AGH

UNIVERSITY OF SCIENCE AND TECHNOLOGY

IN KRAKOW, POLAND

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS, COMPUTER SCIENCE AND ELECTRONICS

INSTITUTE OF COMPUTER SCIENCE

MONITORING OF COMPONENT-BASED APPLICATIONS

ERYK CIEPIELA

MASTER OF SCIENCE THESIS

IN COMPUTER SCIENCE

SUPERVISOR

DR INŻ. MARIAN BUBAK

CONSULTATION

MGR INŻ. MACIEJ MALAWSKI

CRACOW, POLAND

JUNE 2007

Acknowledgements

I would like to kindly acknowledge my grateful thanks to Marian Bubak – supervisor of this work, for his support and valuable consultation. Especially, I would like to express my gratitude to Maciej Malawski for his attention, insightful look, advices and time. Also, the help of ACC Cyfronet AGH GridSpace Team cannot be overestimated.

This work was made possible owing to the ViroLab Project – EU project number: IST-027446. The official ViroLab website can be viewed under http://www.virolab.org.

Abstract

The subject of this thesis is the monitoring of component-based applications, focusing on Common Component Architecture model with Java-based CCA-compliant MOCCA framework, which is built over H2O distributed computing platform. Within the scope of this work the monitoring system called leMonAdE was developed that supports generally all Java-based application. In particular, it is targeted to Mocca framework by extending Mocca framework itself as well as Mocca application manager tool, namely Moccaccino, with monitoring capabilities.

The results of this work comprise Java tools for dynamic bytecode instrumentation employing Aspect-Oriented Programming paradigm and monitoring facilities of Java Management Extensions. Mocca framework and Moccaccino Manager are enhanced with monitoring support by incorporating of aforementioned tools within them. Moreover, a prototype of a monitoring tool is provided as an Eclipse IDE plug-in for leMonAdE-enabled edition of Moccaccino manager. The performance tests have been carried out and proved the usability of the system.

This thesis is organized in the following chapters: in Chapter 1 the rationales of this work are explained and the problem is stated. Chapter 2 outlines the technologies addressed by this work and indicates a target platform for emerging system. Chapter 3 is devoted to problem analysis along with discussion of available solutions and current state of the art. In Chapter 4 the goals of this work are precisely specified and the general concept of a monitoring system is presented. Chapter 5 presents detailed design of an emerging system as well as implementation aspects. Chapter 6 gives an answer on how the system developed meets the performance requirements and estimates usability of a solution. Chapter 7 concludes the thesis and marks up future work directions.

Keywords: Monitoring system, monitoring tools, instrumentation, components, component-based applications, Common Component Architecture Aspect-Oriented Programming, Mocca, Moccaccino, leMonAdE.

Contents

List of Figures

Chapter 1 Motivation for a Monitoring System

This chapter generally characterizes the problem and defines fundamental goal of this work. In sections below, the need for a monitoring system in distributed and especially in grid environments is shown. Further, it is said that a monitoring system is a foundation for advanced application development and management tools. Finally, a specific computation architecture model addressed, namely Common Component Architecture, is presented and a problem of this thesis is stated.

Nowadays, a need for modern monitoring means arises. Monitoring has became a vital and, to a significant extent, critical part of distributed systems architectures, especially in grid computing. Execution of application in distributed environment which is usually geographically dispersed and takes hours to complete, remains feedbackless, unresponsive and impossible to supervise, unless the proper means of monitoring are provided. Application run that is distributed both in space and in time induces application managers' demand for a suitable equipment in order to seize and diagnose running applications.

Application managers, in addition to application submission interface, have to be supported with after-submission management and monitoring tools as well. It becomes especially essential in the case of those application managers who are willing to access intermediary results or to check computation status. However, not only application executors cope with the issues of application monitoring. Application developers constitute the next considerable group that needs monitoring assistance in the prototyping and testing stage. They would like to take advantage of monitoring support in order to introspect running code e.g. for verifying, debugging, compatibility testing, performance analyzing, profiling and optimizing purposes.

Nevertheless, monitoring applicability actually exceeds beyond performance monitoring. Indeed, monitoring may enable incorporation into application of additional behavior performing variety of tasks reactively to monitoring events. Therefore, many of application orthogonal aspects such as logging, billing, tracking, reporting etc may be implemented on the top of the monitoring systems.

Monitoring constitutes indispensable foundation for autonomic computing [47]. In comparison with traditional systems, in autonomic self managing system administrator plays thoroughly altered role. Instead of controlling the system directly and manually, they define policies and rules according to which self-management process is operating. Monitoring events are passed towards to the self-management agent which process them and triggers actions of functional areas such as: self-configuration, self-healing, self-optimization and self-protection.

As long as it concerns distributed applications the motivation for a monitoring system is especially justified, because of a number of involved distributed processes that are to be under supervision. Transparent accessing and harnessing grid resources has to allow inspecting the state of these resources. Therefore, monitoring shall provide feedback information related to the resources employed and provide general information about the execution environment. In particular, it applies to component-based applications whose components are deployed into containers. As long as the containers provide computation and memory resources along with basic services they constitute a specific execution environment that is reasonable to monitor by application executors. On the other hand, container providers and site administrators are willing to inspect and measure utilization of the containers under their authority.

Common Component Architecture (CCA) [1][17] model defines its own specific application structure involving architectural part such as components, ports and connections, which are supposed to be observed by interested parties such as application managers. Furthermore, CCA-based application structure may alter during execution since this model enables dynamic assembling and disassembling of components. From a point of view of the application manager, application architecture dynamicity is significantly worth to monitor.

This work addresses, in general, problem of a monitoring in distributed systems. Precisely, it focuses on applications complying component–based architectures. The general goal of this thesis is to provide a framework supporting component application with enabled monitoring capabilities. A solution has to accomplish this goal by addressing specific target platform, specific component application framework along with some dedicated tools. Therefore, Chapter 2 introduces all technologies addressed by the emerging monitoring system and draws a technological background.

Chapter 2 Technologies to Be Addressed by a Monitoring System

This chapter explains how technological constraints affect a monitoring system and outlines a technological background of this thesis. The concrete implementation of component-based application paradigm is presented along with a target component framework that are addressed. Further subsections contained in this chapter discuss technologies and tools involved in a target platform that constitute a foundation for the emerging monitoring system, namely Java platform, H2O distributed application framework, Common Component Architecture, Mocca component application framework and Moccaccino component application manager.

Implementation of monitoring depends heavily on operating system, software platform and application model. Since operating system manages hardware and software resources it has the authority to monitor their utilization as well as to provide static information and characteristics related to them. Moreover, operating systems define a term of *process* in order to enable acquisition and collection of the information related to a particular execution of a program. In fact, what information is provided depends on operating system. Nevertheless, as long as this information determines monitoring capabilities the operating system is fundamental. In general, infrastructure and application monitoring is determined by to what extent operating system supports resources and processes monitoring.

Software platform is usually tailored to the operating system, unless such a platform is designed to be portable. Portability is the realization of programmers' wish to use once written code everywhere. That implies independence from operating system and makes monitoring system specific rather to software platform than to a particular operating system. Interoperability of systems is an issue arising when dealing with distributed heterogeneous environment where compatibility of software platforms is required. Interoperability relies on standards and specifications that heterogeneous software is expected to comply with.

As a matter of fact, application model affects monitoring with its features and qualities which are specific to a particular model. Monitoring shall fit in the model in order to give comprehensive view of application in terms of applied model.

The following subsections provide more detailed insight into addressed platform, employed middleware technologies and frameworks as well as introduces specifications and tools that constitute a basis for this work.

2.1 *Java Platform*

Java is a software platform which has gained extreme popularity at the turn of the century. However it is not expedient here to judge Java worthiness, abilities and limitations, undeniable is the fact that one of the crucial causes of its success is a portability. Thank to idea of JVM Java OS-independence was achieved.

JVM is an open specification [33], which has several implementations (e.g. Sun's Hot Spot [35]) till now. JVM specification is essential for compiler writers who wish to target the JVM and for programmers who want to implement a compatible virtual machine. While JVM knows nothing about the Java programming language itself, it knows a binary class file format called Java bytecode. The main part of JVM specification is therefore the definition of class file format, which is indeed platform and JVM vendor independent. Thanks to the intermediary code specification it is possible to programmatically reengineer such a code - as many libraries do in order to provide non-trivial functionality. Monitoring system which is discussed in this work relies on JVM specification, and precisely speaking, on the binary class file format definition.

Exploiting its portability, Java has been a leader in proposing non-commercial innovative techniques for implementing distributed systems. It is worth to mention at least its Remote Method Invocation [42], Enterprise Java Beans (EJB) [49] and JINI [50] that are widely and successfully used. Ten years of dynamic development of Java technology is nowadays intensified as Java Development Kit (JDK) versions 6 and 7 are announced by Sun to be released as open source in 2007 with the source code available under the GPL v2 license.

It is noteworthy that there is a significant distinction between JVM and Java programming language itself. While Java defines syntax, semantics and compilers producing valid bytecode in JVM class file format, JVM is intended solely to interpret any valid, not necessary produced by Java, class files.

2.2 H2O

H2O [2][24] is a Java-based middleware platform for building and deploying distributed applications. The main feature of this framework is that it decouples the service deployer and the container provider roles. It induces that H2O allows not merely container owners but any authorized third parties or clients themselves deploying services into a kernel. Containers called *kernels*, host services called *pluglets*, which are Java classes exposing remote interfaces. On the transport layer H2O employs RMIX [22], a multi-protocol RMI [42] extension which overlays transport protocols such as RMI and SOAP [51].

2.3 Common Component Architecture

Common Component Architecture (CCA) [17] is a component standard for High Performance Computing [1]. CCA provides standards necessary for component-level interoperability of components developed within different frameworks such as CORBA [52]. CCA is not a complex specification as it basically introduces the concepts of *provides ports* which are public interfaces that a component realizes and *uses ports* to declare dependencies to other components' *provides ports* required. Specific feature of this model is that the application architecture is not static, e.g. components, ports as well as connections may be both added and removed at the runtime. Therefore, such a model is particularly suitable for applications with dynamic architecture reorganization.

2.4 Mocca

Mocca [3] is a CCA compliant distributed component framework based on H2O platform. Current version, namely *Mocca_Light*, is a pure-Java implementation of the CCA framework and allows building component applications on distributed resources available through H2O. Since Java-based H2O platform is generic and able to support generally all distribution models that rely on Remote Procedure Call (RPC) Mocca leverages it in order to support CCA application written in Java. Mocca fully complies the CCA specification, therefore base classes of application architecture derives from standardized interface definitions.

2.5 Moccaccino Manager

In order to automate the process of deployment and management of Mocca applications, the concept of application manager was introduced. Moccaccino Manager was developed as a responsible for resource (H2O kernels) discovery, deployment planning, execution and management of a running application.

In order to support multiple connection and automated dynamic configuration of component instances, Moccaccino enhances the CCA application model with *port qualifiers* and dedicated *Configuration Port*, respectively. Moreover, Moccaccino introduces specific XML-based Architecture Description Language [11], namely Architecture Description Language for Moccaccino (ADLM) [4], that follows the idea of *qualitative component diagram*. Such a diagram facilitates the quantitive parameterization of application. It introduces the terms of *multiple connection* and *component instances group* to denote a number of component instances with connections that follow the same pattern, instead of explicitly specifying each component instance and each connection instance individually.

It was found reasonable to provide monitoring tools as the Moccaccino Manager extensions. Therefore, a complete integrated tool for launching and monitoring Mocca applications will be assembled.

2.6 Summary

This chapter gave a view of a technological background the emerging monitoring system should fit in. The work will be focused on Java-based components in Mocca framework deployed on H2O kernels. Given with that goal, more detailed analysis and review of relevant available solutions is to be carried out. This is what the subsequent chapter is devoted to.

Chapter 3 State of the Art of the Monitoring Techniques

In previous chapters the general goal of this work and the specific technological background were considered. The following chapter identifies and analyses the general challenges encountered when solving previously stated problem of this thesis. A reference monitoring system architecture is proposed that allows ordering and aligning of monitoring-related techniques of a current state of the art. Dedicated subsections discuss how relevant available solutions address the issues identified on each layer.

Monitoring is quite a broad and divers area and as a term is ubiquitous in variety of specifications, frameworks and tools. Many efforts address application monitoring and, in fact, cover distinct concerns. This chapter is devoted to identify problem concerns and to discuss available solutions covering them.

3.1 Problem Analysis

It is reasonable to consider monitoring crosswise the layers of some reference monitoring system architecture in order to separate distinct concerns. The proposed here layering is intended to be as generic as it is possible and not bound to any specific programming paradigm. It is especially motivated as long as monitoring model faces the challenge of inter-technology handover. Since some efforts such as *GridSpace* [21] attempt to engage heterogeneous middleware technologies, proprietary monitoring systems are to be aligned to some generic architecture. Given with a reference architecture all the solution aiming at monitoring discussed in this section may be aligned and positioned in order to classify them and compare with each other.

In proposed reference architecture which is depicted in Figure 1, four layers are involved. First is the *instrumentation layer* which deals with an issue on how to extract, pull out or drag valuable information from running application processes. Further, the *exposition layer* is to provide means in order to make this information remotely accessible, from outside the target application process, restrictedly according to a given policy. Then the third one - *access layer* aims at on how to efficiently access monitoring information in usually distributed environment. Finally, the *tool layer* is intended to process collected monitoring information and serve the end user with functionality based on it in a convenient way.

As it comes to instrumentation, it challenges the issues of hot-plugging into running applications, dynamic enabling and disabling, avoidance of intrusion into the application code, container and frameworks. Because of fact that monitoring may be oriented either to application low-level introspection or to high-level business logic monitoring, the latter implies an issue of instrumentation with enabled access to application runtime state (e.g. variables values).

Besides runtime state, instrumentation has to be aware of a context which supplements monitoring information and is crucial for monitoring information usability. Identified context are:

- Application context – associates monitoring data with a concrete application
- Place context – associates monitoring data with an architectural part of the application placed somewhere in the environment
- Time context – associates monitoring data with timestamp and/or with a sequence number

Moreover, instrumentation mustn't disrupt proper run of the application, hence it must be in accordance to some security constraints and policies.

The fundamental responsibility of *exposition layer* is to provide remote access that enables interoperability. Therefore, it must comply well established standards and protocols. On this layer another security issues of unauthorized access and information confidentiality are encountered as well.

On the exposition layer monitoring data becomes remotely accessible. The aim of the subsequent layer - *access layer*, is to set up logical monitoring data bus in order to redistribute this data to the interested parties. Such a bus is to be adapted to support conversation modes like query-response mode and notification mode. On the *access layer* exposed monitoring information has to be effectively redistributed in usually distributed environment. This layer deals with the issues common for a domain of distributed systems such as: scalability, network throughput, communication latency, jitter and reliability.

