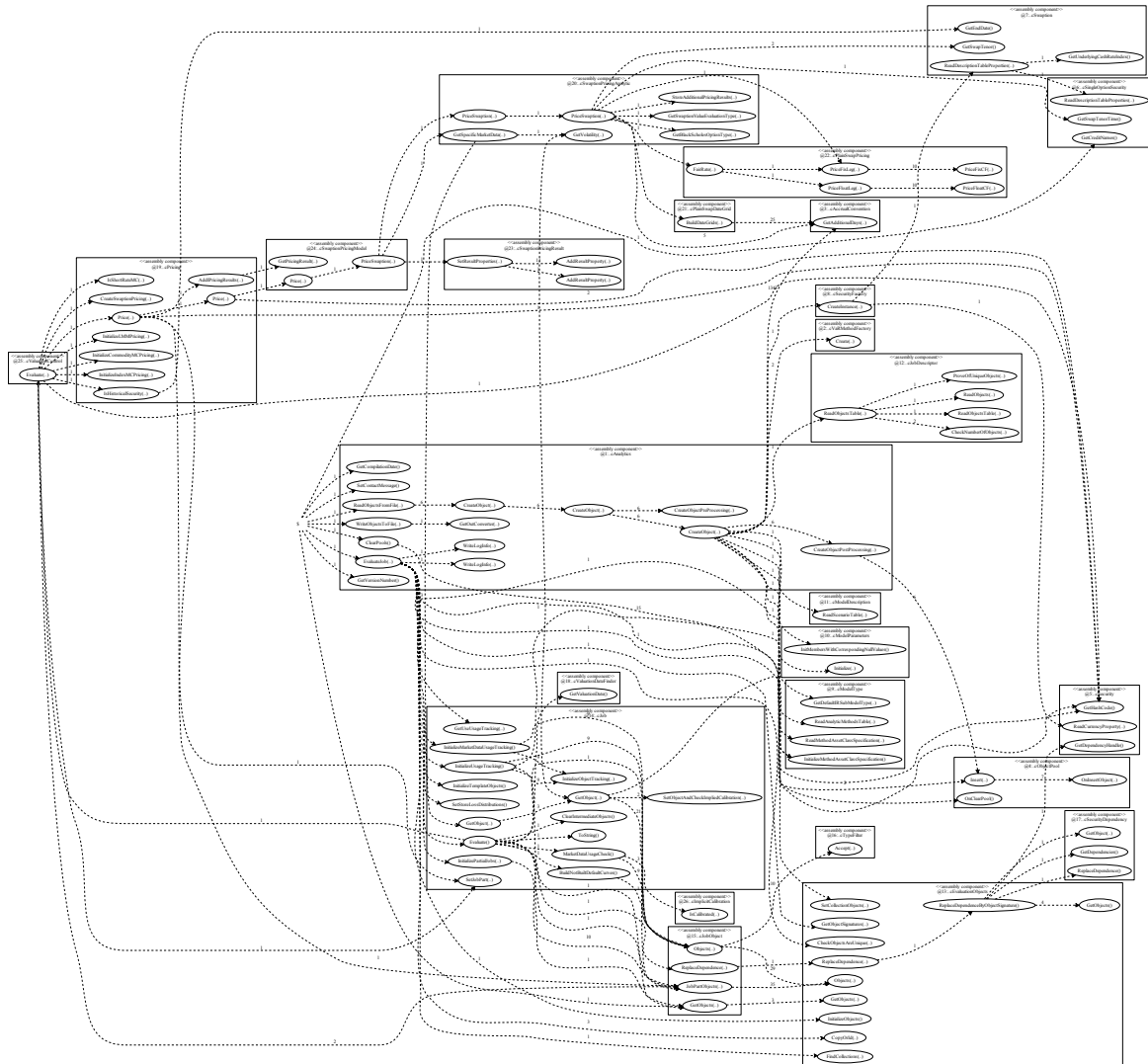


Figure 7.4.: Assembly operation dependency graph of a monitored execution of Nordic Analytics evaluating a Swaton valuation utilizing a Blacksholes model (Job 7).

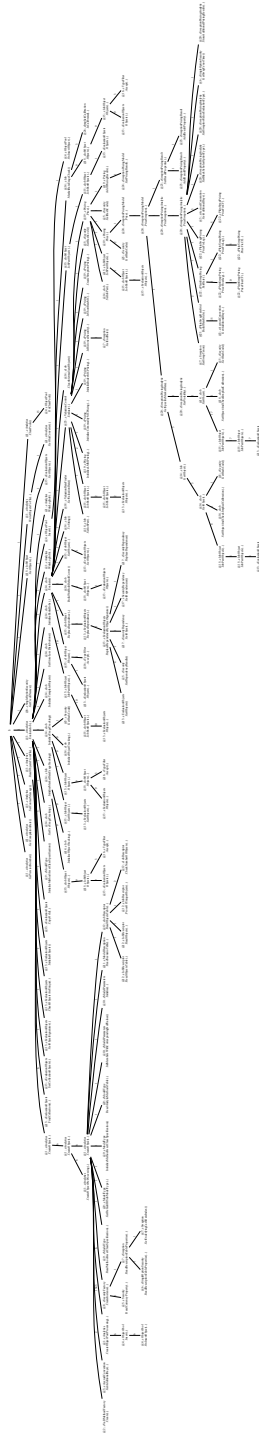


Figure 7.5.: Aggregated assembly call tree of a monitored execution of Nordic Analytics evaluating a Swation valuation utilizing a Blackscholes model (Job 7).

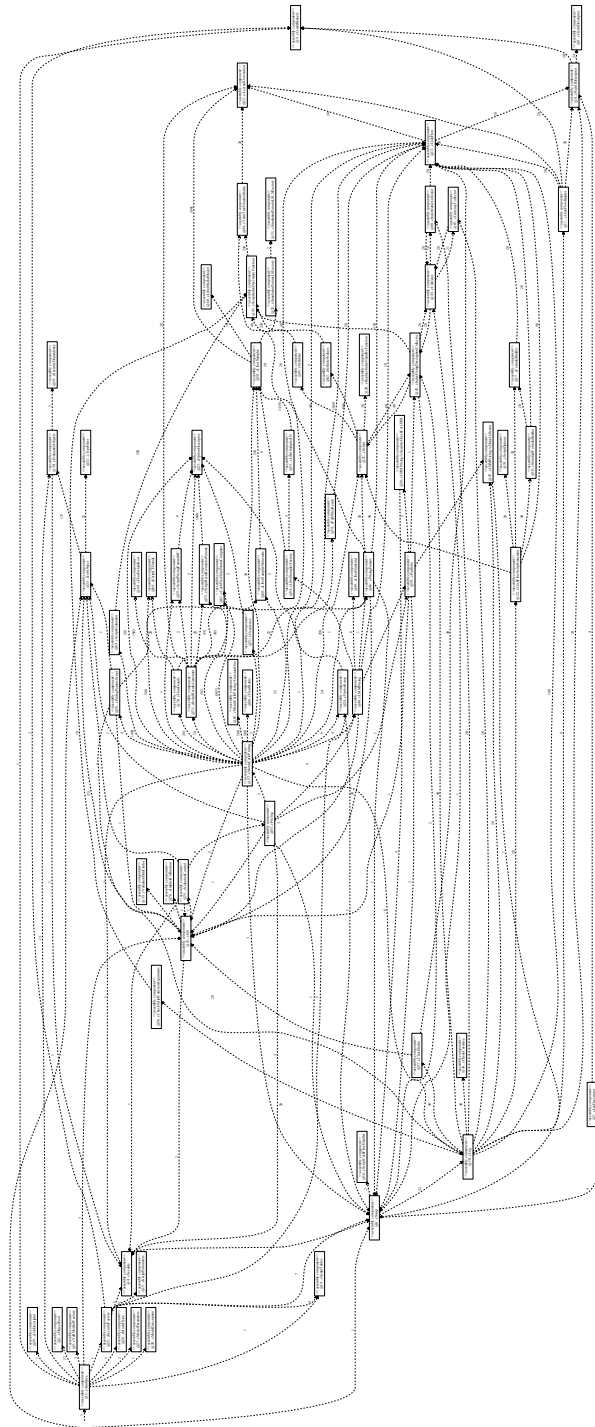


Figure 7.6.: Assembly component dependency graph of a monitored execution of Nordic Analytics evaluating a computationally intensive CMS utilizing a static replication model by Hagan (Job 29).

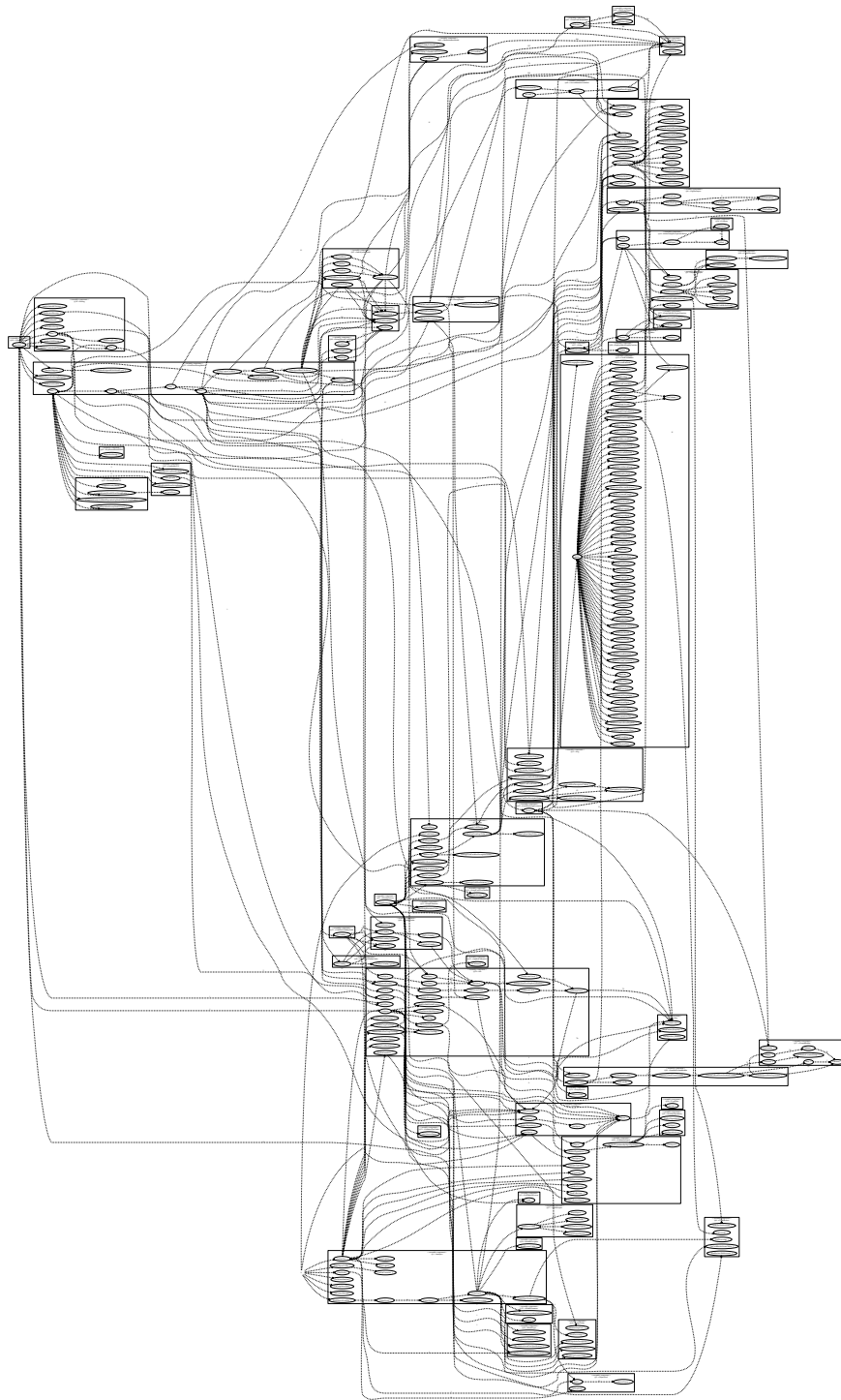


Figure 7.7.: Assembly operation dependency graph of a monitored execution of Nordic Analytics evaluating another complex CMS, again with a static replication model by Hagan (Job 5).

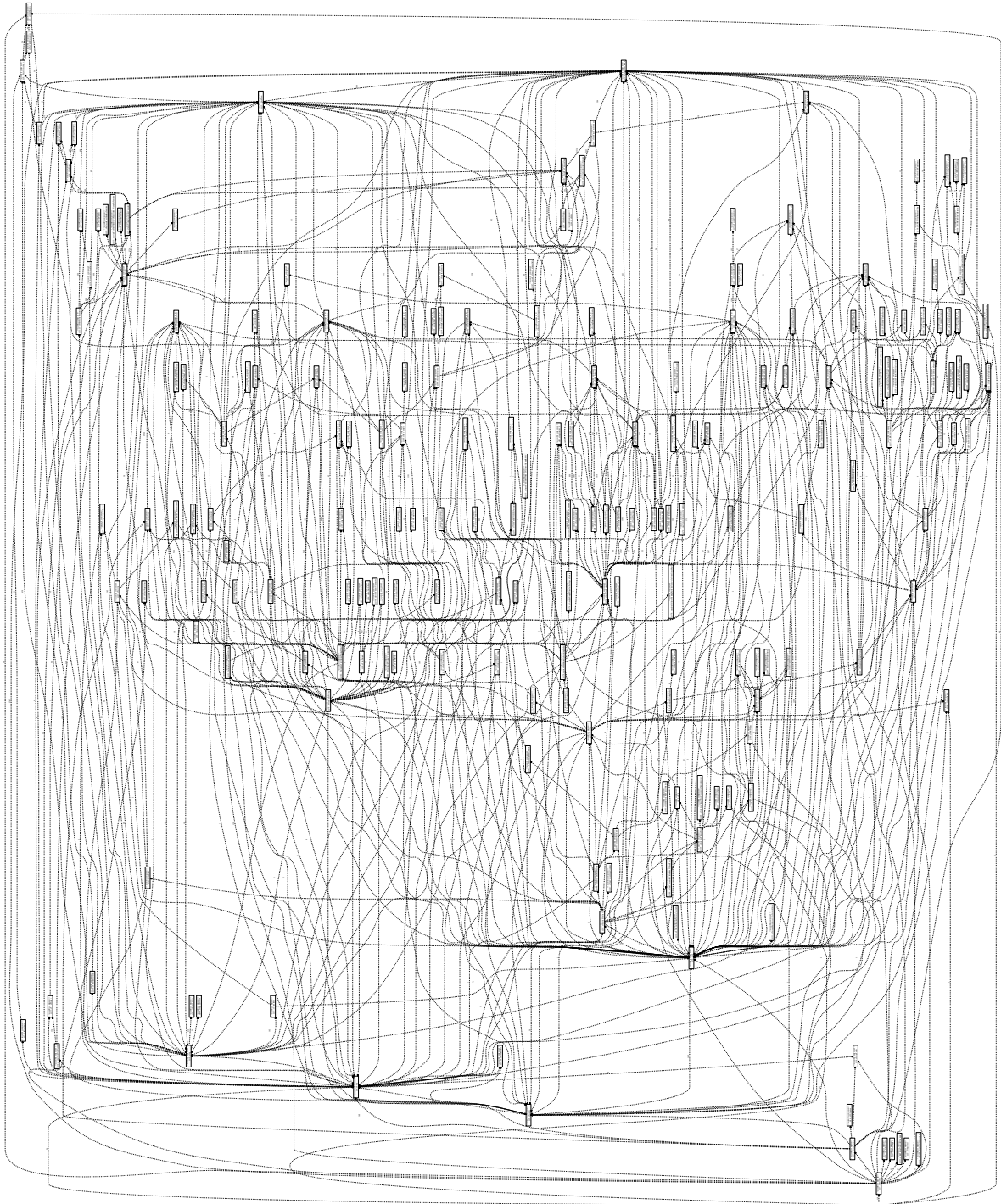


Figure 7.8.: Assembly component dependency graph of the monitored execution of Nordic Analytics evaluating the complete benchmark (Jobs 1 to 29).

7.5. Overhead Analysis

Aside from testing the model extraction capabilities of Kieker.NET, we also seized the opportunity to conduct a comprehensive macro-benchmark using the Nordic Analytics benchmark introduced in the previous sections. For that, we applied the same experiment design we used for the micro-benchmark to our overhead analysis with Nordic Analytics.

7.5.1. Experimental Setting

The test system was a Lenovo Thinkpad T61 with an Intel Core 2 Duo T7300 dual core processor with 2.0 GHz, 2 GB memory, and Windows XP Professional SP3 as operating system. The development environment where we could actually edit the source code of Nordic Analytics (which was not really necessary for monitoring with the final version of Kieker.NET, though) was Visual Studio 2005 with the .NET 2.0 framework. We used Java version 1.6.0_03-b05 (to host Kieker 1.4-dev-SNAPSHOT), JNBridge version 5.1, and Postsharp 2.1.3.3. The version of the analyzed Nordic Analytics build was 2.0.50.

7.5.2. Results

As in the micro-benchmark, the macro-benchmark consisted of multiple runs, one initialization run and five live runs each, of a sequential (batch) execution of the evaluation of the 29 jobs introduced above, where each individual execution time as well as the overall execution times were measured.

The chart in Figure 7.9 and Table 6.2 show the combined results of our analysis, whereas the data contained in Figure 7.10 gives detailed insight into all measured runs in all stages.

The benchmark results support our findings of Chapter 6. Postsharp (Stage II) has a clear (but probably negligible) performance impact, and the two calls through JNBridge made in Stage III (again, one in `OnEntry()` and one in `OnExit()`) take the benchmark's execution time to a higher order of magnitude.

As soon as the `OperationExecutionAspect` is fully applied (and Kieker monitoring is enabled), the performance of the monitored application again suffers almost exponentially.