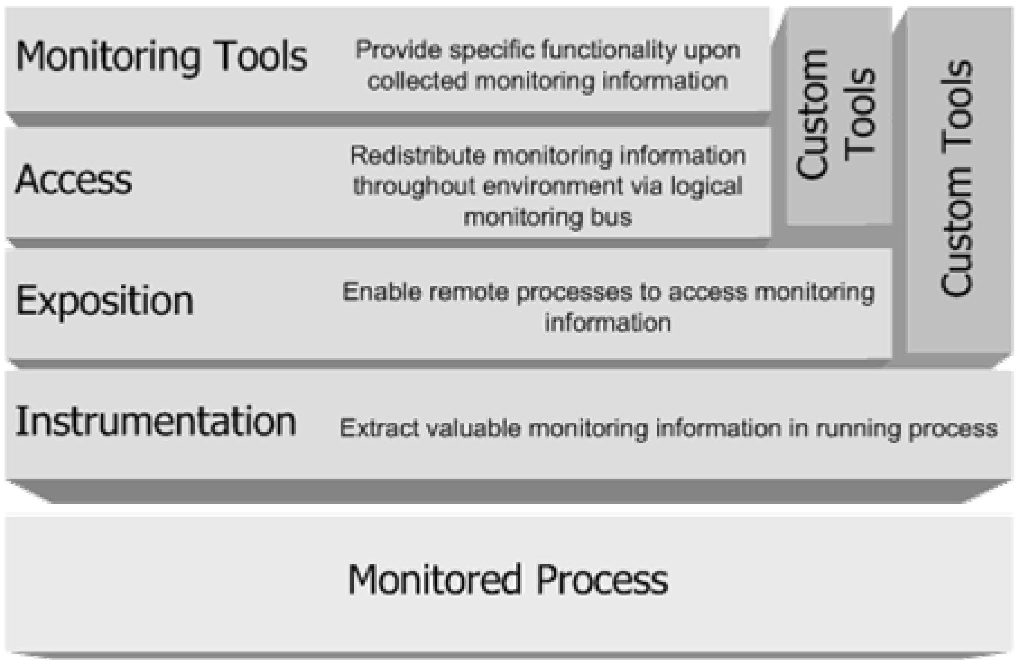

Monitoring Tools
Provide specific functionality upon collected monitoring information
Access
Redistribute monitoring information throughout environment via logical monitoring bus
Exposition
Enable remote processes to access monitoring information
Instrumentation
Extract valuable monitoring information in running process
Monitored Process
Custom Tools
Custom Tools

Figure 1. Monitoring system reference architecture proposed in this work.

On each of the above defined layers, agile monitoring system has to be characterized by a good fitting in the software platform and an easiness of installation and employment. In particular, such a system is expected not to intrude the application development process and impose developers neither to fulfill memory-consuming outgrown library prerequisites nor to deeply derange the execution environment. Especially, adaptation of application in order to enable monitoring involving source code reengineering may be cumbersome, unwished or may affect application reusability.

Especially, it is a principle of the component application model that components perform pure business logic in a container environment that provides them with the common services such as data access, transaction and connectivity services. Following this principle, monitoring should be transparent for component's business logic and should be provided as the container feature.

Moreover, widely applicable monitoring system should adhere to standards and specifications and fit well in the application model paradigms, however, without disrupting it.

3.2 *Discussion of Available Solutions*

This subsection is to discuss available solutions of current state of the art, however it is not expected to give a comprehensive and holistic review of all available solutions. This discussion rather focuses on techniques that are relevant to the specific problem of this thesis and to technologies to be addressed by the emerging monitoring system. In the sections below the approaches, generic models and concrete technologies are presented. The review is organized regarding to the layers of a proposed reference architecture of a monitoring system.

3.2.1 Instrumentation Techniques

Instrumentation techniques in Java world may be generally divided into two types: source code instrumentation and bytecode instrumentation.

The **source code instrumentation** technique relies on developer who is expected to decorate the application code with monitoring-related code. In such an immature approach the monitoring information is emitted e.g. to a console, to a log or to other proprietary tailor-made monitor information collector. Some of the **logging libraries**, such as **log4j** [38] or **Apache commons logging** [53], became universally recognized and accepted as their idea is a standard even outside the Java technology. The strength of the aforementioned two is that they are generic and come along with a number of ready-to-use compatible tools and extensions. However, even state of the art logging libraries do not supports dynamically configurable monitoring and produce lines of logs that are inconvenient to process and analyze.

Java Management Extensions (JMX) [31] incorporated in JDK was expected to equip Java with monitoring facilities. In fact, it gained a lot of popularity in enterprise systems [25], although it isn't suitable for wide range of distributed systems. Among its disadvantages is that it involves source code adaptation and imposes specific design patterns to be applied. **JMX instrumentation layer** assumes that monitorable classes must follow these patterns, and in exchange it offers whole monitoring infrastructure including monitoring agent - MBeanServer, API and remote access facilities.

JMX instrumentation layer imposes some convention and design pattern that has to be applied on early design stage when the crucial design decisions have to be taken. Namely, system architecture has to involve dedicated monitorable classes called Managed Beans (MBeans). It implies that on subsequent development stages it involves disrupting code reengineering. No matter which of the following MBean type is applied such an instrumentation affects system architecture and code:

- Standard MBean – simplest to design and implement as management interface is described by method names,
- Dynamic MBean – must realize `DynamicMBean` interface in order to expose management interface at runtime with generic methods like `getAttribute`, `invokeMethod` which introduce greater flexibility,
- Model MBean – is a Dynamic MBean which is moreover self contained, self described and configurable at runtime,
- Open MBean – Dynamic MBean which involves solely basic data types for universal manageability.

Message-oriented monitoring systems may base on variety of **Message-Oriented Middleware** (MOM) technologies. In order to standardize advanced MOMs for Java industry **Java Message Service** (JMS) API [30] was specified. It allowed keeping the standard by MOM service providers and client applications by decoupling client API and third parties' implementations.

The main and the most significant disadvantage common for all source code instrumentation techniques is that application logic is not separated from the orthogonal concern of monitoring making business logic code disturbed [46]. As it was investigated it requires source code reengineering, plenty of boiler-plate code and recompilation in order to modify monitoring aspect. Source code instrumentation results in a code that is much harder to maintain and enables merely static, poorly configurable monitoring. Last, but not least, such instrumented application remains tightly-coupled with a one particular monitoring system with a direct dependencies to external libraries, which makes the software toughly reusable.

Alternative way is proposed by **bytecode instrumentation** techniques. Since JVM specifies binary class file format - intermediate interpretable program representation, it is possible to carry out instrumentation upon it and to overcome source-code disabilities and defects. Actually, bytecode stands for JVM instruction that is interpreted within virtual machine, however, it is also a customary name for binary class file format.

As long as JVM operates in fact on bytecode, no matter what programming language it originates from, such an instrumentation is specific rather to binary class file format than to Java language itself and may be applied also for applications not written in Java.

Since bytecode instrumentation may be performed on various stages of the bytecode lifecycle, the following policies are to identify:

- **Compile-time bytecode instrumentation –** a class file is instrumented before it is loaded into the JVM. It means that bytecode is transformed by dedicated tools before it is released. In a particular case instrumentation may be carried out along with compilation of source code to bytecode. Hereby, the responsibility of instrumentation is cast to software development process.

– **Load-time bytecode instrumentation –** a class files are released with no instrumentation and remains uninstrumented until they are loaded by JVMs. When class loader is requested to load particular class its bytecode is modified in the fly according to instrumentation request. This mechanism is suitable for one-time instrumentation, however instrumentation is not configurable at the application runtime. It causes performance overhead introduced by bytecode modification when loading classes.

– **Dynamic bytecode instrumentation –** a class which is already loaded, and possibly even running, may be redefined. Classes can be modified multiple times and can be returned to their original state. The mechanism allows instrumentation which changes during the course of execution. It causes performance overhead introduced by bytecode modifications at the runtime, although instrumentation may be dynamically enabled and disabled that allows performance management.

There are a few libraries for Java bytecode engineering. **Byte Code Engineering Library** (BCEL) [15] is intended to analyze, manipulate and even create from scratch at run-time Java class bytecode. Its approach is to provide an object representation of binary class file.

ASM [12] library offers similar functionality as BCEL although with significantly smaller size and increased efficiency. It is especially suitable for dynamic bytecode redefinition [5] [45] in the fly at load time or even when a class have been already loaded thanks to **java.lang.instrument API** incorporated into JDK since version 5.0. The efficiency of ASM in bytecode redefinition is achieved by applying visitor design pattern [37]. The other utilized bytecode engineering libraries are **SERP** [43] and **Javassist** [26].

Bytecode engineering enables variety of non-trivial utilities. Along with reflection and dynamic class loading make Java more dynamic language, and allows generating JVM-compliant code even from non-Java source code. The notable example is scripting language such as Ruby [54] which is interpreted by pure-Java JRuby [55] interpreter using ASM. Moreover, aforementioned libraries are widely employed in many innovative and successful projects such as object-relational mapping framework Hibernate [56], Aspect-Oriented Programming frameworks, compilers, optimizers, code generators and analysis tools. Despite its advantages, bytecode instrumentation itself remains low-level and too cumbersome for a monitoring utility.

Therefore, there are many efforts which are targeted to make both source code and bytecode engineering on higher level and in a more convenient way.

Abstract Syntax Tree (AST) concept is used in parsers as an intermediate between a parse tree and interpreter's internal representation of a program. An AST is derived from a parse tree by omitting syntax elements that do not affect the semantics of the program (e.g. pairs of parentheses are omitted since they actually affect AST structure rather than a program semantics itself). One of the most notable examples is the Eclipse IDE Java Development Tooling [28] which uses its own specific Document Object Model [6] for Java source code. AST nodes constitute join-points where instrumentation may take place.

Standardized Intermediate Representation (SIR) [7] is intended to be an abstract representation for procedural and object-oriented programming languages. It supports Fortran95, Java, C and C++ and it can be generated from either source code or binaries. In latter mode representation involves: packages, classes, methods, invocations and a majority of loops. Representation in source code mode includes additionally all loops, conditionals, exception handling, critical sections and assignments. SIR is utilized by **Monitoring Instrumentation Request** language (MIR) [9] which is XML-based language for performance monitoring of applications. Using MIR, instrumentation tool may obtain SIR owing to SIR-request, then indicate join-points and finally submit instrumentation request.

The alternative approach is introduced by **Aspect-Oriented Programming** (AOP) paradigm. AOP follows a principle of separation of concerns. Whilst functionality is encapsulated in programming language structures such as classes, procedures, methods etc, some cross-cutting concerns code is scattered throughout many structures. Therefore, AOP proposes a technique to handle cross-cutting and orthogonal concerns [46] such as logging, security, monitoring that are loosely-coupled with business logic itself. This technique enables injection of additional behavior code called *advices* which are launched in due course of application control flow. The assembly of advices that cover certain concern is called *aspect* that encloses in an individual module a particular cross-cutting concern. Usually AOP frameworks introduce dedicated Join-Points Models (JPM) which constitute a space of available join-point, along with a pointcut expression language in order to facilitate selection of certain join-point subsets. In the case of Java aspect frameworks their JPMs and pointcut expression languages use terms of Java and OOP phenomena such as classes, methods, fields, inheritance, realization etc.

AOP paradigm may be realized by weaving base source code or bytecode with aspects code, precisely, with the code of aspects' advices. However, some implementations take advantage of a proxy design pattern, using a proxy objects that delegate method calls and wraps them with some additional behavior.

There is a number of AOP framework and among then a majority is addressed to Java, however there exists framework dedicated to e.g. C++ and C#.

AspectJ [13] offers extension of Java programming language enabling development of aspects together with the base business logic. Therefore, it introduces new language constructs such as aspects, inter-type declarations, pointcuts and advices along with its own specific Java compiler which weave source code of a base code with aspects code during compilation.

Subsequent AOP tools such as **JBoss AOP** [27] and **AspectWerkz** [14] do not disturb programming language and they weave bytecodes of ordinary Java base code with ordinary Java aspect code. Instead of introducing dedicated constructs the existing ones are employed: such as annotations or XML-driven configuration. They both are able to weave code at the compilation time, at load time and even, when using Java 5.0 or higher, at the runtime. In contrary to above mentioned frameworks that utilize code weaving, **Spring AOP** [45] takes advantage of proxy design pattern approach and do not support aspect deploying at the runtime. It does not rely on bytecode weaving and cannot be applied outside Spring Framework.

AspectJ, AspectWerkz, JBoss AOP, and Spring AOP are the leading tools in terms of user adoption when taking into account feedback from an active user community as noted in [10]. Basing on insightful investigation and comparison of leading AOP tools made in [10] the most important features compilation may be collected as done in Table 1 and Table 2.

Features	**AspectJ**	**AspectWerkz**	**JBoss AOP**	**Spring AOP**
Aspect declaration	In code	In annotations or XML	In annotations or XML	In XML
Advice bodies	In code	Java method	Java method	Java method
Pointcuts	In code	String value	String value	String value
Configuration	Dedicated `.lst` inclusion list file	Dedicated `aop.xml` file	Dedicated `jboss-aop.xml` file	Dedicated `springconfig.xml` file
Invocation pointcuts	{method, constructor, advice} x {call, execution}	{method, constructor, advice} x {call, execution}	{method, constructor, advice} x {call, execution}	Method execution only
Initialization pointcuts	Class initialization, instance initialization, pre-initialization	Class initialization, instance initialization	Instance initialization	-
Exception handling pointcuts	Supported by dedicated operators	Supported via advices	Supported via advices	Supported via advices
Control flow pointcut expressions	Supported by `cflow` and `cflowbelow` operators	Supported by `cflow` and `cflowbelow` operators	Supported by call stack operators	Supported by `cflow` operator

Table 1. Brief AOP tools comparison according to [10] (part 1).

Features	**AspectJ**	**AspectWerkz**	**JBoss AOP**	**Spring AOP**
Containment pointcut expressions	Supported by `within` and `withincode` operators	Supported by `within`, `withincode`, `has method/field` operators	Supported by `within`, `withincode`, `has method/field` operators	-
Special pointcut expression operators	Conditionals	-	Dynamic `cflow` operator	-
Dynamic advice context	Supported by `this`, `target`, `args` variables passed to advices	Supported by `this`, `target`, `args` variables passed to advices	Supported via reflective access	Supported via reflective access
Extensibility	Thanks to abstract pointcuts	By overriding, advice bindings	By overriding, advice bindings	By overriding, advice bindings

Table 2. Brief AOP tools comparison according to [10] (part 2).

Pointcut expression languages introduced by Aspect-Oriented Programming tools establish an alternative to Abstract Syntax Trees, as long as it provides a language to select subsets of join-points according to some programming language-specific rules. A comparison of AOP-like and AST-like approaches which takes into consideration AspectWerkz and Standard Intermediate Representation, respectively, is contained in Table 3 and Table 4.

AST-like approach of Standard Intermediate Language	**AOP-like approach of AspectWerkz**
Deepened introspection into program – to code regions level	Coarse grained join-point space: before/after/around method calls or field access, exception caught etc. without descending to method bodies; follows the hermitic principle of Object-Oriented Programming - no access to the implementation of methods what induces abstraction layers
No dynamic context – no access to runtime values	Rich dynamic context – reference to target object, class members' signature and call arguments
Only individually and explicitly specified code region is instrumented at once	Pointcuts expression languages are regular-expression-based supporting terms of programming language such as inheritance, realization, encapsulation etc.
Provides only static Abstract Syntax Tree of a program	Besides syntax representation it provides also access to runtime values (dynamic context, aspect context)
Standard supporting C, C++, Java, and Fortran	Java-specific
Supports only build built-in intelligence (e.g. metrics suite)	Enables aspect intelligence - proprietary code (advice) is invoked reaching given join point

Table 3. Comparison of AST-like and AOP-like instrumentation approaches basing on Standard Intermediate Language and AspectWerkz library, respectively (part 1).

AST-like approach of Standard Intermediate Language	**AOP-like approach of AspectWerkz**
Thread-relevant features	No thread-relevant features
Supports snapshots of current state of the process	Aspects are stateful and may be monitored and traced
Join points space is ordered with no relations	Many join point relations introduced (e.g. inheritance, regular expression of method name, location in the code etc); allows finding join point matching to the given criteria
Supporting code regions level not always feasible when given only with the binaries	Bases only on bytecode
Events are propagated through network and processed in monitoring tool-side	Advices are locally executed in target process, events may be pre-processed locally according to some logic, generated events may be better grained

Table 4. Comparison of AST-like and AOP-like instrumentation approaches basing on Standard Intermediate Language and AspectWerkz library, respectively (part 2).

3.2.2 Exposition Techniques

Exposition techniques, they may base on a number of approaches. One of them is Message-Oriented Middleware approach of JMS [30] which involves third-party message service providers for message passing. Generally, JMS supports two message destination types: queues and topics, with corresponding message redistribution modes: producer-consumer and publisher-subscriber. The are lots of JMS implementation supporting variety of features such a message persistence, reliability, acknowledgments etc.

Unlike MOM, JMX descends from remote object idea. **JMX agent layer** introduces *MBeanServer* in which MBeans are registered in order to make them remotely accessible. Remote processes may access *MBeans' features* such as attributes, operations and notifications. The transport layer is separated from *MBeanServer* itself and is provided by JMX Distributed Services Layer.

A specific solution is provided by **Java Platform Debugging Architecture** (JPDA) [32] which is designed for debugging in development environments for desktop systems. It is a specification and, in the same time, a reference implementation of services that JVM must provide for debugging purposes. It allows remote debugger JVM to access debuggee JVM through Java Debug Wire Protocol and control the execution of application contained in debuggee machine in the classical debugger-like manner. Given with such a specification, tool developers are enabled to easily create portable and reusable debugger utilities. However, beyond debugging in development environments JPDA no longer seems to be suitable.

Some other proprietary solutions may base on TCP/IP or UDP/IP protocols stack, that are widely supported in distributed environments fabric. The example to mention is dedicated log4j log appender using plain TCP sockets. Higher level protocols that define additionally interoperable message format such as SOAP [51] may be applied to. Also Web Services intended as a platform-independent Internet standard is nowadays focusing on specification of features supporting various message exchange modes such as **WS-Notification** [57].

3.2.3 Access Techniques

JMX Distributed Services Layer

Over its agent layer JMX provides services for clients and management application to enable access and interaction with MBeanServers and the managed via MBean resources in the servers. Clients connect to MBeanServers thanks to:

- **Connectors** – composed of a pair of client and server-side compatible communication endpoints. Client part provides a protocol-independent API to access the remote MBean Server.
- **Adapters** – intended to creates a facade view of *MBeanServer* through a given protocols such as HTTP or SNMP, provides only server-side part.

Using either connector or adapter, the client API enables manipulation over MBeanServer and registered MBeans. Moreover, MBean Proxy of a given MBean may be created on the client-side. Proxy propagates operations to perform to the corresponding MBean and in the same time MBean propagates notification to all of its proxies.

Nonetheless, interoperability is a vital issue on JMX Distributed Services Layer. Therefore adapters relies on standardized protocols such as HTTP and SNMP. Since Web Services is widely accepted and adopted standard, the efforts are undertaken in order to standardize Web Services connector for JMX agents [58] in order to allow non-Java clients managing Java applications.

JGroups [29] is a one of the libraries for multicast communication that especially puts emphasis on reliability. Thank to it, processes can set up ephemeral group across LANs or WANs, join groups, send messages to members and receive messages from members in the group. It doesn't necessarily mean that IP Multicast is used since JGroups feature is a flexible protocol stack.

The monitoring bus endpoints may be arranged in more decentralized and complex topologies of peer-to-peer network. Such an architecture balances network load and prevents from network throughput bottlenecks. It also enables spontaneous establishment of a scope environment of peer groups. **JXTA** [34] specifies a set of open protocols that standardize the manner in which peers discover each other, self-organize into peer groups, advertise and discover network resources, communicate and monitor each other.

3.3 Summary

The analysis of a problem and introduction of the layers allowed identification and discussion about the more or less partial and relevant solutions. This section gave a view how the issues are addressed by current techniques, which of them are solved and which still remain unaddressed.

Basing on a knowledge of the existing solutions the emerging system may utilize them in order to fulfill specific requirements. Moreover, system will focus on challenges that are poorly covered in the current status.

Chapter 4 Concept of the leMonAdE Monitoring System

In the present chapter the detailed goals of this thesis are specified. These goals are to be achieved with an emerging monitoring system, namely leMonAdE. The goals induce the requirements along with the general concept on how to fulfill them in a given technological background. The further subsections are devoted to the detailed description of the concept of leMonAdE monitoring system.

A concept presented in this thesis addresses generally Java-based applications. Afterwards, upon instrumentation and exposition techniques foundation the concept is extended in order to support Mocca framework applications over grid fabric.

The concept addresses requirements that determine practical usefulness and wide applicability of an emerging solution. Requirements bear in mind software engineering process that the concept has to fit in and efficiently improve. The requirements imposed are to ensure that emerging system will be as much as it is possible reusable and adaptable with other solutions and technologies.

4.1 Detailed goals

The present section provides the enumeration of detailed goals that are to be addressed by a monitoring system. The goals are individually motivated and explained as follows:

- **Separation of business logic and monitoring concern.** From the application development perspective the crucial is a requirement of no involvement of monitoring concern in course of application development. The fundamental intention of this work is to eliminate monitoring-related code disrupting the application business logic implementation.
- **Adherence to existing standards and specifications.** The solution is expected to remain universal as long as it's possible, by adhering to specifications and standards. Since there exist instrumentation and exposition techniques which are said universal, that rely solely on general specifications, they may be successfully employed in whole Java world no matter what application model, framework or architecture is applied. Although access layer by definition strongly adheres to the architecture of application, tools layer may cover underlying application model owing to some well-specified interface between monitoring infrastructure and monitoring tool. Nonetheless, access layer being the one who is to be aware of actually employed application model has address to CCA model.
- **Introspective monitoring.** The solution has to assist application developer and executor in low-level introspection monitoring while testing debugging, validating profiling applications. It has to be usable to monitor running application from the developer’s point of view. It is to monitor how application works in terms of programming model, language and paradigms.
- **Monitoring of high-level business logic.** Nonetheless, monitoring has to provide means for high-level business logic monitoring. It has to enable developer to instrument application with some additional behavior that will expose monitoring and management valuable information. A custom monitoring intelligence is to be of the developer's choice and may be responsible for e.g. extraction of intermediary results or tracing the status of executed application. High-level business logic monitoring is to monitor how application works in terms of problem domain.
- **Dynamic instrumentation.** Monitoring aspect should be dynamically pluggable at the runtime and easy to enable and disable repeatedly in order to do not constantly affect performance. It implies hot-plugging and hot-unplugging into running application.

– **Agile adaptation.** For a developer's convenience the emerging solution has to be well coordinated with existing techniques, and as much as its possible free of external dependencies. Then, the adaptation of target application code and development environment in order to enable monitoring will become effortless.

– **Monitored application model.** Another issue to address is monitored application model based on CCA specification. A formal description model is necessary since application instrumentation specification has to refer to the application architecture. Application model has to constitute a space for *place contexts* denoting which architectural part of application the monitoring event is related to.

– **Minimizing overhead.** The crucial factor that determines usability of a monitoring system is the overhead that is introduced. One of the method of minimization that was already mentioned is a dynamic instrumentation which makes monitoring pluggable so that overhead is introduced only when the monitoring is turned on. The emerging solution has to employ only efficient mechanisms.

– **Security.** While designing monitoring system security issue has to be borne in mind. Although emerging solution is not expected to implement security concern, but rather to take into consideration plugging custom security modules into it on the design stage.

4.2 The name of the system

The emerging monitoring system was named **leMonAdE** that stands for Agi**le Mon**itoring **Ad**herence **E**nvironment. The terms used in its name reflect concept along with characteristics expected from such a system:

– **Agile** – not imposing application framework, well coordinated with existing technologies, (as much as it’s possible) free of external dependencies and effortlessly employable

– **Adherence** – not requiring adaptation of target application code and easy to employment in the case of already existing applications, rather adherable than application-intrusive, adhered to standards and specifications

– **Environment** – providing a monitoring utilities toolset.

4.3 *Concept Overview*

Roughly explained, the concept consists of several ideas. Each idea addresses one of the layer of a reference monitoring system architecture. On instrumentation layer, in general, the Aspect-Oriented Programming paradigm are to be supported by dedicated custom class loader, namely Aspect-Oriented Class Loader (AOCL) that is presented in section 4.4. Instrumentation will rely on bytecode weaving of a base application code with a code of the aspects. Such the aspects are named Monitoring Aspects and are further discussed in section 4.5. On the exposition layer, the Monitoring Aspects instances are to be monitoring-enabled entities instrumented as MBeans. Such the aspects registered in MBeanServer will expose the remote interfaces as it is described in details in section 4.6. The client stubs for accessing remote aspect MBeans are to be contained in Aspect-Oriented Class Loader Registry as explained in section 4.7. Over the registry the monitoring tools may emerge that will support the aspect-based monitoring of an application as said in details in section 4.11.

The concept may be adopted in Mocca framework and Moccaccino Manager and as shown in sections 4.8 and 4.10. In particular, it may take advantage of the idea of Architecture Description Language for Moccaccino which is presented in section 4.9.

How the above described ideas map on architecture is shown in Figure 2. The subsequent sections present successively the ideas layer by layer.

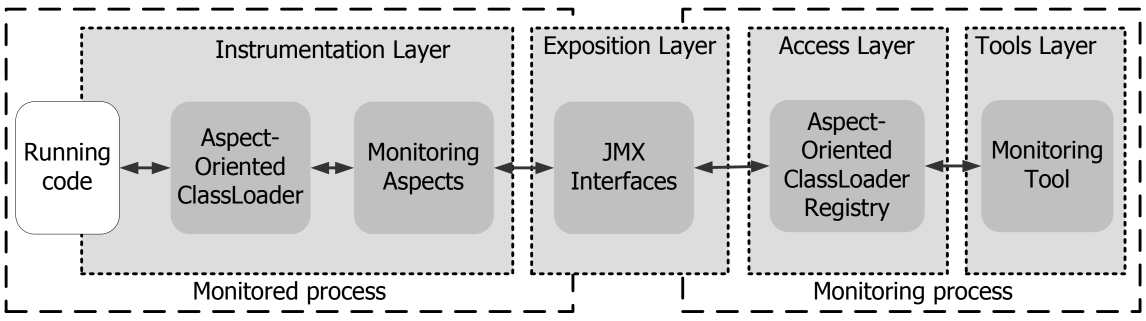


Figure 2. The ideas involved in a monitoring system concept spread over the reference monitoring system architecture.

4.4 *Aspect-Oriented Class Loader*

This work is intended to fit well in Java platform, therefore it should be pluggable in the well-defined extension points that Java platform defines. One of them is as custom class loading policy [40]. Custom class loaders define the manner in which JVM dynamically loads classes, and in the same time constitutes classes' namespace context.

Thanks to custom class loader providing classes' bytecodes from arbitrary, even remote location may be achieved. Particularly, it may handle on demand loading of foreign bytecode that is to be woven with application base code. Moreover, as long as class loaders preserve class namespaces, bytecode transformation may be scoped to the classes loaded by particular classloader without interfering other classes. This introduces isolation, which is especially significant in the case of middleware containers. Basing on one of the aspect frameworks, such a custom class loader may provide a functionality of aspect weaving and may enable convenient and easy to employ tools for aspect-oriented programming. The concept of Aspect-Oriented Class Loader (AOCL) exceeds beyond monitoring targets, and may be utilized as a generic AOP utility, nevertheless, it is particularly suitable for monitoring use [8]. As long as Java object is given with a reference to a class loaded, it provides aspect object with the AOCL context information. If so application context and space context information may be contained in AOCL instance. By encapsulating all the instrumentation functionality within custom class loader, it is easy to adapt middleware containers by simply replacing appropriate class loader with AOCL or by extending class loader hierarchy by AOCL as shown in Figure 3.

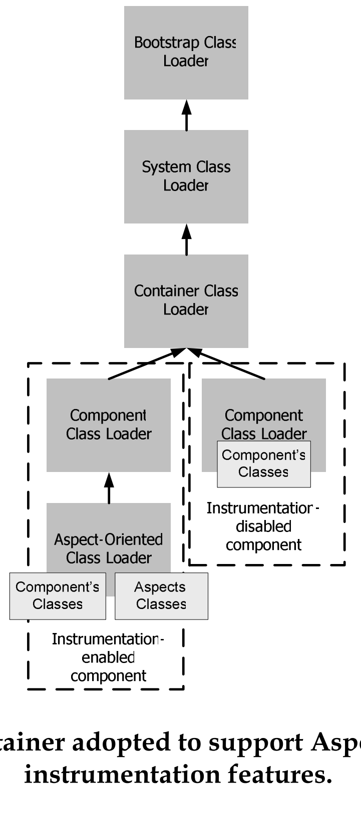


Figure 3. Middleware container adopted to support Aspect-Oriented Class Loader instrumentation features.

Owing to bytecode instrumentation techniques AOCL-like instrumentation affects solely containers without reengineering of application source code. However, AOP frameworks are bound with their own Join-Point Model. This in fact puts restrictions for the foreign code to be inserted only in defined join points, therefore, developer has to bear in mind the Join-Point Model while developing aspect code.

4.5 Monitoring Aspects

AOCL enables deployment of aspects of every kind, accomplishing unrestricted variety of functionalities and, in particular, monitoring.