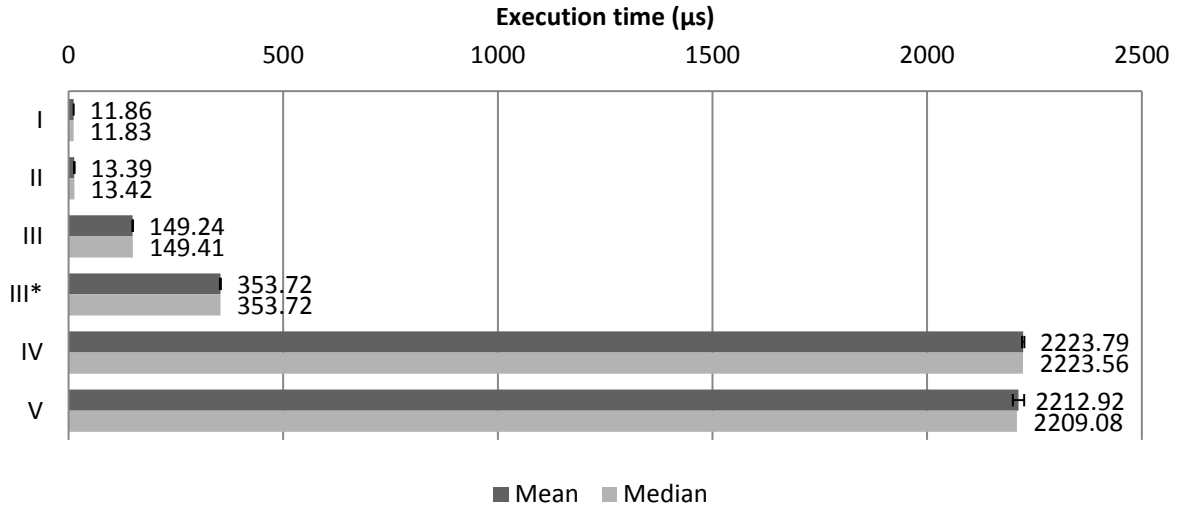


Figure 7.9.: Nordic Analytics macro-benchmark overall performance analysis results (the row marked with (*) is Stage III_{TCP}).

STAGE	I	II	III	III _{TCP}	IV	V
MEAN	11.86	13.39	149.24	353.72	2,223.79	2,212.92
MEDIAN	11.83	13.42	149.41	353.72	2,223.56	2,209.08
STD. DEV.	0.08	0.06	0.26	1.06	2.94	13.35
RATIO	1.00	1.13	12.58	29.83	187.52	186.60

Table 7.3.: Nordic Analytics macro-benchmark overall performance analysis results. All values in microseconds, ratio values are with respect to Stage I for all Stages.

Δ	II	III	III _{tcp}	IV	V	Σ
MEAN	1.53	135.85	(340.33)	2,074.55	-10.87	2,201.06
MEDIAN	1.59	135.99	(340.30)	2,074.15	14.48	2,226.21

Table 7.4.: Nordic Analytics macro-benchmark overhead results. All values in microseconds.

7. Case Study

Macro-Benchmark (Nordic Analytics)		Job Evaluation (in records)																															
Stage	Run	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29			
I	0	13.25	46.3%	6.13	0.52	0.03	0.30	0.30	0.02	0.03	0.02	0.25	0.02	0.33	0.39	0.14	0.00	0.14	0.11	0.08	0.16	0.52	0.12	0.03	0.05	0.53	0.27	0.27	0.14	0.00	0.05	0.08	
	1	11.83	44.0%	5.21	0.58	0.03	0.30	0.30	0.02	0.03	0.02	0.25	0.02	0.28	0.38	0.14	0.00	0.14	0.28	0.09	0.14	0.52	0.12	0.03	0.05	0.52	0.25	0.27	0.14	0.00	0.05	0.05	
	2	11.83	44.0%	5.21	0.53	0.03	0.30	0.30	0.02	0.03	0.02	0.23	0.02	0.28	0.38	0.14	0.02	0.14	0.30	0.08	0.16	0.52	0.12	0.03	0.05	0.52	0.25	0.27	0.14	0.00	0.05	0.03	
	3	11.75	44.9%	5.28	0.55	0.02	0.31	0.31	0.02	0.03	0.02	0.27	0.02	0.28	0.38	0.14	0.02	0.16	0.27	0.09	0.16	0.53	0.12	0.02	0.05	0.53	0.25	0.27	0.14	0.00	0.03	0.03	
	4	11.95	43.4%	5.19	0.55	0.03	0.31	0.30	0.03	0.02	0.27	0.00	0.30	0.36	0.14	0.02	0.14	0.02	0.14	0.30	0.08	0.16	0.53	0.12	0.02	0.05	0.52	0.27	0.14	0.00	0.03	0.03	
II	0	11.94	43.2%	5.16	0.53	0.03	0.30	0.30	0.02	0.03	0.02	0.27	0.00	0.28	0.36	0.14	0.00	0.14	0.33	0.08	0.16	0.52	0.12	0.02	0.05	0.52	0.27	0.12	0.00	0.03	0.05		
	1	11.86	43.9%	5.21	0.55	0.03	0.30	0.30	0.02	0.03	0.02	0.27	0.01	0.28	0.36	0.14	0.01	0.14	0.30	0.08	0.16	0.52	0.12	0.03	0.05	0.52	0.26	0.27	0.12	0.00	0.04	0.04	
	2	11.83	44.0%	5.21	0.55	0.03	0.30	0.30	0.02	0.03	0.02	0.27	0.01	0.28	0.38	0.14	0.02	0.14	0.30	0.08	0.16	0.52	0.12	0.03	0.05	0.52	0.25	0.27	0.14	0.00	0.03	0.05	
	3	13.38	46.7%	6.27	0.72	0.05	0.39	0.44	0.02	0.05	0.02	0.27	0.28	0.02	0.36	0.39	0.19	0.00	0.17	0.33	0.14	0.23	0.56	0.14	0.05	0.08	0.55	0.23	0.31	0.17	0.00	0.05	0.06
	4	13.45	46.8%	6.29	0.70	0.05	0.38	0.45	0.02	0.06	0.02	0.27	0.02	0.34	0.39	0.19	0.02	0.17	0.30	0.16	0.23	0.58	0.14	0.03	0.08	0.55	0.25	0.31	0.16	0.00	0.05	0.08	
III	0	13.44	47.2%	6.34	0.75	0.05	0.38	0.47	0.02	0.05	0.00	0.27	0.27	0.02	0.36	0.39	0.17	0.00	0.17	0.36	0.17	0.23	0.56	0.14	0.03	0.08	0.56	0.25	0.31	0.17	0.00	0.05	0.06
	1	13.28	46.8%	6.22	0.69	0.05	0.38	0.45	0.02	0.06	0.00	0.25	0.28	0.00	0.36	0.39	0.17	0.02	0.16	0.34	0.17	0.22	0.56	0.14	0.03	0.09	0.56	0.25	0.31	0.16	0.00	0.05	0.06
	2	13.39	46.9%	6.28	0.71	0.05	0.38	0.46	0.02	0.05	0.00	0.27	0.28	0.02	0.36	0.39	0.18	0.01	0.17	0.34	0.16	0.23	0.56	0.14	0.03	0.08	0.56	0.25	0.31	0.16	0.00	0.05	0.06
	3	13.42	46.8%	6.27	0.70	0.05	0.38	0.45	0.02	0.05	0.00	0.27	0.28	0.02	0.36	0.39	0.19	0.02	0.17	0.34	0.16	0.23	0.56	0.14	0.03	0.08	0.56	0.25	0.31	0.16	0.00	0.05	0.06
	4	13.45	46.8%	6.29	0.73	0.05	0.38	0.46	0.02	0.05	0.00	0.27	0.28	0.02	0.36	0.39	0.19	0.02	0.17	0.34	0.16	0.23	0.56	0.14	0.03	0.08	0.56	0.25	0.31	0.16	0.00	0.05	0.06
IV	0	14.39	49.3%	6.32	0.75	0.03	0.38	0.48	0.02	0.05	0.02	0.25	0.27	0.01	0.36	0.38	0.19	0.00	0.17	0.36	0.14	0.23	0.56	0.14	0.03	0.09	0.56	0.25	0.31	0.17	0.00	0.05	0.06
	1	13.38	46.9%	6.27	0.67	0.05	0.38	0.48	0.02	0.05	0.00	0.27	0.28	0.02	0.36	0.39	0.19	0.02	0.17	0.30	0.16	0.23	0.58	0.14	0.03	0.08	0.55	0.25	0.31	0.16	0.00	0.05	0.06
	2	13.45	46.8%	6.29	0.70	0.05	0.38	0.45	0.02	0.06	0.00	0.27	0.27	0.02	0.34	0.39	0.19	0.02	0.17	0.36	0.16	0.22	0.56	0.14	0.03	0.08	0.55	0.25	0.31	0.16	0.00	0.05	0.08
	3	13.44	47.2%	6.34	0.75	0.05	0.38	0.47	0.02	0.05	0.00	0.27	0.27	0.02	0.36	0.39	0.17	0.00	0.17	0.36	0.17	0.23	0.56	0.14	0.03	0.08	0.56	0.25	0.31	0.17	0.00	0.05	0.06
	4	13.28	46.8%	6.22	0.69	0.05	0.38	0.45	0.02	0.06	0.00	0.25	0.28	0.00	0.36	0.39	0.17	0.02	0.16	0.34	0.17	0.22	0.56	0.14	0.03	0.09	0.56	0.25	0.31	0.16	0.00	0.05	0.06
V	0	14.92	57.9%	8.65	1.13	0.66	26.51	6.92	0.22	0.09	0.05	1.21	0.65	0.24	4.99	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.49	0.17	0.06	1.21	5.08	3.00	3.03	0.25	0.07	1.34	0.18
	1	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	2	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	3	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	4	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
VI	0	14.91	57.9%	8.65	1.13	0.66	26.51	6.92	0.22	0.09	0.05	1.21	0.65	0.24	4.99	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.49	0.17	0.06	1.21	5.08	3.00	3.03	0.25	0.07	1.34	0.18
	1	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	2	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	3	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	4	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
VII	0	14.91	57.9%	8.65	1.13	0.66	26.51	6.92	0.22	0.09	0.05	1.21	0.65	0.24	4.99	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.49	0.17	0.06	1.21	5.08	3.00	3.03	0.25	0.07	1.34	0.18
	1	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	2	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	3	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	4	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
VIII	0	14.91	57.9%	8.65	1.13	0.66	26.51	6.92	0.22	0.09	0.05	1.21	0.65	0.24	4.99	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.49	0.17	0.06	1.21	5.08	3.00	3.03	0.25	0.07	1.34	0.18
	1	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	2	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	3	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.66	0.25	5.00	3.55	0.38	0.06	0.44	1.66	0.34	22.14	3.48	0.17	0.06	1.22	5.08	3.00	3.03	0.25	0.08	1.34	0.19
	4	14.91	57.9%	8.65	1.12	0.66	26.48	6.91	0.22	0.09	0.05	1.22	0.6																				

One interesting detail is the difference between the standard deviation of Stages IV and V. It is obvious, that—even though Mean and Median do not significantly differ from each other—the hard disk access of Stage V has an influence on the deviation of execution times.

The per-Job results of Figure 7.10 show that some Jobs have a relatively small impact on the execution time increase from one stage to another, whereas other Jobs have a much more significant part in it. Jobs 18 and 24 are an example for that, particularly noticeable by the difference of their ratio values. Job 18 has a huge performance impact ratio of $R_{IV:I} = 2,325$, whereas Job 24 affects the execution time of Stage IV compared to Stage I only with a ratio of $R_{IV:I} = 4.08$. This is somewhat related to the sheer amount of operations each Job “produces”. For our example, Job 18 consist of 230,099 operation executions (i.e., monitored method calls) and Job 24 of 44,574. But this alone is not the reason for the difference, as other Jobs have far less operation executions, but a greater ratio. A deeper analysis of the nature of those Jobs would be required to understand the exact reasons for this behavior.

8. Conclusion

With the results presented in this thesis it is now possible to conduct comprehensive monitoring and dynamic analysis of applications developed with and targeting the Microsoft .NET Framework by employing the Java-based Kieker framework [Software Engineering Group, University of Kiel 2011].

In Section 8.1 we summarize our work by outlining the contents of the individual chapters of this thesis. The overall results are discussed in Section 8.2, including some final thoughts about the applicability of the presented techniques in real-world software environments.

8.1. Summary

Extending Kieker with its dynamic analysis capabilities to target the Microsoft .NET framework required us to go through several development stages, including some initial technological decisions.

.NET Integration of Kieker

After giving a brief introduction to some basic terms and foundations to the field of our work in Chapter 2, we focused on finding answers to the question of how the integration of Kieker into the .NET framework can be accomplished (Chapter 3). We presented three possible approaches and subsequently, based on a short evaluation, our solution—an implementation based on JNBridge [JNBridge LLC. 2011a].

Dynamic Analysis with Kieker.NET

The outcome of that first development stage was a .NET-compatible function library (KiekerProxy.dll) that enabled us to directly use and call the exposed Kieker Java classes and their methods, respectively, in .NET. Running examples using a sample application throughout Chapter 4 proved that our solution was already capable of allowing basic code instrumentation and dynamic analysis after that first stage.

AOP-Based Monitoring with Kieker.NET

The next step was finding a way to employ techniques based on the ideas of aspect-oriented programming to the newly developed *Kieker.NET* framework. In Chapter 5 we provide a detailed description on how we implemented non-intrusive instrumentation techniques using Postsharp [SharpCrafters s.r.o. 2011a]. With the post-compile-time aspect weaver of Postsharp, our solution supports different options for weaving the cross-cutting concern (i.e., the monitoring probes) into a software system.

Overhead Evaluation

A comprehensive evaluation of our solution was the closing development stage. Chapter 6 provided an overhead evaluation in which we introduced a micro-benchmark to find out to what extent monitoring with Kieker.NET impacts the performance of a system under observation.

Case Study: Nordic Analytics

Finally, we applied Kieker.NET to our case study system *Nordic Analytics* by HSH Nordbank AG and presented our findings in Chapter 7. This helped proving the practicality and robustness of our solution. We also conducted the same overhead evaluation we developed for the micro-benchmark mentioned above with a Nordic Analytics test suite that was also provided to us by HSH Nordbank AG.

During first examinations of the Kieker.NET-generated component and operation dependency models presented in Chapter 7 in cooperation with some of the developers of Nordic Analytics, we already identified spots where—according the HSH team—far too many calls to some specific components were obviously made and recorded by our monitoring framework. We take this as a proof-of-concept and are confident that the extended models we extracted, and which we will provide to the HSH team shortly, will contribute to further increasing the maturity, robustness, and performance of Nordic Analytics.

Thus, the first, second, and fourth goal of our work can be considered accomplished. For accomplishing the first goal, we provided technical instrumentation of .NET application, as presented in Chapters 3 to 5. Towards reaching the second goal, we extracted several system models out of monitoring data obtained by dynamic analysis of both our bookstore sample application as well as the case study system Nordic Analytics. With that, we also successfully evaluated our methodologies and therefore accomplished the fourth goal.

Test Generation

Because the case study system Nordic Analytics already had an elaborated and complete test suite, we shifted our focus in the later stages of our work towards comprehensive overhead analysis and model extraction presented in Chapters 6 and 7. Therefore, we could not reach our third goal, and did not provide any new methodologies for (automated) test generation.

8.2. Discussion

Technological Choices

In hindsight, the employment of JNBridge might not have been the ideal solution for the integration of Kieker into .NET. JNBridge is a mature .NET/Java intercommunication utility that can easily be used and held up well to most of the promises that are made on its product website and the comprehensive users' guide [JNBridge LLC. 2011b]. But our overhead evaluation showed, with both the micro- and the macro-benchmark having similar results, that the performance impact of our `OperationExecutionAspect` is quite significant. This is primarily caused by the calls to the Java-side through JNBridge. Comparing our overhead analysis results with the findings made by van Hoorn et al. [2009] makes it clear that a Java-side monitoring controller is definitely not the cause for such overhead. Their reported overhead is only about 2.5 microseconds per monitored operation, opposed to the 1.5 milliseconds we determined for Kieker.NET.

The employment of Postsharp was convenient and no real drawbacks can be mentioned in that context. In our opinion, the only downside of Postsharp is that it does not support load-time aspect weaving, but that is probably more a fundamental characteristic of the approach taken by Postsharp than a yet unimplemented feature.

Finding the “Right” Monitoring Points

For most instrumented methods, applying a rather complex aspect like the `OperationExecutionAspect` increases the amount of code to be executed by large. When instrumenting, for example, a simple “getter”, it suddenly becomes a multi-line operation with several components involved. In our case, even proxy methods may be called via the TCP/binary mechanism of JNBridge when configured this way. This fact points to the main struggle we had when trying to apply our lab-tested methodologies to the complex case study system Nordic Analytics. It literally took weeks of suffering from

disk space shortages and repeatedly crashing tools for graph generation caused by the massive overflow of monitored operation execution data before we managed to elaborate a decent instrumentation.

Dynamic analysis should be taken into account in the earliest development stages of any software system. This could be done by defining layers for monitoring points, so that it is easier to differentiate between pure calculations and architecture- and component-wide calls. Monitoring of both categories most certainly has its use, but probably often not at the same time. During the refining process of the set of monitoring points for our case study system Nordic Analytics, it was obvious that a clearer line between pure calculations and “infrastructure” would have made dynamic analysis of this system much easier.

8.3. Future Work

Future work on the topics of this thesis could consist of the following points.

8.3.1. Kieker.NET

- Integrating Kieker.NET into the Kieker distribution and build process.
- Combining JNBridge configuration files with Kieker’s properties file by adding some new entries to the latter. Likewise, configuration of the Java-side could be done programmatically. For this, a small Java client could be implemented that replaces `start-kieker-jvm.bat` and starts the Java-side with all needed classes (JNBridge, Kieker) on the Java classpath.
- Adding support for other probes, records, etc. to Kieker.NET. All Java types can be bridged with JNBridge.
- Finding a way to support load-time aspect weaving in .NET as AspectJ does in Java.
- Adding support for other .NET technologies like the Spring framework.
- Re-implementation of Records and MonitoringControllers would probably make sense due to large overhead caused by JNBridge communication.

- Re-ordering of calls to the Java-side in `OnEntry()` and `OnExit()` could also minimize communication overhead. The same holds for evaluating alternative ways to measure a methods execution time, as Kieker.NET makes a Java call for that in its current state.

8.3.2. Model Extraction

It was originally intended to add support for other model types to Kieker's analysis component. Envisioned for this thesis were:

- *Use Case Maps*.
- *UML Timing Diagrams*.
- *UML Sequence Diagrams* annotated with timing information.

8.3.3. Test Generation

As we made no further research into this topic, hardly any advice for future work can be given.