The monitoring aspect is a stateful entity that is not limited to emitting out notifications in a sensor-like manner. Since it is stateful it may contain some monitoring intelligence and may process interception according to some business logic. Such a monitoring aspect acts rather like a monitor, because it is able to provide information of higher level of abstraction, to handle queries and operation invocations. It may be as well containing some static context information that are passed while instantiating to aspect instance, as well as dynamic context information that is programmatically accessible via variables within advice method bodies.

In contrary to monitoring aspects the ordinary foreign code is acting like a sensor. It is a stateless behavior, not managed and devoid of dynamic context, therefore, is not able to enable non-trivial processing. It may solely forward interceptions by emitting out notifications. Moreover, it cannot handle queries and operation invocations. Comparison of sensor-like and monitor-like instrumentation is depicted in Figure 4.

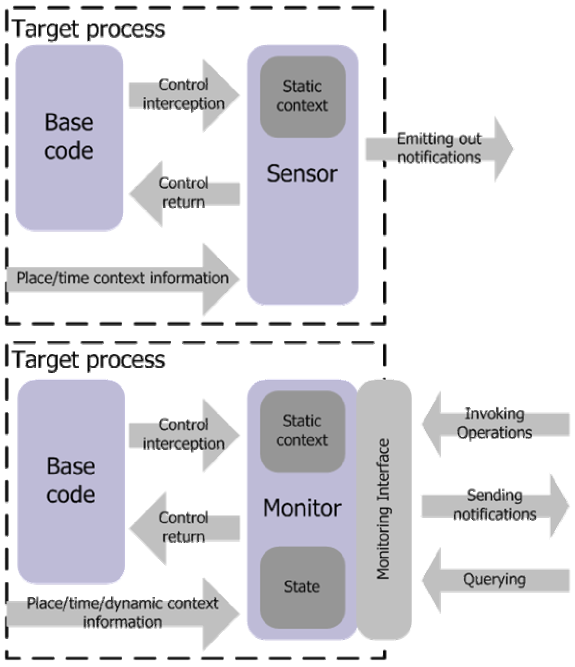


Figure 4. Comparison of sensor-like and monitor-like instrumentation.

4.6 *JMX Interface*

In order to make aspects monitorable and manageable from remote location JMX technology is suitable and was successfully employed in several similar solutions [41].

The concept lies in making both AOCL and aspects manageable via JMX by making their classes MBeans. In a particular case of standard MBeans it imposes solely, besides reasonable, design pattern and naming convention of remote interfaces. Having MBeans registered in so-called MBeanServer, some of their features such as attributes, operations and notification are exposed. JMX clients may then connect MBeanServer and access features of registered MBeans.

Which class' attributes and operations are to be exposed depends on MBean type. In a basic standard MBean it is determined by a corresponding MBean remote interface, which name is imposed by a convention by adding `MBean` suffix to the MBean implementation class name.

AOCL may expose management interface containing operations related to AOCL configuration, codebase locations, aspect deploying, aspect destroying, setting context information etc. Moreover, it may expose some valuable information via MBean's attributes. What is more, each aspect may expose its own arbitrary set of attributes and operation and may emit out notification of its choice towards the subscribers.

MBeanServer ensures secure access from remote parties by supporting pluggable security management, that addresses issues of unauthorized access and information confidentiality. JMX is another well-defined extension point of JVM that fully relies on standard specifications.

4.7 *Aspect-Oriented Class Loader Registry*

Having MBeans registered in MBeanServer connected clients may request for MBean attributes, operations and notifications. Such a client may be ordinary JMX consoles as well as more elaborate tools, or custom clients built upon JMX API.

Basing on this API AOCL's MBean client stub can be developed and considered as a monitor plugged to corresponding AOCL, namely AOCL Monitor. Such a monitor may provide access not only to AOCL itself, but as well to all the aspect deployed in a scope of given AOCL.

Since usually a number of AOCLs would be involved in application execution they has to be registered in one or more *MBeanServers* distributed in grid environment, each per one JVM. The application managers may then maintain dedicated registry providing AOCL Monitors, one per each AOCL. Regarding to AOCL specification, all that AOCL Registry needs is the MBeanServer address and AOCL's unique object name. Basing of such AOCL Registry monitoring tools have access to all registered AOCLs and aspects in their scope.

4.8 Adaptation of Mocca Framework

The aforementioned concepts may be applied e.g. to Mocca framework which will result in monitoring-enabled Mocca edition. However, it is worth to emphasize that they can be applied to other framework as well. Since this work is focused on component-based applications the Mocca framework will be addressed. The instrumentation of Mocca framework can be carried out as like in the case of any other Java-based framework by replacing appropriate class loader with AOCL.

4.9 Architecture Description Language for Moccaccino

Among Architecture Description Languages (ADLs) [11] for component-based architectures ADL for Moccaccino (ADLM) [4] can be distinguishes as a language dedicated to CCA. ADLM introduces a concept of *qualitative component diagram* that focuses on component groups and their (possibly multiple) connections, rather than on individual component instances and actual connections multiplicity. It allows architecture modelers dealing with a higher level, concise, easy to refactor UML-like diagram instead of overgrown and inconvenient component instances' map (see: Figure 5). ADLM is a textual representation of Application Object Model which is the API incorporated into Moccaccino Manager for modeling application architecture programmatically.

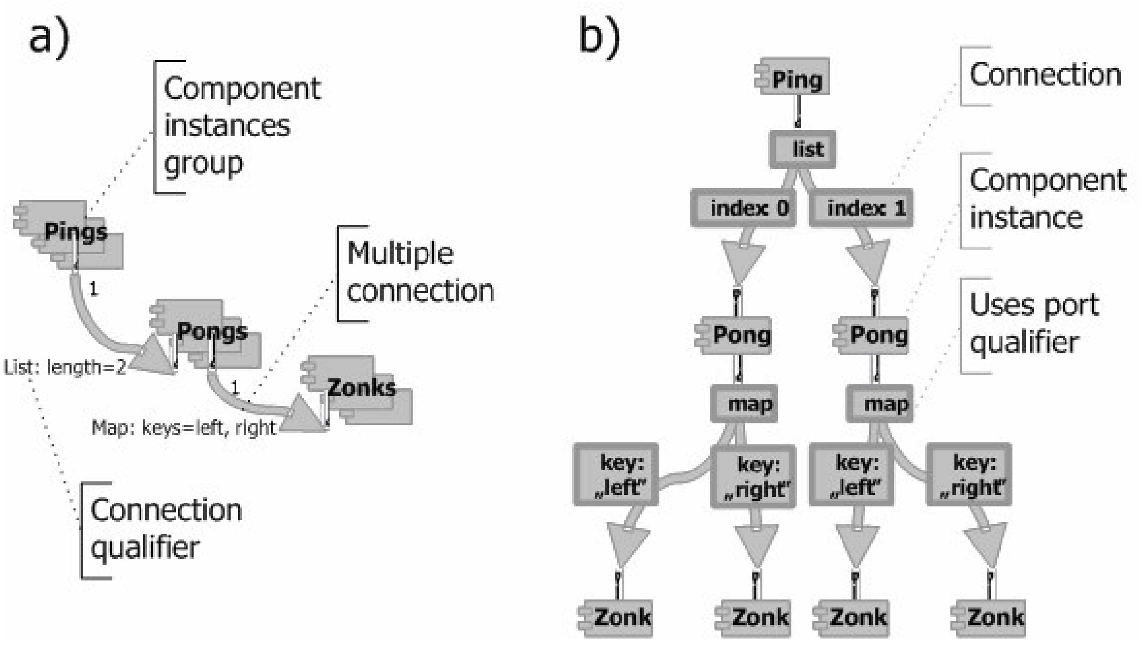


Figure 5. Concise qualitative component diagram of ADLM (a) that is resolved to the plain component diagram (b).

Sample component application architecture of Figure 5 may expressed in ADLM format as shown in User Manual in Code Snippet 28.

4.10 Extensions to Moccaccino Manager

As mentioned earlier, application managers are obliged to maintain AOCL Registry. Therefore, each time they request instantiation of AOCL they have to add it in the registry. In the case of Moccaccino Manager, Application Handler is an entity responsible for handling running application component instances and may accept as well responsibility of AOCL Registry.

Moreover, component deployment activity has to be changed as long as it has to take care of a proper configuration and initialization of AOCL. Component deployment request has to include the configuration of AOCL within which component instance would be deployed. The role of the AOCL configuration provider has to be fulfilled by dedicated *Instrumentator* module. From the one hand it has to take as an input instrumentation specification on the application level, from the other it has to return concrete configurations for AOCLs that correspond to component instances. Such a configuration should contain e.g. application and place context information. Application Instrumentation Specification for Moccaccino (AISM) may be expressed in an

XML-based format that refers to architectural parts of application as described in the ADLM document.

4.11 Monitoring Tools

Monitoring utilities toolkit will be provided as a set of ready-to-use generic aspects which could be deployed, managed and monitored by any of third party's JMX consoles. Moreover, more tailored JMX-based consoles will take advantage of AOCL Registry in order to enable simultaneous monitoring of all distributed AOCLs involved in application. Such aspects toolkit is intended to assist in introspection monitoring.

Besides using ready-to-use generic aspects, developers are allowed extending them or even create from scratch custom aspects. Custom aspect provided by developer may realize monitoring functionality on higher level of abstraction.

Chapter 5 Design and Implementation of the leMonAdE Monitoring System

This chapter provides more insightful view in the monitoring system concepts and shows how this concepts were realized. The following subsections present an architecture of a system, detailed design and discuss implementation issues. This chapter is organized regarding to subprojects that has been carried out in scope of this work: leMonAdE AOCL which constitutes a core of the monitoring system, Mocca leMonAdE edition that incorporates monitoring capabilities into Mocca framework, Moccaccino leMonAdE edition that supports management of monitoring-enabled Mocca application and an Eclipse IDE plug-in that provides UI for monitoring-enabled Moccaccino manager.

Since architecture, generally, follows proposed reference architecture, design is divided into four above described layers as it is depicted in Figure 6. Detailed design has been carried out separately for each layer.

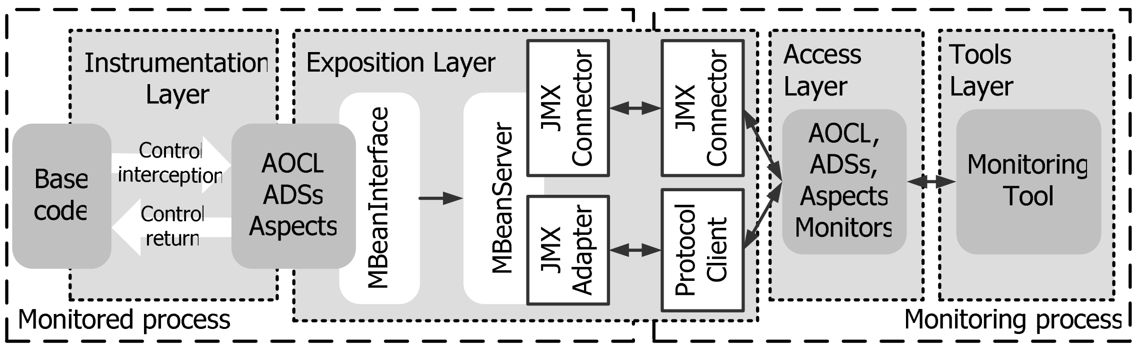


Figure 6. Overall general architecture of the monitoring system.

Therefore, design is divided into several subprojects:

– **leMonAdE AOCL** – core subproject that provide full functionality of instrumentation, exposition and access layer to utilize in arbitrary Java-based application model as it is not specific to any.

– **Mocca leMonAdE edition** – Mocca framework adopted to support leMonAdE, introducing AOCL.

– **Moccaccino leMonAdE edition** – Moccaccino Manager adequately modified in order to support Mocca leMonAdE edition.

– **Moccaccino leMonAdE edition Eclipse plug-in** – introduces UI to Moccaccino leMonAdE edition functionality in a form of Eclipse IDE plug-in for Mocca application developers' convenience.

Dependency between above enumerated subprojects is that both Mocca leMonAdE edition and Moccaccino leMonAdE edition uses leMonAdE AOCL libraries. Moreover, Moccaccino Manager obviously requires Mocca framework libraries while Eclipse plug-in relies on Moccaccino libraries.

5.1 leMonAdE AOCL

5.1.1 Instrumentation Layer

Obviously, `AspectOrientedClassLoader` class is a subclass of `java.lang.ClassLoader`. The overridden method `loadClass` performs seeking class' bytecode in dynamically added locations such as remotely staged `jar` file or local file system path.

Seeking of class bytecode is delegated to `BytecodeProvider` class which is composed into `AspectOrientedClassLoader` class. Currently, it performs seeking in either local file system path or in arbitrary located `jar` file under given URL that is resolvable.

What is crucial, class loaders are to be arranged hierarchically. It means that class' bytecode seeking must be first delegated to the parent. This prevents from loading the same class twice in distinct class loader what causes incompatibility of such loaded classes.

As an aspect framework the AspectWerkz was chosen as the one best fitting in the requirements. It enables runtime bytecode weaving, provides comprehensive Join-Point Model along with powerful and straightforward pointcut expression language. What is more, it provides programming utilities that facilitate aspect development (e.g. annotations-driven development). It is constantly improved, well documented and successfully employed in several solutions.

AOCL manages zero or more *Aspect Deployment Scopes* (ADS) which actually specify a subset of join-points which are pre-instrumented. They restrict join-point space since only pre-instrumented join-points are eligible for instrumentation, and therefore prevent from unwished instrumentation and injection of disrupting foreign code.

The join-point subsets of ADSs are expressed in pointcut expression language and have to be specified before loading application classes in order to perform required pre-instrumentation of join-points while loading application bytecode. It is required to do so, as long as pre-instrumentation modifies class structure, which is not allowed by modern JVMs to redefine class structure after it is loaded. Nonetheless, aspects deployment doesn't affect instrumented class structure and, therefore, it may be performed after the instrumented class is loaded. ADS constitutes moreover a logical context within which aspects are to be deployed. The role of ADS and its relation with aspects it graphically explained in Figure 7.

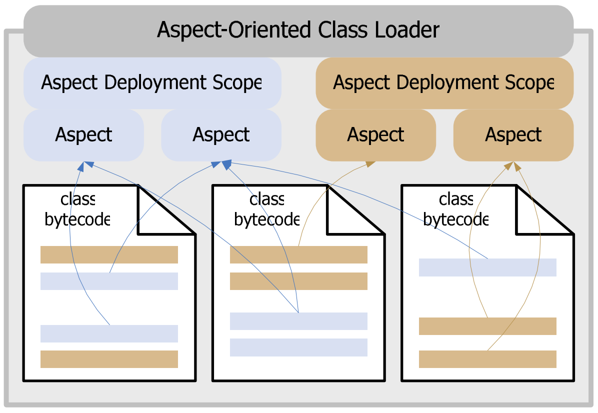


Figure 7. Aspect Deployment Scopes (ADS) as ones that pre-instrument certain subsets of join-point (filled bars denote join-points pre-instrumented by ADSs). Aspects may instrument only join-points previously pre-instrumented by ADS that scopes given aspect (arrows denote instrumentation by concrete aspects).