A. Kieker.NET Visual Studio Solution

A.1. The Bookstore Projects

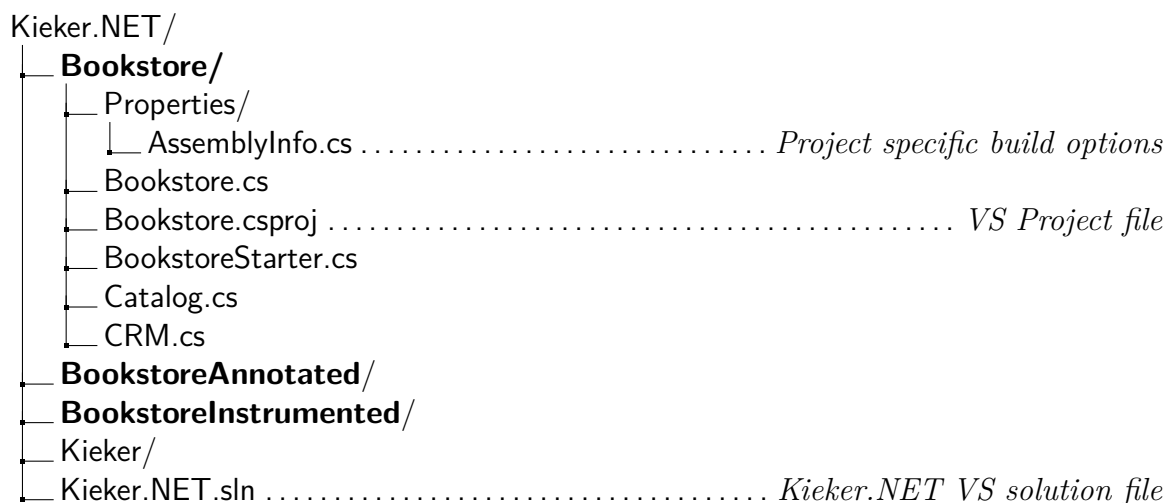


Figure A.1.: Bookstore Visual Studio project structure.

A.1.1. Project Configuration

All Bookstore sample applications are plain .NET 4.0 C# console applications with no further customization.

A.1.2. References

There are no references other than the System library of .NET needed for the uninstrumented Bookstore sample application. For manual instrumentation (BookstoreInstrumented), we need our generated KiekerProxy.dll as well as JNBridge's JNBShare.dll and JNBSharedMem.dll to access the Kieker Java types. When using Postsharp aspects

in our annotated Bookstore example (BookstoreAnnotated), we also need a reference to the Postsharp.dll library.

A.1.3. Classes

See the attached CD-ROM for the Visual Studio projects containing the different .NET-based Bookstore sample application classes.

A.2. The Kieker Project

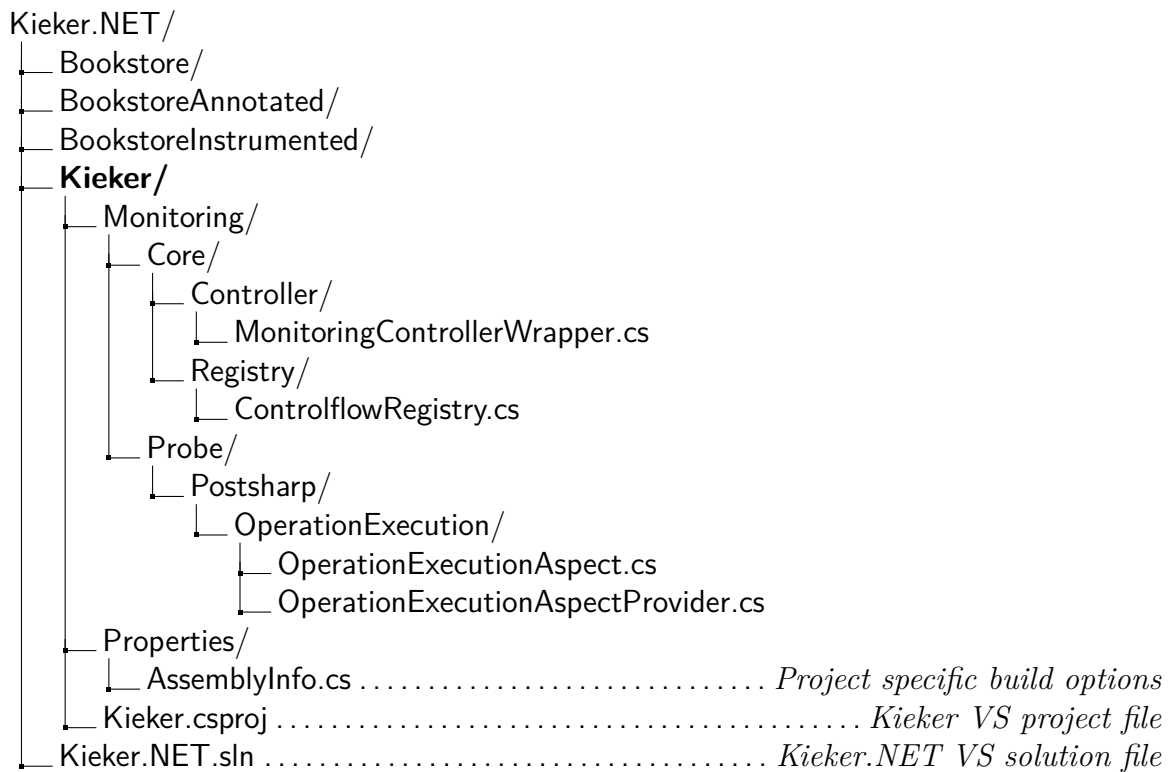


Figure A.2.: Visual Studio project structure of the .NET-based Kieker.

A.2.1. Project Configuration

In Figure A.3 we present the basic configuration for our Kieker.NET project. On the left side are the project related settings, and the *Solution Explorer* on the right side

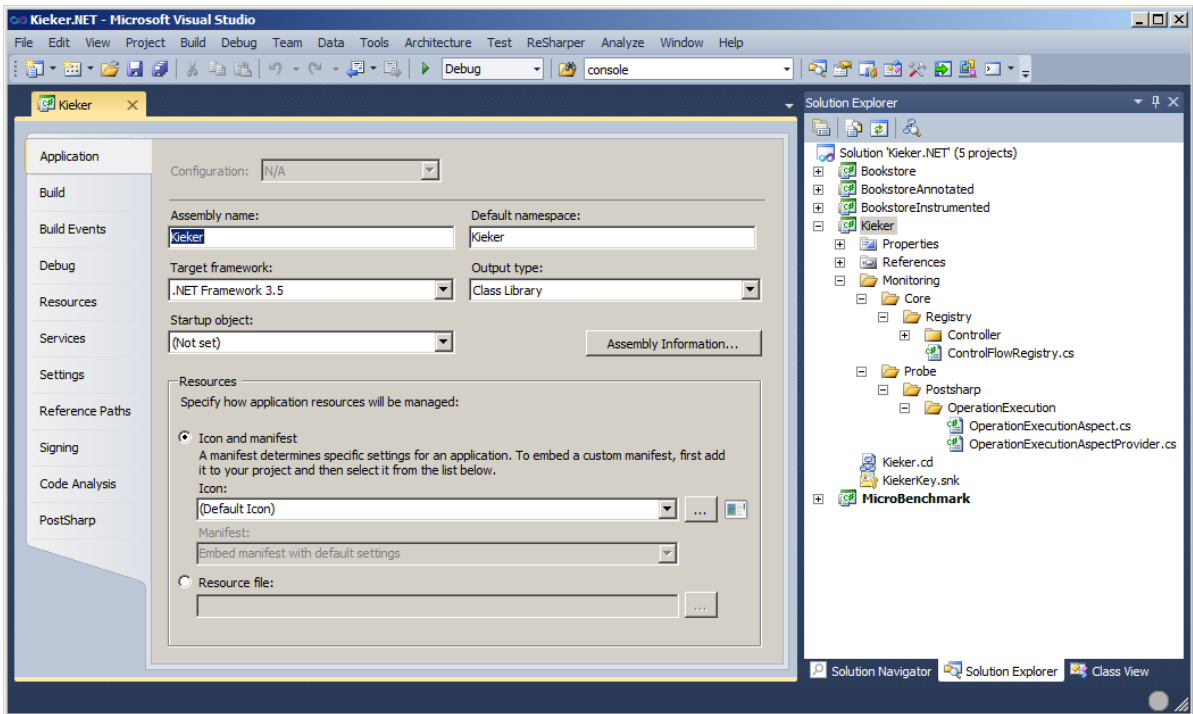


Figure A.3.: Kieker.NET Visual Studio configuration.

shows our project references (i.e., third party libraries, see Section A.2.2) and the classes Kieker.NET consists of (Section A.2.3).

Kieker.NET targets .NET Framework 3.5, as our major case study system, Nordic Analytics by HSH Nordbank AG, is a .NET 2.0 application. These .NET versions are compatible.

The desired output type of our .NET project is just a class library, as we do not need an application. All other settings are default for our Kieker.NET Visual Studio project.

A.2.2. References

JNBShare.dll is the runtime library of JNBridge that needs to be referenced in order to allow the referencing project to call Java methods through the proxy class. This library is needed for TCP/Binary or HTTP/SOAP based communication.

JNBSharedMem.dll is needed if JNBridge is configured to communicate with the Java-side via shared memory, i.e., the .NET side and Java side are running on the same machine.

KiekerProxy.dll is the proxy library generated with the JNBProxy proxy generation tool. This library basically contains all exposed Java methods and therefore provides those to the .NET side to be called by any .NET project that references this library.

Postsharp.dll comes as part of the Postsharp distribution and is needed for creating aspects out of attributes as well as compile-time aspect weaving.

A.2.3. Classes

All classes shown here can also be found in the Visual Studio project and solution folders on the attached CD-ROM.

OperationExecutionAspect

```
1 using System;
2 using System.Reflection;
3 using System.Text;
4 using kieker.common.record; // Bridged Java class.
5 using kieker.monitoring.core.controller; // Bridged Java class.
6 using Kieker.Monitoring.Core.Controller;
7 using Kieker.Monitoring.Core.Registry;
8 using PostSharp.Aspects;
9
10 namespace Kieker.Monitoring.Probe.Postsharp.OperationExecution
11 {
12     [Serializable]
13     public class OperationExecutionAspect : OnMethodBoundaryAspect
14     {
15         // According to the Postsharp documentation, fields that are only
16         // needed at runtime, and are unknown at compile-time, should be
17         // marked with the [NonSerialized] attribute.
18         [NonSerialized]
19         private static IMonitoringController _ctrlInst;
20         [NonSerialized]
21         private static ControlFlowRegistry _cfRegistry;
22
23         private string _componentName;
24         private string _method;
25
26         /// <summary>
27         /// Compile-time initialization of component name, method name,
28         /// and parameter types of the method to which the current aspect
29         /// instance has been applied.
```

```
30     /// </summary>
31     /// <remarks>
32     /// This improves runtime performance, as it avoids use of
33     /// <c>System.Reflection</c> as well as string building at
34     /// runtime.
35     /// </remarks>
36     /// <param name="method">
37     /// Method to which the current aspect instance has been applied.
38     /// </param>
39     public override void CompileTimeInitialize(MethodBase method,
40         AspectInfo aspectInfo)
41     {
42         _componentName = FormatType(method.DeclaringType.FullName);
43         _method = FormatMethodName(method.Name) + FormatParameters(
44             method.GetParameters());
45     }
46     /// <summary>
47     /// Initializes the current aspect at run time.
48     /// </summary>
49     /// <param name="method">
50     /// Method to which the current aspect is applied.
51     /// </param>
52     public override void RuntimeInitialize(MethodBase method)
53     {
54         _ctrlInst = MonitoringControllerWrapper.MonitoringController;
55         _cfRegistry = ControlFlowRegistry.Instance;
56     }
57     /// <summary>
58     /// Method executed <b>before</b> the body of methods to which
59     /// this aspect is applied.
60     /// </summary>
61     /// <param name="args">
62     /// Event arguments specifying which method is being executed,
63     /// which are its arguments, etc.
64     /// </param>
65     public override void OnEntry(MethodExecutionArgs args)
66     {
67         if (!_ctrlInst.isMonitoringEnabled())
68         {
69             return;
70         }
71         OperationExecutionRecord execData = InitExecutionData();
72         int eoi = 0; // execution order index
73         int ess = 0; // execution stack size
74         if (execData.isEntryPoint)
75         {
76             _cfRegistry.StoreThreadLocalEoi(0);
77             _cfRegistry.StoreThreadLocalEss(1);

```

```

78     }
79     else
80     {
81         eoi = _cfRegistry.IncrementAndRecallThreadLocalEoi();
82         ess = _cfRegistry.RecallAndIncrementThreadLocalEss();
83     }
84     if ((eoi == -1) || (ess == -1))
85     {
86         Console.WriteLine("eoi and/or ess have invalid values:" +
87             " eoi == " + eoi + " ess == " + ess);
88         _ctrlInst.terminateMonitoring();
89     }
90     execData.eoi = eoi;
91     execData.ess = ess;
92     // Time when monitored method begins execution.
93     execData.tin = _ctrlInst.getTimeSource().getTime();
94     // Store execData for use in OnExit().
95     args.MethodExecutionTag = execData;
96 }
97
98 /// <summary>
99 /// Method executed <b>after</b> the body of methods to which this
100 /// aspect is applied, even when the method exits with an
101 /// exception (this method is invoked from the <c>finally</c>
102 /// block).
103 /// </summary>
104 /// <param name="args">
105 /// Event arguments specifying which method is being executed and
106 /// which are its arguments.
107 /// </param>
108 public override void OnExit(MethodExecutionArgs args)
109 {
110     if (!_ctrlInst.isMonitoringEnabled())
111     {
112         return;
113     }
114
115     // Restore execData.
116     OperationExecutionRecord execData = (OperationExecutionRecord)
117         args.MethodExecutionTag;
118
119     // Time the monitored method is finished.
120     execData.tout = _ctrlInst.getTimeSource().getTime();
121     if (execData.isEntryPoint)
122     {
123         _cfRegistry.UnsetThreadLocalTraceId();
124     }
125
126     // Create a new monitoring record with the measured data.

```

```
126     _ctrlInst.newMonitoringRecord(execData);
127     if (execData.isEntryPoint)
128     {
129         _cfRegistry.UnsetThreadLocalEoi();
130         _cfRegistry.UnsetThreadLocalEss();
131     }
132     else
133     {
134         _cfRegistry.StoreThreadLocalEss(execData.ess);
135     }
136 }
137
138 /// <summary>
139 /// Initializes all relevant data for monitoring.
140 /// </summary>
141 /// <returns>
142 /// The <c>OperationExecutionRecord</c> for this monitoring
143 /// session.
144 /// </returns>
145 private OperationExecutionRecord InitExecutionData()
146 {
147     long traceId = _cfRegistry.RecallThreadLocalTraceId();
148     OperationExecutionRecord execData = new
149         OperationExecutionRecord(_componentName, _method, traceId);
150     execData.isEntryPoint = false;
151     if (execData.traceId == -1)
152     {
153         execData.traceId = _cfRegistry.
154             GetAndStoreUniqueThreadLocalTraceId();
155         execData.isEntryPoint = true;
156     }
157     execData.hostName = _ctrlInst.getHostName();
158     execData.experimentId = _ctrlInst.getExperimentId();
159     return execData;
160 }
161
162 /// <summary>
163 /// Can be used to cut the names of generic types that contain the
164 /// string "'1[...]".
165 /// </summary>
166 /// <param name="name">
167 /// The type name as string to be formatted.
168 /// </param>
169 /// <returns>The formatted type name as string.</returns>
170 private static string FormatType(string name)
171 {
172     if (name != null && name.Contains("'"))
173     {
174         string[] names = name.Split(new char[] { ',' });
175         name = names[0];
176     }
177 }
```

```
174     }
175     return name;
176 }
177
178
179     /// <summary>
180     /// For some reason, <code>args.Method.Name</code> is ".ctor" for
181     /// constructors, whereas there is no "." for all other methods.
182     /// </summary>
183     /// <param name="name">The string to be formatted.</param>
184     /// <returns>The formatted string without a leading ".".</returns>
185     private static string FormatMethodName(string name)
186     {
187         if (name.IndexOf('.') == 0)
188         {
189             name = name.Substring(1);
190         }
191         return name;
192     }
193
194     /// <summary>
195     /// Creates a string that contains the parameter types.
196     /// </summary>
197     /// <param name="parameters">
198     /// A method's list of parameters.
199     /// </param>
200     /// <returns>
201     /// The formatted parameter list as string, e.g.
202     /// <c>(System.Int32, System.String)</c>.
203     /// </returns>
204     private static string FormatParameters(ParameterInfo[] parameters)
205     {
206         StringBuilder formattedParameters = new StringBuilder("(");
207         for (int i = 0; i < parameters.Length; i++)
208         {
209             formattedParameters.Append(FormatType(parameters[i].
210                 ParameterType.FullName));
211             if (parameters.Length > 1 && i < parameters.Length - 1)
212             {
213                 formattedParameters.Append(", ");
214             }
215             formattedParameters.Append(")");
216             return formattedParameters.ToString();
217         }
218     }
219 }
```

Listing A.1: OperationExecutionAspect class.

OperationExecutionAspectProvider

```

1 using System;
2 using System.Collections.Generic;
3 using System.Reflection;
4 using PostSharp.Aspects;
5
6 namespace Kieker.Monitoring.Probe.Postsharp.OperationExecution
7 {
8     /// <summary>
9     /// This class allows build-independent aspect application to
10    /// 3rd-party libraries where no source code is available.
11    /// </summary>
12    public class OperationExecutionAspectProvider : IAspectProvider
13    {
14        readonly OperationExecutionAspect _aspectToApply = new
15            OperationExecutionAspect();
16
17        public IEnumerable<AspectInstance> ProvideAspects(object
18            targetElement)
19        {
20            Assembly assembly = (Assembly)targetElement;
21            List<AspectInstance> instances = new List<AspectInstance>();
22            foreach (Type type in assembly.GetTypes())
23            {
24                ProcessType(type, instances);
25            }
26            return instances;
27        }
28
29        private void ProcessType(Type type, List<AspectInstance> instances
30            )
31        {
32            foreach (MethodInfo targetMethod in type.GetMethods(
33                BindingFlags.Instance | BindingFlags.Public | BindingFlags.
34                DeclaredOnly))
35            {
36                instances.Add(new AspectInstance(targetMethod,
37                    _aspectToApply));
38            }
39
40            foreach (Type nestedType in type.GetNestedTypes())
41            {
42                ProcessType(nestedType, instances);
43            }
44        }
45    }
46 }

```

Listing A.2: OperationExecutionAspectProvider class.

MonitoringControllerWrapper.cs

```
1 using System;
2 using kieker.monitoring.core.configuration; // Bridged Java class.
3 using kieker.monitoring.core.controller; // Bridged Java class.
4
5 namespace Kieker.Monitoring.Core.Controller
6 {
7     /// <summary>
8     /// This class wraps the Java MonitoringController object.
9     /// </summary>
10    internal class MonitoringControllerWrapper
11    {
12        private static readonly object SyncRoot = new Object();
13        private static IMonitoringController _monitoringController;
14
15        internal static IMonitoringController MonitoringController
16        {
17            get
18            {
19                if (_monitoringController == null)
20                {
21                    lock (SyncRoot)
22                    {
23                        if (_monitoringController == null)
24                            _monitoringController = kieker.monitoring.core
25                                .controller.MonitoringController.
26                                createInstance(Configuration.
27                                    createSingletonConfiguration());
28                            AppDomain.CurrentDomain.ProcessExit +=
29                                CurrentDomainProcessExit;
30                    }
31                }
32                return _monitoringController;
33            }
34        }
35
36        private static void CurrentDomainProcessExit(object sender,
37            EventArgs e)
38        {
39            if (_monitoringController != null)
40            {
41                _monitoringController.terminateMonitoring();
42            }
43        }
44    }
45 }
```

Listing A.3: MonitoringControllerWrapper class.