ADS are designed to be either *static*, such that having a fixed pointcut expression specified before ADS deployment, or *parameterized*, such that having a parameterized pointcut expression specified, whereas the actual pointcut expression is resolved at the ADS deployment time, according to the parameter values provided. Above mentioned two are represented as distinct subclasses of `AspectDeploymentScope` class. ADS are deployable via AOCL methods. Static ADS definition include only a pointcut expression, while parameterized ADS are to be provided as a class or as a properly manifested `jar` file. Such a jar file is expected to has manifest file with attribute `Aspect-Deployment-Scope-Class-Name` indicating what is a class implementing *parameterized* ADS.

Deploying ADS requires ADS definition along with the name identifier of ADS, moreover, in the case of parameterized ADS it requires as well a complete set of parameters values.

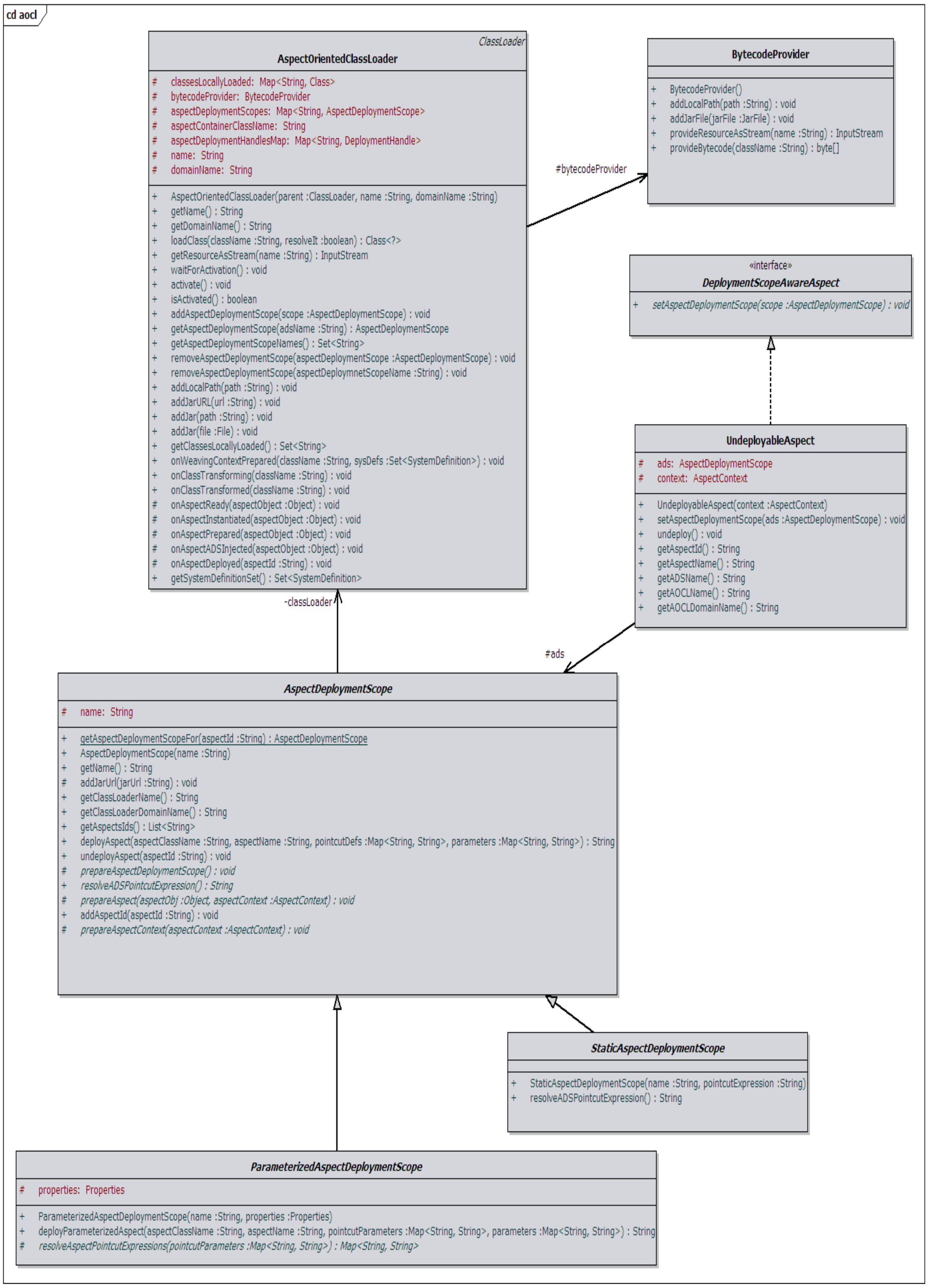

cd aocl
ClassLoader
AspectOrientedClassLoader
# classesLocallyLoaded: Map<String, Class>
# bytecodeProvider: BytecodeProvider
# aspectDeploymentScopes: Map<String, AspectDeploymentScope>
# aspectContainerClassName: String
# aspectDeploymentHandlesMap: Map<String, DeploymentHandle>
# name: String
# domainName: String
+ AspectOrientedClassLoader(parent :ClassLoader, name :String, domainName :String)
+ getName() : String
+ getDomainName() : String
+ loadClass(className :String, resolveIt :boolean) : Class<?>
+ getResourceAsStream(name :String) : InputStream
+ waitForActivation() : void
+ activate() : void
+ isActivated() : boolean
+ addAspectDeploymentScope(scope :AspectDeploymentScope) : void
+ getAspectDeploymentScope(adsName :String) : AspectDeploymentScope
+ getAspectDeploymentScopeNames() : Set<String>
+ removeAspectDeploymentScope(aspectDeploymentScope :AspectDeploymentScope) : void
+ removeAspectDeploymentScope(aspectDeploymnetScopeName :String) : void
+ addLocalPath(path :String) : void
+ addJarURL(url :String) : void
+ addJar(path :String) : void
+ addJar(file :File) : void
+ getClassesLocallyLoaded() : Set<String>
+ onWeavingContextPrepared(className :String, sysDefs :Set<SystemDefinition>) : void
+ onClassTransforming(className :String) : void
+ onClassTransformed(className :String) : void
# onAspectReady(aspectObject :Object) : void
# onAspectInstantiated(aspectObject :Object) : void
# onAspectPrepared(aspectObject :Object) : void
# onAspectADSInjected(aspectObject :Object) : void
# onAspectDeployed(aspectId :String) : void
+ getSystemDefinitionSet() : Set<SystemDefinition>
#bytecodeProvider
BytecodeProvider
+ BytecodeProvider()
+ addLocalPath(path :String) : void
+ addJarFile(jarFile :JarFile) : void
+ provideResourceAsStream(name :String) : InputStream
+ provideBytecode(className :String) : byte[]
«interface»
DeploymentScopeAwareAspect
+ setAspectDeploymentScope(scope :AspectDeploymentScope) : void
UndeployableAspect
# ads: AspectDeploymentScope
# context: AspectContext
+ UndeployableAspect(context :AspectContext)
+ setAspectDeploymentScope(ads :AspectDeploymentScope) : void
+ undeploy() : void
+ getAspectId() : String
+ getAspectName() : String
+ getADSName() : String
+ getAOCLName() : String
+ getAOCLDomainName() : String
-classLoader
#ads
AspectDeploymentScope
# name: String
+ getAspectDeploymentScopeFor(aspectId :String) : AspectDeploymentScope
+ AspectDeploymentScope(name :String)
+ getName() : String
# addJarUrl(jarUrl :String) : void
+ getClassLoaderName() : String
+ getClassLoaderDomainName() : String
+ getAspectsIds() : List<String>
+ deployAspect(aspectClassName :String, aspectName :String, pointcutDefs :Map<String, String>, parameters :Map<String, String>) : String
+ undeployAspect(aspectId :String) : void
# prepareAspectDeploymentScope() : void
+ resolveADSPointcutExpression() : String
# prepareAspect(aspectObj :Object, aspectContext :AspectContext) : void
+ addAspectId(aspectId :String) : void
# prepareAspectContext(aspectContext :AspectContext) : void
StaticAspectDeploymentScope
+ StaticAspectDeploymentScope(name :String, pointcutExpression :String)
+ resolveADSPointcutExpression() : String
ParameterizedAspectDeploymentScope
# properties: Properties
+ ParameterizedAspectDeploymentScope(name :String, properties :Properties)
+ deployParameterizedAspect(aspectClassName :String, aspectName :String, pointcutParameters :Map<String, String>, parameters :Map<String, String>) : String
# resolveAspectPointcutExpressions(pointcutParameters :Map<String, String>) : Map<String, String>

Figure 8. Class diagram depicting relation between AOCL, `BytecodeProvider`, ADSs and generic aspect.

Likewise ADS, aspects may be static and parameterized as well. Parameterized ADS supports parameterized aspects while static ADS – static aspects.

In the case of static aspects pointcut are to be either:

- specified explicitly coded in aspect class annotations, or
- specified explicitly as parameters passed to the aspect deploy request

Parameterized aspects, in turn, have pointcuts resolved by ADS according to the parameter values:

- specified implicitly with parameters passed to the aspect deploy request while actual pointcut is resolved by parameterized ADS

Irrespectively, custom configuration parameters organized in key-value pairs can be passed to the aspect while deploying. Aspect deploy request is performed via ADS methods.

When `DeploymentScopeAwareAspect` interface is realized by concrete aspect class the reference to encompassing ADS will be injected to its instances via `setAspectDeploymentScope` method call just after aspect instantiation. Given with such a reference to ADS aspect may access ADS functionality e.g. may undeploy itself.

For developers' convenience generic class such as `UndeployableAspect` is provided that realizes `DeploymentScopeAwareAspect` interface and thanks to it may access basic information about itself. Aspect developer may use its functionality by simply extending this class.

Reconcilement of AOCL with AspectWerkz requires slight modification of AspectWerkz library. By default AspectWerkz instrumentation engine reads configuration related to given class loader when it is requested to load a first class. The configuration is extracted from XML document located in well-known location in local file system. This behavior is found not suitable for AOCL since AOCL is intended to be programmatically configurable and it is to store its configuration itself instead avoiding awkward manipulating of local file. Furthermore it is reasonable to read configuration from AOCL instance not necessarily when first class loading occurs but in arbitrary moment of the developer's choice. It especially applies to parameterized ADS classes which are to be loaded by AOCL before loading application classes and are not expected to be pre-instrumented. Therefore, the following modification are indispensable:

- AspectWerkz instrumentation engine has to distinguish load class request handled by AOCL and other class loaders,
- In the case of AOCL the configuration is to be read not from local file but from AOCL instance configuration,

– AOCL configuration has to be read in arbitrary moment of the developer's choice called activation moment: before activation instrumentation engine is not supposed do perform any instrumentation, activation causes reading configuration and after that all subsequently classes to load are properly pre-instrumented according to configuration provided.

5.1.2 Exposition Layer

The architecture of exposition layer is shown in Figure 9. Nomenclature of JMX layers and reference monitoring architecture layers may cause some confusion, so in Figure 9 JMX layers and monitoring system architecture layers are clearly decoupled.

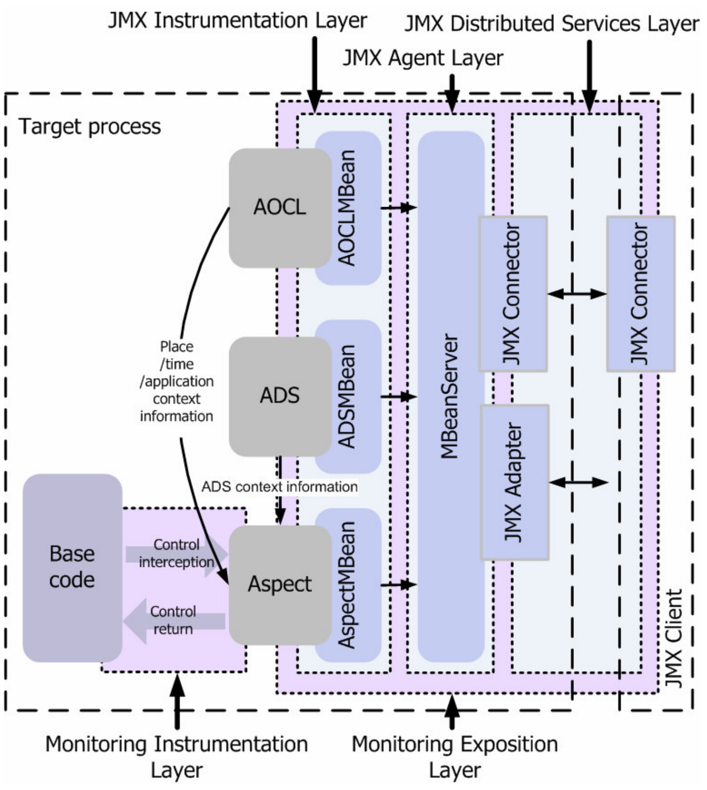


Figure 9. Architecture of Exposition Layer with JMX layers depicted and interactions with neighbor monitoring system layers.

Exposition layer in main part consists of JMX-enabled subclasses of instrumentation layer classes. Therefore derivatives such as `JMXAspectOrientedClassLoader`, `JMXStaticAspectDeploymentScope`, `JMXParameterizedAspectDeploymentScope` along with other helper class are present on this layer. Each of above listed classes are instrumented as standard MBeans, thus each of them has a corresponding MBean interface.