ControlFlowRegistry.cs

```
1 using System;
2
3 namespace Kieker.Monitoring.Core.Registry
4 {
5     /// <summary>
6     /// This class is more ore less a direct copy of the java class.
7     /// </summary>
8     /// <remarks>
9     /// Re-implemented because of the multi-threading capabilities that
10    /// should be handled directly in .NET.
11    /// </remarks>
12    internal class ControlFlowRegistry
13    {
14        private static readonly object SyncRoot = new Object();
15        private static volatile ControlFlowRegistry _instance;
16
17        private long _lastThreadId;
18
19        [ThreadStatic]
20        private static long _threadLocalTraceId;
21        [ThreadStatic]
22        private static int _threadLocalEoi;
23        [ThreadStatic]
24        private static int _threadLocalEss;
25
26        /// <summary>
27        /// Thread-safe Singelton implementation
28        /// </summary>
29        internal static ControlFlowRegistry Instance
30        {
31            get
32            {
33                if (_instance == null)
34                {
35                    lock (SyncRoot)
36                    {
37                        if (_instance == null)
38                            _instance = new ControlFlowRegistry();
39                    }
40                }
41                return _instance;
42            }
43        }
44
45        /// <summary>
46        /// In order to (probabilistically!) avoid that other instances in
47        /// our system (on another node, in another vm, ...) generate the
48        /// same thread ids, we fill the left-most 16 bits of the thread
```

```
49     /// id with a uniquely distributed random number
50     /// (0,0000152587890625 = 0,00152587890625%).
51     /// As a consequence, this constitutes a uniquely distributed
52     /// offset of size  $2^{(64-1-16)} = 2^{47} = 140737488355328L$  in the
53     /// worst case. We restrict ourselves to the positive long values
54     /// so far. Of course, negative values may occur (as a result of
55     /// an overflow) -- this does not hurt!
56     /// </summary>
57     private ControlFlowRegistry()
58     {
59         Random r = new Random();
60         long baseValue = ((long)r.Next(65536) << (sizeof(long) - 16 -
61             1));
62         _lastThreadId = baseValue;
63     }
64     /// <summary>
65     /// This (thread-safe) method provides a trace id.
66     /// </summary>
67     /// <remarks>
68     /// Since we use -1 as a marker for an invalid trace id, it must
69     /// not be returned! The 0 stands for an uninitialized trace id
70     /// and must also not be returned.
71     /// </remarks>
72     /// <returns>A globally unique trace id.</returns>
73     public long GetUniqueTraceId()
74     {
75         lock (SyncRoot)
76         {
77             _lastThreadId++;
78             if (_lastThreadId == -1)
79             {
80                 _lastThreadId++;
81             }
82             if (_lastThreadId == 0)
83             {
84                 _lastThreadId++;
85             }
86         }
87         return _lastThreadId;
88     }
89     /// <summary>
90     /// This method returns a thread-local trace id which is globally
91     /// unique and stores it locally for the thread.
92     /// </summary>
93     /// <remarks>
94     /// The thread is responsible for invalidating the stored
95     /// trace id using the method <c>unsetThreadLocalTraceId()</c>!
96     /// </remarks>
```

```
98     /// <returns>A thread-local trace id</returns>
99     public long GetAndStoreUniqueThreadLocalTraceId()
100     {
101         _threadLocalTraceId = GetUniqueTraceId();
102         return _threadLocalTraceId;
103     }
104
105     /// <summary>
106     /// This method stores a thread-local trace id.
107     /// </summary>
108     /// <remarks>
109     /// The thread is responsible for invalidating the stored
110     /// trace id using the method <c>unsetThreadLocalTraceId()</c>!
111     /// </remarks>
112     /// <param name="traceId">
113     /// The current trace id to be stored.
114     /// </param>
115     public void StoreThreadLocalTraceId(long traceId)
116     {
117         _threadLocalTraceId = traceId;
118     }
119
120     /// <summary>
121     /// This method returns the thread-local trace id previously
122     /// registered.
123     /// </summary>
124     /// <returns>
125     /// The trace id. -1 if no trace id has been registered for this
126     /// thread.
127     /// </returns>
128     public long RecallThreadLocalTraceId()
129     {
130         if (_threadLocalTraceId == 0)
131         {
132             _threadLocalTraceId = -1;
133         }
134         return _threadLocalTraceId;
135     }
136
137     /// <summary>
138     /// This method unsets a previously registered trace id.
139     /// </summary>
140     public void UnsetThreadLocalTraceId()
141     {
142         _threadLocalTraceId = -1;
143     }
144
145     /// <summary>
146     /// Used to explicitly register an execution order index (eoi).
147     /// </summary>
```

```
148     /// <remarks>
149     /// The thread is responsible for invalidating the stored eoi
150     /// using the method <c>UnsetThreadLocalEoi()</c>!
151     /// </remarks>
152     /// <param name="eoi">
153     /// The execution order index (eoi) to store.
154     /// </param>
155     public void StoreThreadLocalEoi(int eoi)
156     {
157         _threadLocalEoi = eoi;
158     }
159
160     public int IncrementAndRecallThreadLocalEoi()
161     {
162         int curEoi = _threadLocalEoi;
163         if (curEoi == -1)
164         {
165             return -1;
166         }
167         int newEoi = curEoi + 1;
168         _threadLocalEoi = newEoi;
169         return newEoi;
170     }
171
172     /// <summary>
173     /// Returns the thread-local execution order index (eoi)
174     /// previously registered.
175     /// </summary>
176     /// <returns>
177     /// The execution order index (eoi). -1 if no eoi registered.
178     /// </returns>
179     public int RecallThreadLocalEoi()
180     {
181         return _threadLocalEoi;
182     }
183
184     /// <summary>
185     /// Unsets a previously registered execution order index (eoi).
186     /// </summary>
187     public void UnsetThreadLocalEoi()
188     {
189         _threadLocalEoi = -1;
190     }
191
192     /// <summary>
193     /// Used to explicitly register an execution stack size (ess).
194     /// </summary>
195     /// <remarks>
196     /// The thread is responsible for invalidating the stored ess
```

```
198     /// using the method <c>UnsetThreadLocalEss()</c>!
199     /// </remarks>
200     /// <param name="ess">
201     /// The execution stack size (ess) to store.
202     /// </param>
203     public void StoreThreadLocalEss(int ess)
204     {
205         _threadLocalEss = ess;
206     }
207
208     /// <summary>
209     /// Recalls and increments the current thread-local execution
210     /// stack size (ess).
211     /// </summary>
212     /// <returns>The incremented execution stack size (ess)</returns>
213     public int RecallAndIncrementThreadLocalEss()
214     {
215         int curEss = _threadLocalEss;
216         if (curEss == -1)
217         {
218             return -1;
219         }
220         _threadLocalEss = curEss + 1;
221         return curEss;
222     }
223
224     /// <summary>
225     /// Returns the thread-local execution stack size (ess).
226     /// </summary>
227     /// <returns>
228     /// The thread-local execution stack size (ess). -1 if no ess
229     /// registered.
230     /// </returns>
231     public int RecallThreadLocalEss()
232     {
233         return _threadLocalEss;
234     }
235
236     /// <summary>
237     /// Unsets a previously registered execution stack size (ess).
238     /// </summary>
239     public void UnsetThreadLocalEss()
240     {
241         _threadLocalEss = -1;
242     }
243 }
244 }
```

Listing A.4: ControlFlowRegistry class.

B. JNBridge Related Configuration Files and Scripts

B.1. Kieker Proxy Generation with JNBProxy

```
1 @ECHO OFF
2 SETLOCAL
3
4 SET JNBRIDGE=C:\Program Files (x86)\JNBridge\JNBridgePro v5.1\
5 SET KIEKER=C:\Kieker
6 SET CLASSPATH=%KIEKER%\dist;%KIEKER%\dist\kieker-1.4-dev-SNAPSHOT.jar;
   %KIEKER%\lib;%KIEKER%\lib\commons-logging-1.1.1.jar
7 SET CLASSLIST=C:\Kieker.JNB\KiekerProxy-classList.txt
8 SET DLL_NAME=KiekerProxy
9 SET DLL_VERSION=1.4.0.0
10 SET JAVA=C:\Windows\system32\java.exe
11
12 "%JNBRIDGE%2.0-targeted\jnbproxy.exe" /ns /pd n2j /cp "%CLASSPATH%" /f "
   %CLASSLIST%" /d . /n %DLL_NAME% /jp "%JNBRIDGE%jnbcore\jnbcore.jar" /bp
   "%JNBRIDGE%jnbcore\bcel-5.1-jnbridge.jar" /pro b /host localhost /port
   8085 /java "%JAVA%" /vn %DLL_VERSION%
```

Listing B.1: Proxy generation BATCH script.

```
1 java.lang.Class r
2 java.lang.Object r
3 kieker.common.record.AbstractMonitoringRecord r
4 kieker.common.record.BranchingRecord r
5 kieker.common.record.CPUUtilizationRecord r
6 kieker.common.record.CurrentTimeRecord r
7 kieker.common.record.IMonitoringRecord r
8 kieker.common.record.MemSwapUsageRecord r
9 kieker.common.record.OperationExecutionRecord r
10 kieker.common.record.ResourceUtilizationRecord r
11 kieker.monitoring.core.configuration.Configuration r
12 kieker.monitoring.core.controller.AbstractController r
```

B. JNBridge Related Configuration Files and Scripts

```
13 kieker.monitoring.core.controller.IMonitoringController r
14 kieker.monitoring.core.controller.MonitoringController r
15 kieker.monitoring.core.sampler.ISampler r
16 kieker.monitoring.core.sampler.ScheduledSamplerJob r
17 kieker.monitoring.timer.ITimeSource r
```

Listing B.2: Kieker proxy class list text file that defines which Java classes to expose.

B.2. Monitoring Configuration Files

B.2.1. Java-Side

```
1 @ECHO OFF
2 SETLOCAL
3
4 SET JNBRIDGE=C:\Program Files (x86)\JNBridge\JNBridgePro v5.1
5 SET KIEKER=C:\Kieker
6 SET CLASSPATH=%KIEKER%\;%KIEKER%\dist\kieker-1.4-dev-SNAPSHOT.jar;%KIEKER%
   \lib\commons-logging-1.1.1.jar;%JNBRIDGE%\jnbcore\jnbcore.jar;
   %JNBRIDGE%\jnbcore\bcel-5.1-jnbridge.jar
7
8 START java -cp "%CLASSPATH%" com.jnbridge.jnbcore.JNBMain /props "
   %JNBRIDGE%\jnbcore\jnbcore_tcp.properties"
```

Listing B.3: Batch file to start Kieker JVM.

```
1 # Java-side (.NET-to-Java) properties
2 javaSide.serverType=tcp
3 javaSide.workers=5
4 javaSide.timeout=10000
5 javaSide.port=8085
6 # .NET-side (Java-to-.NET) properties
7 dotNetSide.serverType=tcp
8 dotNetSide.host=localhost
9 dotNetSide.port=8086
```

Listing B.4: jnbcore properties file for JNBridge (TCP).

```
1 # Java-side (.NET-to-Java) properties
2 javaSide.serverType=http
3 javaSide.workers=5
4 javaSide.timeout=10000
5 javaSide.port=8085
```

```
6 # .NET-side (Java-to-.NET) properties
7 dotNetSide.serverType=http
8 dotNetSide.host=localhost
9 dotNetSide.port=8086
```

Listing B.5: jnbcore properties file for JNBridge (HTTP).

```
1 # .NET-side (Java-to-.NET) properties
2 dotNetSide.serverType=sharedmem
3
4 # edit optional paths to .NET-side assemblies here
5 # IMPORTANT: Use front-slashes (/) in all file paths below
6 dotNetSide.assemblyList.1=path to first .NET-side assembly
7 dotNetSide.assemblyList.2=path to second .NET-side assembly
8 # supply as many .NET-side assemblies as necessary
9
10 # edit path to JNBJavaEntry.dll here
11 # IMPORTANT: Use front-slashes (/) in all file paths below
12 dotNetSide.javaEntry=C:/Program Files/JNBridge/JNBridgePro v4.1/2.0-
    targeted/JNBJavaEntry.dll
13
14 # edit optional path to .NET-side application configuration file below
15 # IMPORTANT: Use front-slashes (/) in all file paths below
16 # dotNetSide.appConfig=path to .NET-side application configuration file
```

Listing B.6: jnbcore properties file for JNBridge (shared memory, only needed for Java/.NET directed communication).

B.2.2. .NET-Side

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <configuration>
3   <configSections>
4     <sectionGroup name="jnbridge">
5       <section name="dotNetToJavaConfig"
6         type="System.Configuration.SingleTagSectionHandler,
7         System, Version=2.0.0.0,
8         Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
9       <section name="javaToDotNetConfig"
10        type="System.Configuration.SingleTagSectionHandler,
11        System, Version=2.0.0.0,
12        Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
13      <section name="tcpNoDelay"
14        type="System.Configuration.SingleTagSectionHandler,
15        System, Version=2.0.0.0,
16        Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
```

B. JNBridge Related Configuration Files and Scripts

```
17     <section name="javaSideDeclarations"
18         type="System.Configuration.NameValueSectionHandler,
19         System, Version=2.0.0.0,
20         Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
21     </sectionGroup>
22 </configSections>
23 <jnbridge>
24     <dotNetToJavaConfig scheme="jtcp"
25         host="localhost"
26         port="8085"
27         useSSL="false" />
28 </jnbridge>
29 </configuration>
```

Listing B.7: .NET application configuration file for TCP/Binary communication with JNBridge.

```
1 <jnbridge>
2     <dotNetToJavaConfig scheme="sharedmem"
3         jvm="%JAVA_HOME%\server\jvm.dll"
4         jnbcore="%JNBRIDGE%\jnbcore\jnbcore.jar"
5         bcel="%JNBRIDGE%\jnbcore\bcel-5.1-jnbridge.jar"
6         classpath="%KIEKER%;
7             %KIEKER%\dist;
8             %KIEKER%\dist\Kieker-1.4.jar;
9             %KIEKER%\lib;
10            %KIEKER%\lib\commons-logging-1.1.1.jar" />
11 </jnbridge>
```

Listing B.8: .NET application configuration file entry for shared memory communication with JNBridge.

Bibliography

- [Bennett 1995] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12: 19–23, January 1995.
- [Bisbal et al. 1999] J. Bisbal, D. Lawless, B. Wu, and J. Grimson. Legacy information systems: issues and directions. *Software, IEEE*, 16(5):103–111, September 1999.
- [Brodie and Stonebraker 1995] M. Brodie and M. Stonebraker. *Migrating legacy systems: gateways, interfaces & the incremental approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.
- [Comella-Dorda et al. 2000] S. Comella-Dorda, K. Wallnau, R. Seacord, and J. Robert. A survey of black-box modernization approaches for information systems. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 173–183, 2000.
- [Cornelissen et al. 2009] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.
- [Desiderata Software 2011] Desiderata Software. EZ JCom version 1.8, May 2011. URL <http://www.ezjcom.com/>. Last visited July 12, 2011.
- [Ehmke et al. 2011] N. Ehmke, A. van Hoorn, and R. Jung. Kieker 1.4 User Guide, October 2011. URL <http://se.informatik.uni-kiel.de/kieker/documentation/>. Last visited October 13, 2011.
- [Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

- [Intrinsyc Software International, Inc. 2011] Intrinsyc Software International, Inc. J-Integra .NET version 2.4, May 2011. URL <http://j-integra.intrinsyc.com/net.asp>. Last visited July 12, 2011.
- [JNBridge LLC. 2011a] JNBridge LLC. JNBridgePro version 5.1, July 2011a. URL <http://www.jnbridge.com/jnbpro.htm>. Last visited September 15, 2011.
- [JNBridge LLC. 2011b] JNBridge LLC. JNBridgePro Users' Guide version 6.0, May 2011b. URL <http://www.jnbridge.com/usersguide.pdf>. Last visited September 15, 2011.
- [JNBridge LLC. 2011c] JNBridge LLC. JNBridgePro Evaluation and Quick Start Guide version 6.0, April 2011c. URL <http://www.jnbridge.com/evaluationguide.pdf>. Last visited September 15, 2011.
- [Khusidman 2008] V. Khusidman. ADM Transformation, June 2008. URL <http://adm.omg.org/>.
- [Kiczales et al. 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, pages 220–242, 1997.
- [OMG 2009] OMG. Architecture-Driven Modernization (ADM): Software Metrics Meta-Model (KDM), March 2009. URL <http://www.omg.org/spec/SMM/Current>.
- [OMG 2010] OMG. Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), December 2010. URL <http://www.omg.org/spec/KDM/Current/>.
- [SharpCrafters s.r.o. 2011a] SharpCrafters s.r.o. Postsharp version 2.1, September 2011a. URL <http://www.sharpcrafters.com>.
- [SharpCrafters s.r.o. 2011b] SharpCrafters s.r.o. Postsharp reference documentation, September 2011b. URL <http://www.sharpcrafters.com/postsharp/documentation>.
- [Software Engineering Group, University of Kiel 2011] Software Engineering Group, University of Kiel. Kieker version 1.4, 2011. URL <http://se.informatik.uni-kiel.de/kieker/>. Last visited August 29, 2011.

- [Stahl and Völter 2006] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, Chichester, UK, 2006.
- [Ulrich and Khusidman 2007] W. Ulrich and V. Khusidman. Architecture-Driven Modernization: Transforming the Enterprise, October 2007. URL <http://adm.omg.org/>.
- [van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, University of Kiel, Germany, November 2009. URL http://www.informatik.uni-kiel.de/uploads/tx_publication/vanhoorn_tr0921.pdf.
- [van Hoorn et al. 2011] A. van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, and N. Wittmüss. DynaMod project: Dynamic analysis for model-driven software modernization. In *Joint Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM 2011) and the 5th International Workshop on Software Quality and Maintainability (SQM 2011)*, pages 12–13, March 2011.

Acknowledgments

I'd like to thank

- my advisor André van Hoorn for his support and advice.
- HSH Nordbank AG for providing desk space and a laptop with a copy of Nordic Analytics, and specifically Sönke Köster and Dr. Marcus Steinkamp of the Financial Engineering department for all the information and courtesy for the case study with their risk evaluation system.
- Prof. Dr. Wilhelm Hasselbring for making this diploma thesis possible.

Declaration

This thesis is my own work and contains no material that has been submitted for any degree or examination at any other university.

To the best of my knowledge and belief, this thesis contains no material previously published by any other person except where due acknowledgment has been made.

Kiel, October 15, 2011

Felix Magedanz