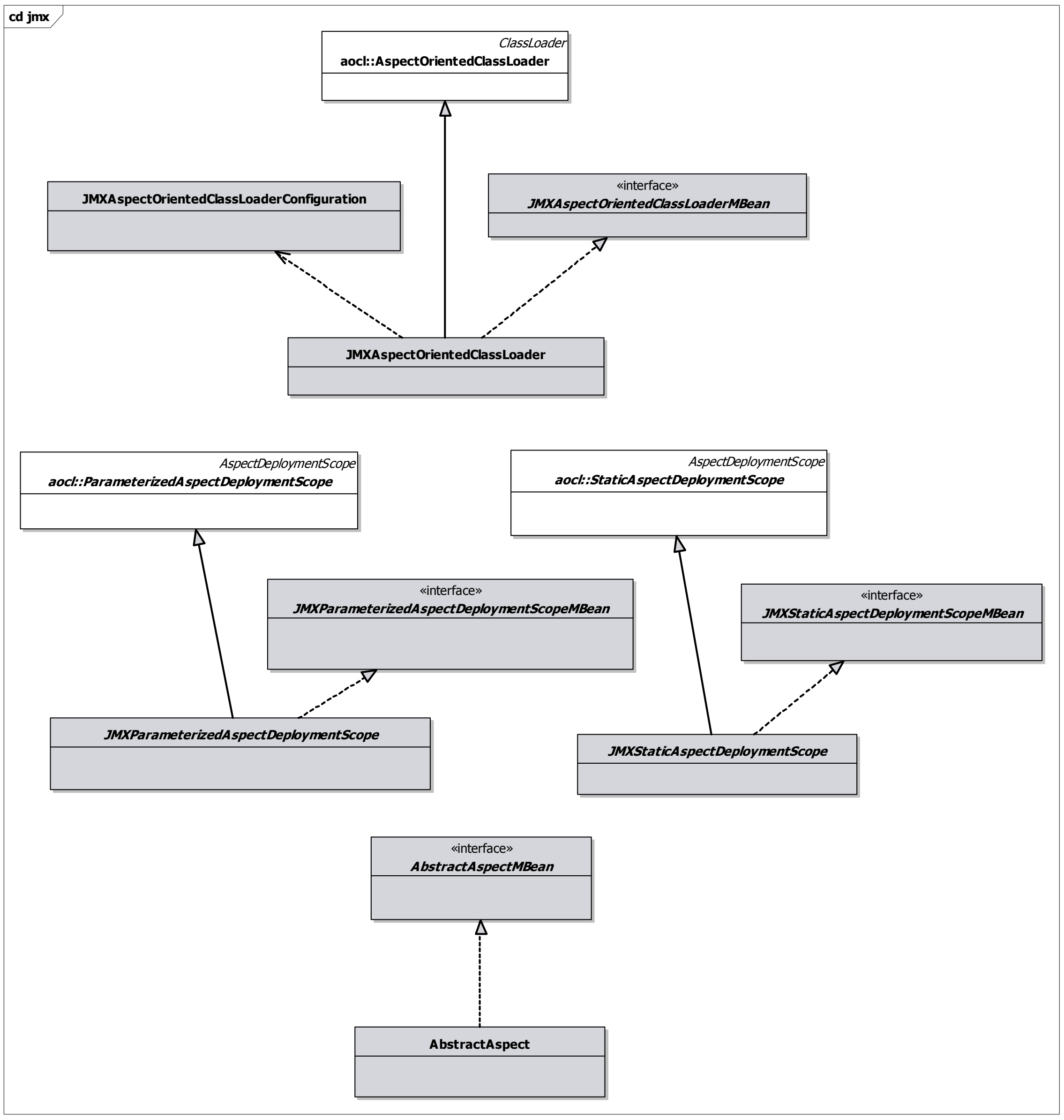

cd jmx
ClassLoader
aocl::AspectOrientedClassLoader
JMXAspectOrientedClassLoaderConfiguration
«interface»
JMXAspectOrientedClassLoaderMBean
JMXAspectOrientedClassLoader
AspectDeploymentScope
aocl::ParameterizedAspectDeploymentScope
AspectDeploymentScope
aocl::StaticAspectDeploymentScope
«interface»
JMXParameterizedAspectDeploymentScopeMBean
«interface»
JMXStaticAspectDeploymentScopeMBean
JMXParameterizedAspectDeploymentScope
JMXStaticAspectDeploymentScope
«interface»
AbstractAspectMBean
AbstractAspect

Figure 10. Core classes of JMX-enabled AOCL (marked as colored) as derivatives of corresponding AOCL classes (marked as white).

There is also provided `JMXAspectOrientedClassLoaderConfiguration` class (see: Figure 10) responsible for `JMXAspectOrientedClassLoader` initialization. Configuration may be specified via `JMXAspectOrientedClassLoaderConfiguration` methods or may be read form XML document such as presented in Code Snippet 1.

```
<?xml version="1.0" encoding="UTF-8"?>

<jmxaocl name="myapp" domainName="org.foo.myapp">

    <jarUrl>
            file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-sample-app.jar
    </jarUrl>

    <staticADS name="staticADS"
            pointcutExpression="execution(* java.lang.Runnable+.*(..))">

            <staticAspect name="staticAspect"
                    class="pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect8"
                    jar="file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-
sample-aspect.jar">
                    <pointcut name="myPointcut"
                            expression="execution(* java.lang.Runnable+.*(..))" />
                    <parameter name="author" value="me" />
                    <parameter name="date" value="today" />
            </staticAspect>

    </staticADS>

    <parameterizedADS name="parameterizedADS"
            class="ec.aocl.sample.scope.MethodTracerAspectDeploymentScope"
            jar="file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-sample-
scope.jar">
            <pointcutParameter name="packageScope"
                    value="pl.edu.agh.lemonade.aocl.sample.app" />

            <parameterizedAspect name="parameterizedAspect"
                    class="pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect7"
                    jar="file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-
sample-aspect.jar">
                    <pointcutParameter name="callerMethodName"
                            value="void
pl.edu.agh.lemonade.aocl.sample.app.Clock.run()" />
                    <pointcutParameter name="calleeMethodName"
                            value="void
pl.edu.agh.lemonade.aocl.sample.app.Clock.runInternal()" />
                    <parameter name="author" value="me" />
                    <parameter name="date" value="today" />
            </parameterizedAspect>

    </parameterizedADS>

</jmxaocl>
```

Code Snippet 1. Sample XML-based document expressing configuration of `JMXAspectOrientedClassLoader`.

Aspect developer is given with generic `AbstractAspect` class to extend. It provides basic functionality and exposes MBean interface of `AbstractAspectMBean`. Since `AbstractAspect` is a standard MBean is has to comply naming convention and specific inheritance pattern shown in Figure 11: concrete aspects that extends `AbstractAspect` has to be provided with corresponding MBean interface that extends abstract aspect's MBean interface.

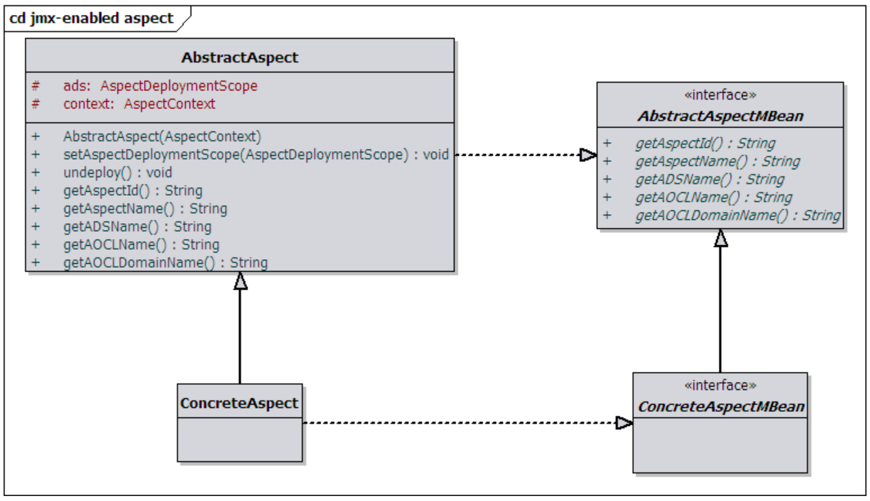


Figure 11. Aspect inheritance according to standard MBean convention.

5.1.3 Access Layer

As noticed earlier on the application manager's side AOCL Registry has to be maintained as long as it provides client stubs for AOCLs. Such a registry instances of `AOCLDescriptor` class, each per corresponding AOCL.

`AOCLDescriptor` contains information such as AOCL configuration, AOCL MBean's object name and address of MBeanServer where AOCL is being registered. On the other hand, `AOCLDescriptor` acts like factory which creates on demand instances of `AOCLMonitor`s.

`AOCLMonitor` is a local handle which maintains connection to the corresponding remote AOCL. A logical tree structure that AOCLs, ADSs, and aspects are arranged to is reflected in monitors structure. Thus `AOCLMonitors` keep references to corresponding `ADSMonitors` which, in turn, keeps references to `AspectMonitors` (see: Figure 12).

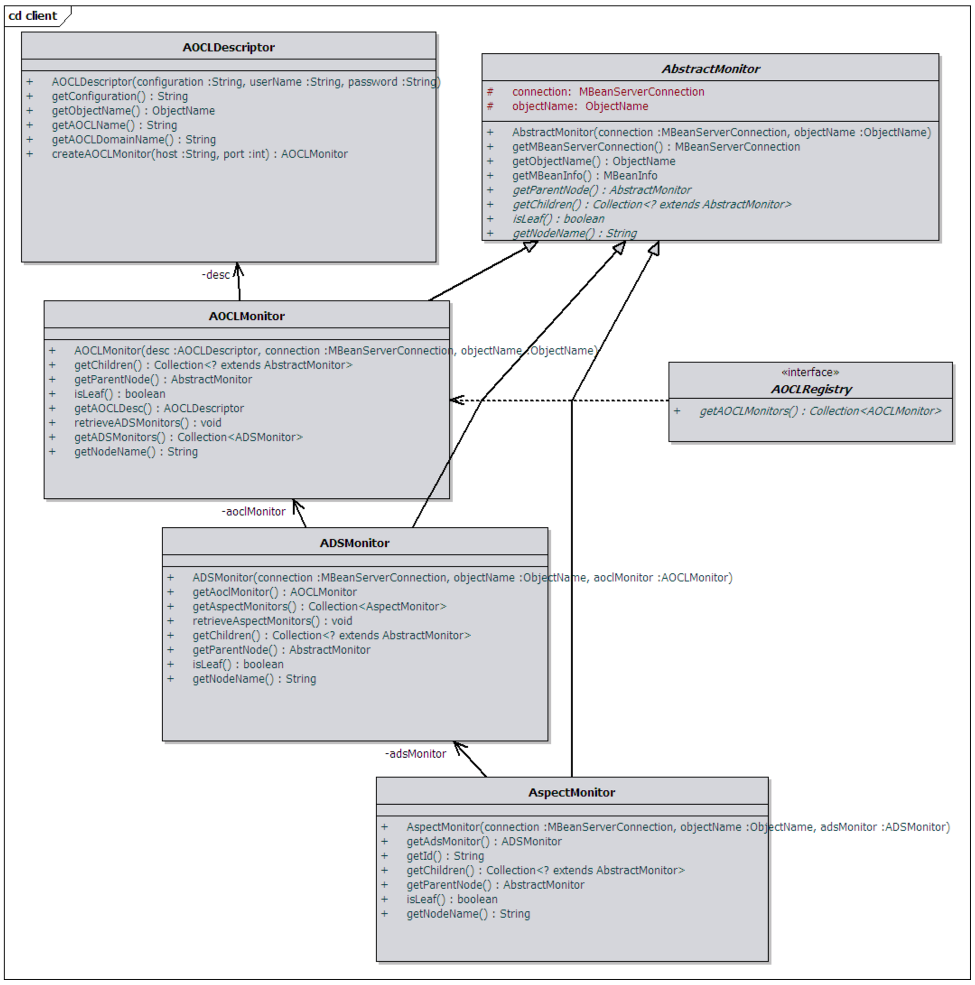

cd client
AOCLDescriptor
+ AOCLDescriptor(configuration :String, userName :String, password :String)
+ getConfiguration() : String
+ getObjectName() : ObjectName
+ getAOCLName() : String
+ getAOCLDomainName() : String
+ createAOCLMonitor(host :String, port :int) : AOCLMonitor
AbstractMonitor
# connection: MBeanServerConnection
# objectName: ObjectName
+ AbstractMonitor(connection :MBeanServerConnection, objectName :ObjectName)
+ getMBeanServerConnection() : MBeanServerConnection
+ getObjectName() : ObjectName
+ getMBeanInfo() : MBeanInfo
+ getParentNode() : AbstractMonitor
+ getChildren() : Collection<? extends AbstractMonitor>
+ isLeaf() : boolean
+ getNodeName() : String
-desc
AOCLMonitor
+ AOCLMonitor(desc :AOCLDescriptor, connection :MBeanServerConnection, objectName :ObjectName)
+ getChildren() : Collection<? extends AbstractMonitor>
+ getParentNode() : AbstractMonitor
+ isLeaf() : boolean
+ getAOCLDesc() : AOCLDescriptor
+ retrieveADSMonitors() : void
+ getADSMonitors() : Collection<ADSMonitor>
+ getNodeName() : String
«interface»
AOCLRegistry
+ getAOCLMonitors() : Collection<AOCLMonitor>
-aoclMonitor
ADSMonitor
+ ADSMonitor(connection :MBeanServerConnection, objectName :ObjectName, aoclMonitor :AOCLMonitor)
+ getAoclMonitor() : AOCLMonitor
+ getAspectMonitors() : Collection<AspectMonitor>
+ retrieveAspectMonitors() : void
+ getChildren() : Collection<? extends AbstractMonitor>
+ getParentNode() : AbstractMonitor
+ isLeaf() : boolean
+ getNodeName() : String
-adsMonitor
AspectMonitor
+ AspectMonitor(connection :MBeanServerConnection, objectName :ObjectName, adsMonitor :ADSMonitor)
+ getAdsMonitor() : ADSMonitor
+ getId() : String
+ getChildren() : Collection<? extends AbstractMonitor>
+ getParentNode() : AbstractMonitor
+ isLeaf() : boolean
+ getNodeName() : String

Figure 12. Diagram of classes involved in AOCL Registry.

5.2 *Mocca leMonAdE Edition*

Adaptation of Mocca-enabled H2O container is realized merely by providing additional JVM arguments required by AspectWerkz library at H2O startup. Mocca itself, was altered solely in `mocca.srv.impl.MoccaComponentPlugletImpl` class. The modification affects mainly initialization of component *pluglet*, so that AOCL configuration given as an initialization parameter is read and according to AOCL is then instantiated. Finally, AOCL is let to be the one to load component class. The extended class loader hierarchy is depicted on Figure 13. In this way, every component instance is associated with exactly one corresponding AOCL instance.

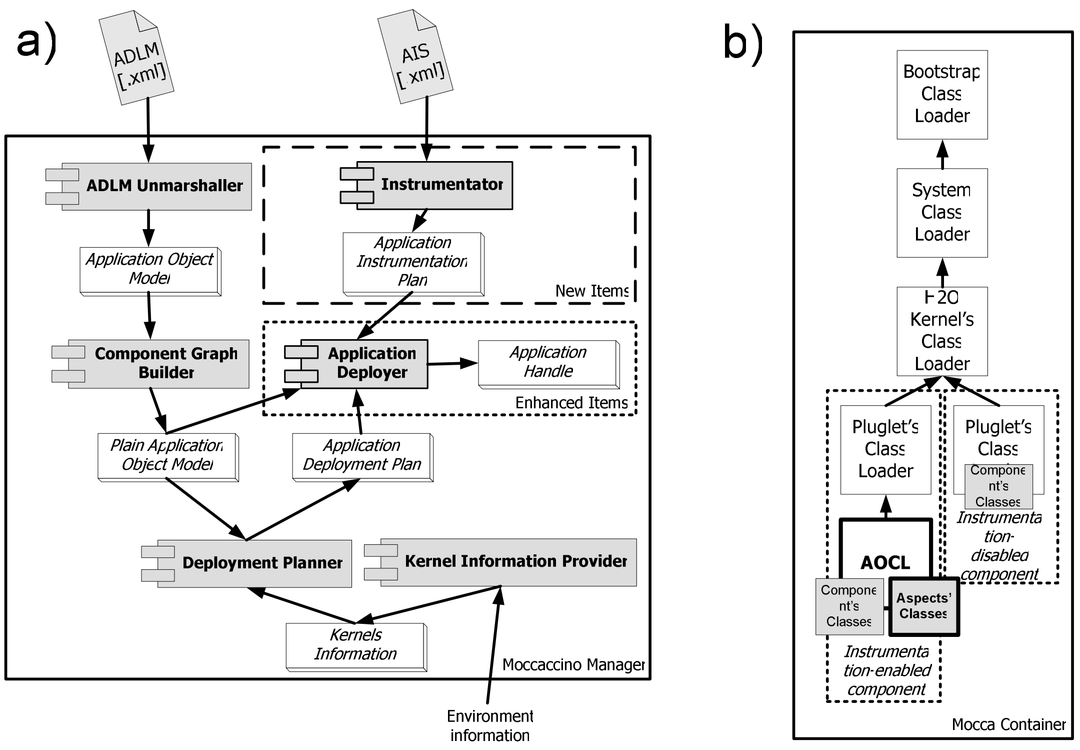


Figure 13. Inherited, added and modified items in Moccaccino Manager's data flow (a). Extended class loader hierarchy within H2O/Mocca container (b).

5.3 Moccaccino leMonAdE Edition

In Figure 13 (a) all the inherited, added and modified items in Moccaccino Manager's data flow are depicted. New *Instrumentator* module is to make an *Application Instrumentation Plan* from XML-based AISM language, that is used by *Deployer* module while deploying application. The result of deployment is *Application Handler* that is enriched with AOCL Registry that contain records for each AOCL involved in application run. The classes involved in application instrumentation are depicted in Figure 14.

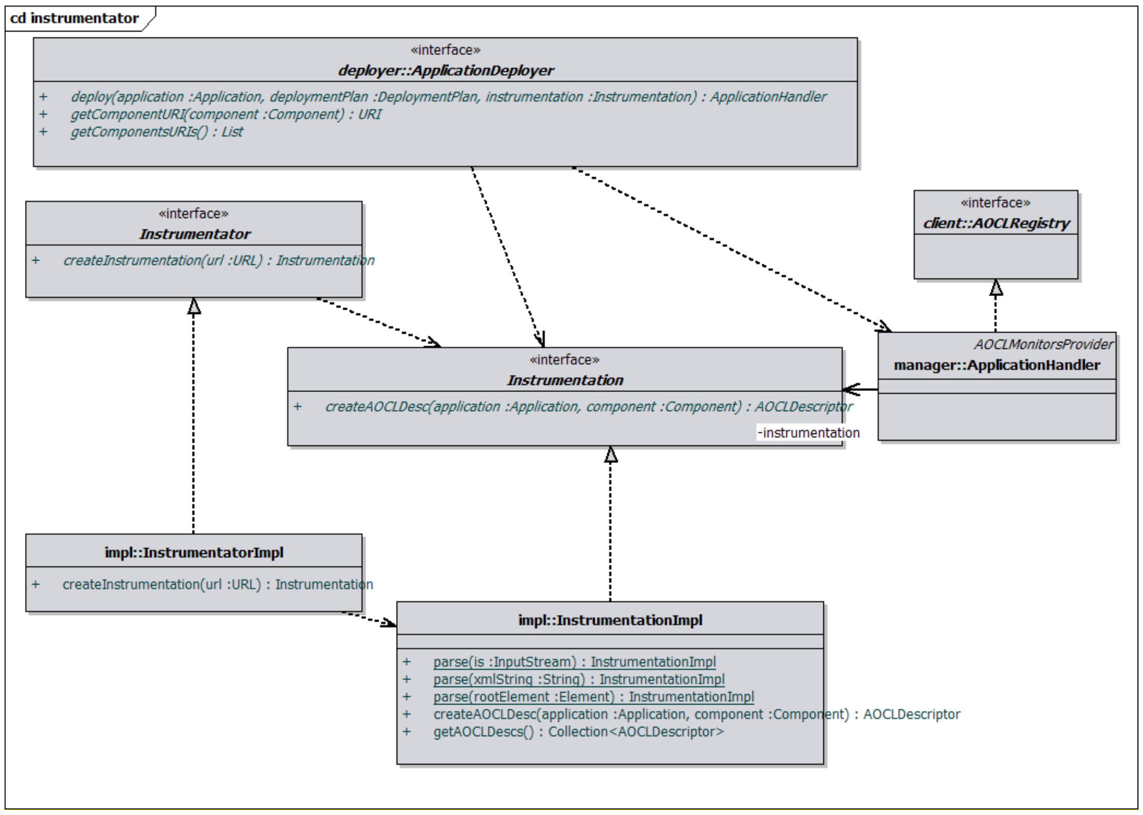

cd instrumentator
«interface»
deployer::ApplicationDeployer
+ deploy(application :Application, deploymentPlan :DeploymentPlan, instrumentation :Instrumentation) : ApplicationHandler
+ getComponentURI(component :Component) : URI
+ getComponentsURIs() : List
«interface»
Instrumentator
+ createInstrumentation(url :URL) : Instrumentation
«interface»
client::AOCLRegistry
«interface»
Instrumentation
+ createAOCLDesc(application :Application, component :Component) : AOCLDescriptor
-instrumentation
AOCLMonitorsProvider
manager::ApplicationHandler
impl::InstrumentatorImpl
+ createInstrumentation(url :URL) : Instrumentation
impl::InstrumentationImpl
+ parse(is :InputStream) : InstrumentationImpl
+ parse(xmlString :String) : InstrumentationImpl
+ parse(rootElement :Element) : InstrumentationImpl
+ createAOCLDesc(application :Application, component :Component) : AOCLDescriptor
+ getAOCLDescs() : Collection<AOCLDescriptor>

Figure 14. Classes involved in application instrumentation within Moccaccino leMonAdE edition.

AISM refers to component groups from ADLM and bind them with the AOCL configurations provided as it is presented on Figures Code Snippet 2 and Code Snippet 3.

```
<component-group name="ping" component-class="Ping" weight="5">
  <connection usesPort="pongs" qualifier-attribs="length=2"
   providesPort="PongPort" weight="5" shared="false">
    <component-group name="pongs" component-class="Pong" weight="5">
      <connection usesPort="zonks" qualifier-attribs="keys=one;two"
       providesPort="ZonkPort" weight="5" shared="false">
        <component-group name="zonks" component-class="Zonk" weight="5"/>
      </connection>
    </component-group>
  </connection>
</component-group>
```

Code Snippet 2. Sample application architecture expressed in ADLM. It represents three level tree-like structure of sub-workers.

```
<instrumentation userName="someone" password="secret">
  <componentGroupInstrumentation>

    <componentGroup name="pongs" />

    <parameterizedADS>. . .</parameterizedADS>
    <staticADS>. . .<staticADS>
    <jarUrl>. . .</jarUrl>
    <classpath>. . .</classpath>

  </componentGroupInstrumentation>
</instrumentation>
```

Code Snippet 3. Sample application instrumentation expressed in AIS as it refers to the ADLM application architecture presented on Figure Code Snippet 2 in order to bind *pongs* component group with given AOCL configuration. The inessential AOCL configuration details are omitted.

5.4 Tool Layer

As a sample monitoring tool prototype the Eclipse plug-in for monitoring-enabled Moccaccino manager has been developed. It is actually based on Eclipse-JMX [20] adapted for management of distributed MBeans registered in multiple MBeanServers.

As stated recently, access layer introduces `AOCLRegistry` that stores monitors associated with corresponding to AOCLs, ADSs and aspects. In UI layer these entities are easy to arrange in tree-like structure owing to `AbstractMonitor` class that is generalization of all above enumerated monitors as shown in Figure 15. Each `AbstractMonitor` has `MBeanServerConnection`, object name, `MBeanInfo`, as well as parent and children monitors. Such a data structure fully enabled monitoring and is easy to apply in custom graphical JMX console.

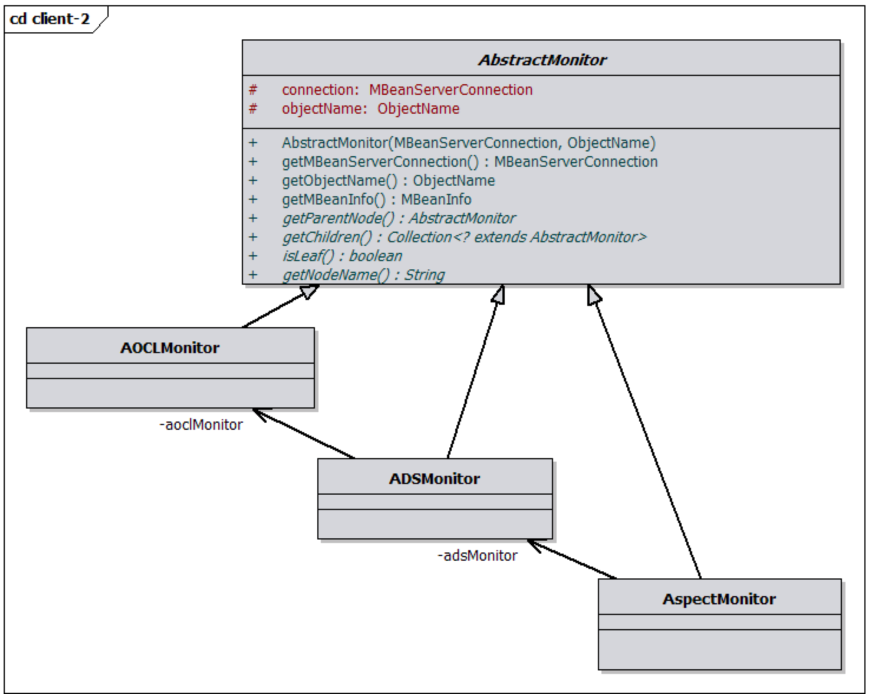
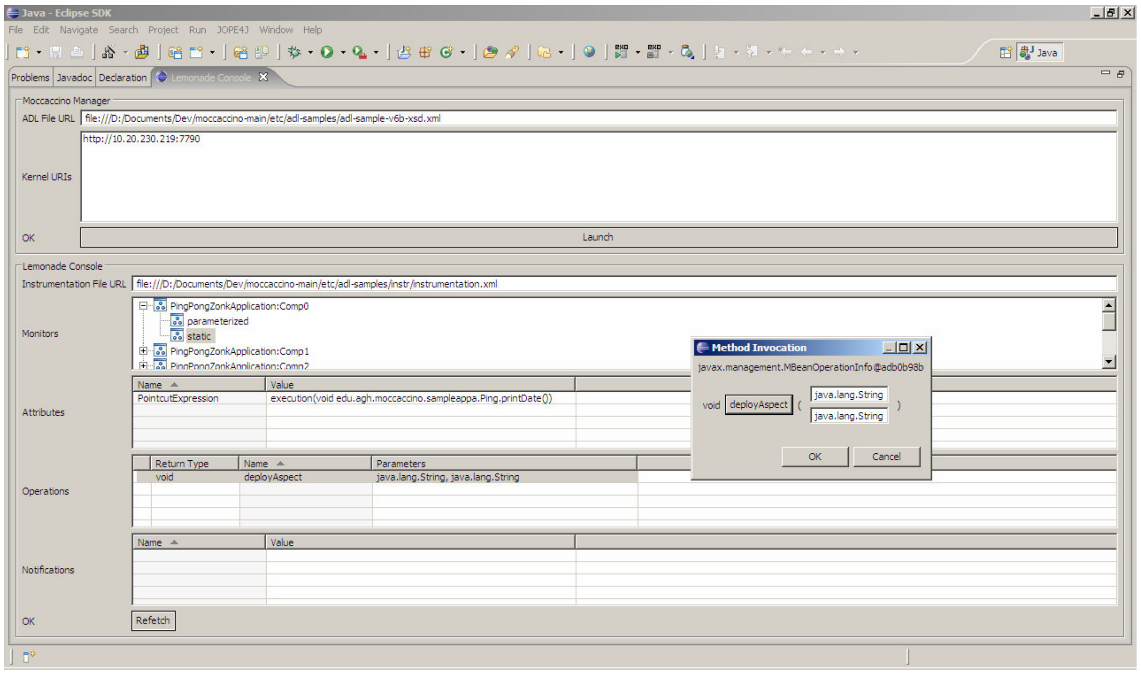

cd client-2
AbstractMonitor
# connection: MBeanServerConnection
# objectName: ObjectName
+ AbstractMonitor(MBeanServerConnection, ObjectName)
+ getMBeanServerConnection() : MBeanServerConnection
+ getObjectName() : ObjectName
+ getMBeanInfo() : MBeanInfo
+ getParentNode() : AbstractMonitor
+ getChildren() : Collection<? extends AbstractMonitor>
+ isLeaf() : boolean
+ getNodeName() : String
AOCLMonitor
-aoclMonitor
ADSMonitor
-adsMonitor
AspectMonitor

Figure 15. `AbstractMonitor` data structure to apply in graphical JMX consoles dedicated to AOCL registry.

Basing on `AbstractMonitor` structure and Eclipse-JMX console the Moccaccino leMonAdE edition Eclipse plug-in was developed. It is presented in Figure 16.


Java - Eclipse SDK
File Edit Navigate Search Project Run JOPE4J Window Help
Java
Problems Javadoc Declaration Lemonade Console

Moccaccino Manager

ADL File URL: file:///D:/Documents/Dev/moccaccino-main/etc/adl-samples/adl-sample-v6b-xsd.xml

Kernel URIs: http://10.20.230.219:7790

OK: Launch

Lemonade Console

Instrumentation File URL: file:///D:/Documents/Dev/moccaccino-main/etc/adl-samples/instr/instrumentation.xml

Monitors:
- PingPongZonkApplication:Comp0
  - parameterized
  - static
- PingPongZonkApplication:Comp1
- PingPongZonkApplication:Comp2

Attributes:

Name	Value
PointcutExpression	execution(void edu.agh.moccaccino.sampleappa.Ping.printDate())

Operations:

Return Type	Name	Parameters
void	deployAspect	java.lang.String, java.lang.String

Notifications:

Name	Value

OK: Refetch

Method Invocation

javax.management.MBeanOperationInfo@adb0b98b

void deployAspect (java.lang.String java.lang.String)

OK Cancel

Figure 16. Screenshot of a Moccaccino leMonAdE edition Eclipse plug-in in work.

5.5 Implementation summary

The software developed in a scope of this work include four subprojects, all developed in pure Java programming language. Development process generated about 110 classes of 15 packages contained in 4 projects. The examples and sample application was developed as well. The main external technologies harnessed in the system was:

– AspectWerkz AOP library

– Java Management Extensions (JMX)

Moreover, realization of concept imposed the slight modification of AspectWerkz library. The overall number of 15 of external libraries was used.

Chapter 6 leMonAdE Monitoring System Performance Analysis

This chapter gives answers on how the solution developed meets the performance requirements. The following subsections explain tests methodology, discuss the results obtained and estimates usability of a solution.

The preliminary experiments with a prototype are intended to investigate the overhead introduced by leMonAdE instrumentation as compared to the source code instrumentation technique. The values to measure are the overhead introduced by AOCL itself, by bytecode instrumentation and by the notification emitted out. Moreover, not only instrumentation overhead is to be measured. Since dynamic aspect deployment is allowed only in pre-instrumented (at class load time) join-points, the impact of such a pre-instrumentation on bytecode performance has to be estimated. The detailed description of the test methodology is explained in section 6.1.

The test application was a simple stand-alone Java application performing millions of simple numerical floating-point or integer computations. The computations were arranged either in loops nested in recursive calls or in recursive calls nested in loops, in order to investigate how stack dynamicity affects performance overhead. Section 6.2 is devoted to deepened insight into the test application.

Performance test application along with performance test suites are incorporated into leMonAdE AOCL distribution so that developers can carry out tests out of a box in production environments on every platform. Moreover, there are dedicated Ant targets provided in order to facilitate test suite executions. Further details on how the test suites are designed are covered by section 6.3. Sample runs of the test suites that has been carried out are described in section 6.4 and discussed in section 6.5.

6.1 Test methodology

The subsequent experiment runs were carried out according to the following scheme (also shown in Figure 17):

– Test application with no instrumentation – base application (run I)

– Test application with source code instrumentation

  – with instrumentation not involving JMX notifications emitted out (run II)

  – with instrumentation involving JMX notification emitted out (run III)

– Test application with leMonAdE instrumentation

  – only with AOCL used as an application class loader (run IV)

  – with AOCL and pre-instrumentation of ADS (run V)

  – with AOCL, pre-instrumentation of ADS and instrumentation by an aspect (run VI)

  – with AOCL, pre-instrumentation of ADS and instrumentation by an aspect with JMX notification (run VII)

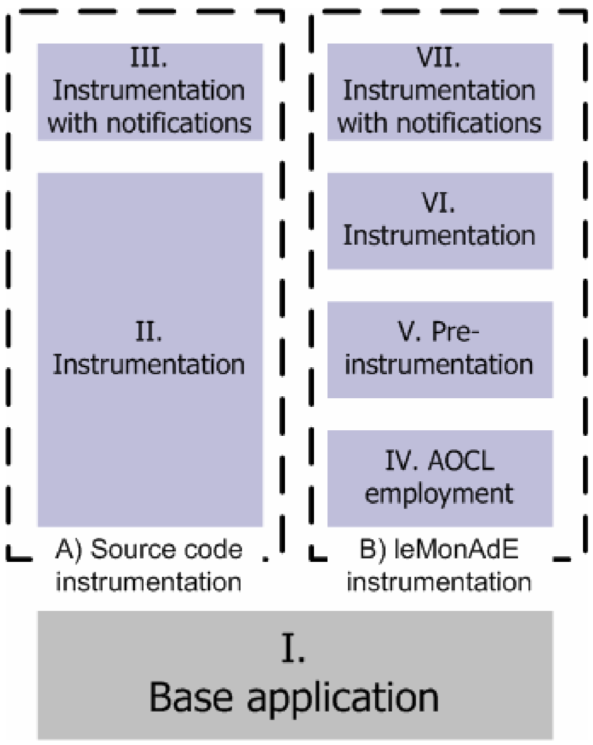

III. Instrumentation with notifications
II. Instrumentation
A) Source code instrumentation
VII. Instrumentation with notifications
VI. Instrumentation
V. Pre-instrumentation
IV. AOCL employment
B) leMonAdE instrumentation
I. Base application

Figure 17. Test runs scheme. Separate runs of base test application (I), application with source code instrumentation (A: II, III), with leMonAdE instrumentation (B: IV, V, VI, VII). Separate runs for instrumentation with (III, VII) and without notifications (II, IV, V, VI) emitted out.

Moreover, the test application is to perform variable number of computations in order to estimate linear approximation of overhead introduced. On the other hand, in order to find out whether and how stack dynamicity affects the overhead, the test application performs computations arranged in either:

– loops nested in recursive calls or

– recursive calls nested in loops

What is more, in the test application interception overhead has to be compared to the time consumed by sample standard computations such as simple floating-point computation (e.g. *sin(x)*) or integer computation (e.g. *50!*). Therefore, in the test application each call to the method performing one of the aforementioned computation are to be instrumented so that we have one interception per one computation such as *sin(x)* and *50!*.

Given with a test methodology a suitable test application is presented in the subsequent section.

6.2 Test application

The test application is a stand-alone Java application that performs simple computation, but arranged in a specific way. As shown in Code Snippet 4 the actual computation is performed in `instrumentedMethod()` body, while the rest of the methods just arrange computation either in loops nested in recursive calls or in recursive calls nested in loops.

```
doIterationsWithinRecursion(recursionTimes, iterationTimes) {
  if (recursionTimes > 0) {
    for (i = 0; i < iterationTimes; i++) instrumentedCall();
    if (recursionTimes > 1)
      doIterationsWithinRecursion(recursionTimes-1, iterationTimes);
  }
}
doRecursionWithinIterations(iterationTimes, recursionTimes) {
  for (i = 0; i < iterationTimes; i++) this.doRecursion(recurionTimes);
}
doRecursion(times) {
  if (times > 0) instrumentedCall();
  if (times > 1) this.doRecursion(times - 1);
}
instrumentedMethod() {
  simple-floating-point-or-integer-calculation;
}
```

Code Snippet 4. Test application pseudocode. Measured was a time consumed for calls `doIterationsWithinRecursion()` and `doRecursionWithinIterations()`.

Since the `instrumentedMethod()` is to be instrumented, in a source code instrumentation case its body will look like in Code Snippet 5. Instrumentation involves counting of calls to `instrumentedMethod()` and (additionally in the case of test run III) sending notification about count attribute changed.

```
instrumentedMethod() {
  // instrumentation goes here
  count++;

  // notification is sent below
  sendCountChangedNotification(count);

  simple-floating-point-or-integer-calculation;
}
```

Code Snippet 5. Test application pseudocode of method `instrumentedMethod()` with source code instrumentation that counts number of calls and sends JMX notification about count attribute changed. Test cases will measure instrumentation overhead with (run III) and without (run II) notifications sent.

As stated above, in the `instrumentedMethod()` body some simple computation are performed, either:

- floating-point computation of *sin(x)* or
- integer computation of *50!*

In the case of test run VI application is instrumented with an aspect shown in Code Snippet 6 that performs logically the same operations as source code instrumentation in the case of run II.

Similarly, in the case of test run V the aspect used that is shown in Code Snippet 7 is logically equivalent to the source code instrumentation of run III that is presented in Code Snippet 5.

```
@Aspect
public class PerformanceTestAppAspect implements PerformanceTestAppAspectMBean,
            DeploymentScopeAwareAspect {
    private int count = 0;

    @Around("execution(void
pl.edu.agh.lemonade.aocl.sample.app.PerformanceTestApp.doSth())")
    public Object monitor(StaticJoinPoint jp) throws Throwable {
            this.count++;
            return jp.proceed();
    }

    public int getCount() {
            return this.count;
    }
}
```

Code Snippet 6. Aspect that instruments the sample application in the case of test run VI – without JMX notifications sent.

```
@Aspect
public class PerformanceTestAppAspect2 extends NotificationBroadcasterSupport
            implements PerformanceTestAppAspect2MBean, DeploymentScopeAwareAspect,
            NotificationBroadcaster {
    private int count = 0;

    @Around("execution(void
pl.edu.agh.lemonade.aocl.sample.app.PerformanceTestApp.doSth())")
    public Object monitor(StaticJoinPoint jp) throws Throwable {
            this.count++;
            Notification not = new AttributeChangeNotification(this, this.count,
                           System.currentTimeMillis(), "", "count", "int",
this.count - 1,
                           this.count);
            this.sendNotification(not);
            return jp.proceed();
    }

    public int getCount() {
            return this.count;
    }
}
```

Code Snippet 7. Aspect that instruments the sample application in the case of test run VII – with JMX attribute changed notifications.

In order to facilitate launching of the test application with accordance to the test methodology dedicated test suites were developed and are presented in the following section.

6.3 Test suites

The test suites are included in the leMonAdE distribution and follow the above assumed methodology. They organize test application runs as it is shown in pseudocode in Code Snippet 8.

There are two complementary suites:

– First, performing test run I where the base application is run, and test runs II-III where the application is run with source code instrumentation

– Second, performing test runs IV-VII involving leMonAdE instrumentation.

For both of the above mentioned two dedicated ant targets shown in Code Snippet 9 are included in `build.xml` file.

```
for each computation times in (9 000 000, 10 525 500, 12 250 000, 14 062 500,
16 000 000) do
  for each computation type in (sin(x), 50!) do
    for each application run in (I, II, III, IV, V, VI, VII) do
      for each computation stack characteristics in (iterations within recurion,
recursion within iterations) do
            run sample application(computation times, computation type,
application run, computation stack characteristics)
```

Code Snippet 8. Test suite pseudocode.

```
    <target name="ptest0" depends="jar">
            <java
classname="pl.edu.agh.lemonade.aocl.sample.start.PerformanceTestSuiteNoAOCL"
fork="true">
                    <classpath location="${lemonade.aocl.jar.name}" />
                    <classpath location="${lemonade.sample.app.jar.name}" />
            </java>
     </target>

     <target name="ptest1" depends="jar">
            <java
classname="pl.edu.agh.lemonade.aocl.sample.start.PerformanceTestSuiteAOCL"
fork="true">
                    <classpath refid="lemonade.aocl.run.classpath" />
                    <jvmarg value="-javaagent:${lemonade.aocl.jar.name}" />
                    <jvmarg value="-Dcom.sun.management.jmxremote.port=1234" />
                    <jvmarg value="-
Dcom.sun.management.jmxremote.authenticate=false" />
                    <jvmarg value="-Dcom.sun.management.jmxremote.ssl=false" />
            </java>
     </target>
```

Code Snippet 9. The ant targets dedicated to launch performance test suites: `ptest0` target dedicated to suit that cover test runs I-III, `ptest1` target dedicated to suit that cover test runs IV-VII.

The above described suites are ready to run evermore, in each and every environment. The following section provides sample results obtained in one of the tests that have been performed.

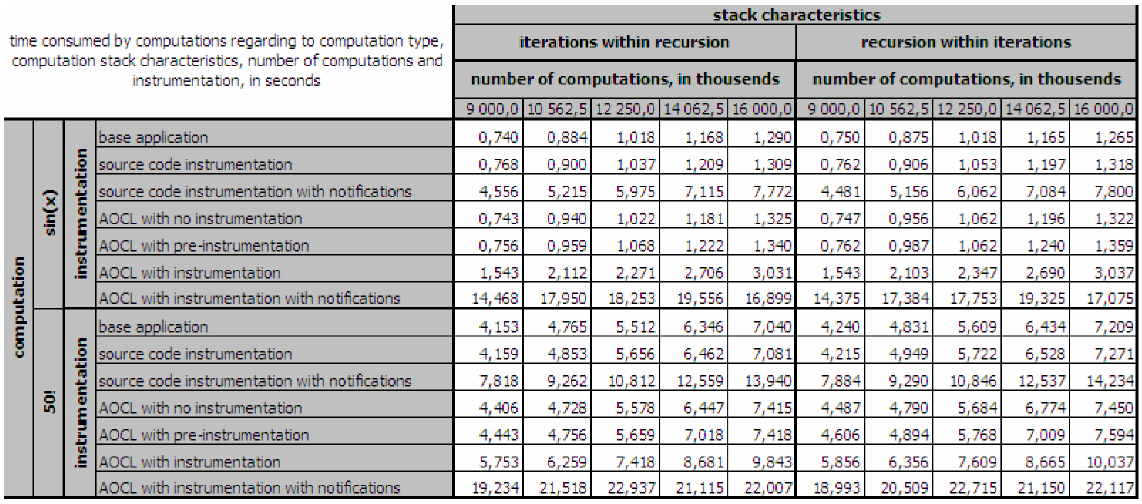
6.4 Sample results

This section is devoted to a test that was carried out with Java HotSpot Client VM (build 1.5.0_11-b03, mixed mode) on PC-class node:

– Processor Intel Pentium M 1.73 GHz, 2MB cache

– RAM 1GB 795 MHz

Both complementary test suites were launched 5 times each, in order to obtain complete result set that was subsequently put into tables and plotted as presented in Figure 18.



time consumed by computations regarding to computation type, computation stack characteristics, number of computations and instrumentation, in seconds				stack characteristics									
				iterations within recursion					recursion within iterations				
				number of computations, in thousends					number of computations, in thousends				
				9 000,0	10 562,5	12 250,0	14 062,5	16 000,0	9 000,0	10 562,5	12 250,0	14 062,5	16 000,0
computation	sin(x)	instrumentation	base application	0,740	0,884	1,018	1,168	1,290	0,750	0,875	1,018	1,165	1,265
			source code instrumentation	0,768	0,900	1,037	1,209	1,309	0,762	0,906	1,053	1,197	1,318
			source code instrumentation with notifications	4,556	5,215	5,975	7,115	7,772	4,481	5,156	6,062	7,084	7,800
			AOCL with no instrumentation	0,743	0,940	1,022	1,181	1,325	0,747	0,956	1,062	1,196	1,322
			AOCL with pre-instrumentation	0,756	0,959	1,068	1,222	1,340	0,762	0,987	1,062	1,240	1,359
			AOCL with instrumentation	1,543	2,112	2,271	2,706	3,031	1,543	2,103	2,347	2,690	3,037
			AOCL with instrumentation with notifications	14,468	17,950	18,253	19,556	16,899	14,375	17,384	17,753	19,325	17,075
	50!	instrumentation	base application	4,153	4,765	5,512	6,346	7,040	4,240	4,831	5,609	6,434	7,209
			source code instrumentation	4,159	4,853	5,656	6,462	7,081	4,215	4,949	5,722	6,528	7,271
			source code instrumentation with notifications	7,818	9,262	10,812	12,559	13,940	7,884	9,290	10,846	12,537	14,234
			AOCL with no instrumentation	4,406	4,728	5,578	6,447	7,415	4,487	4,790	5,684	6,774	7,450
			AOCL with pre-instrumentation	4,443	4,756	5,659	7,018	7,418	4,606	4,894	5,768	7,009	7,594
			AOCL with instrumentation	5,753	6,259	7,418	8,681	9,843	5,856	6,356	7,609	8,665	10,037
			AOCL with instrumentation with notifications	19,234	21,518	22,937	21,115	22,007	18,993	20,509	22,715	21,150	22,117

Table 5. Time consumed by computations regarding to computation type, computation stack characteristics, number of computations and instrumentation, in seconds.

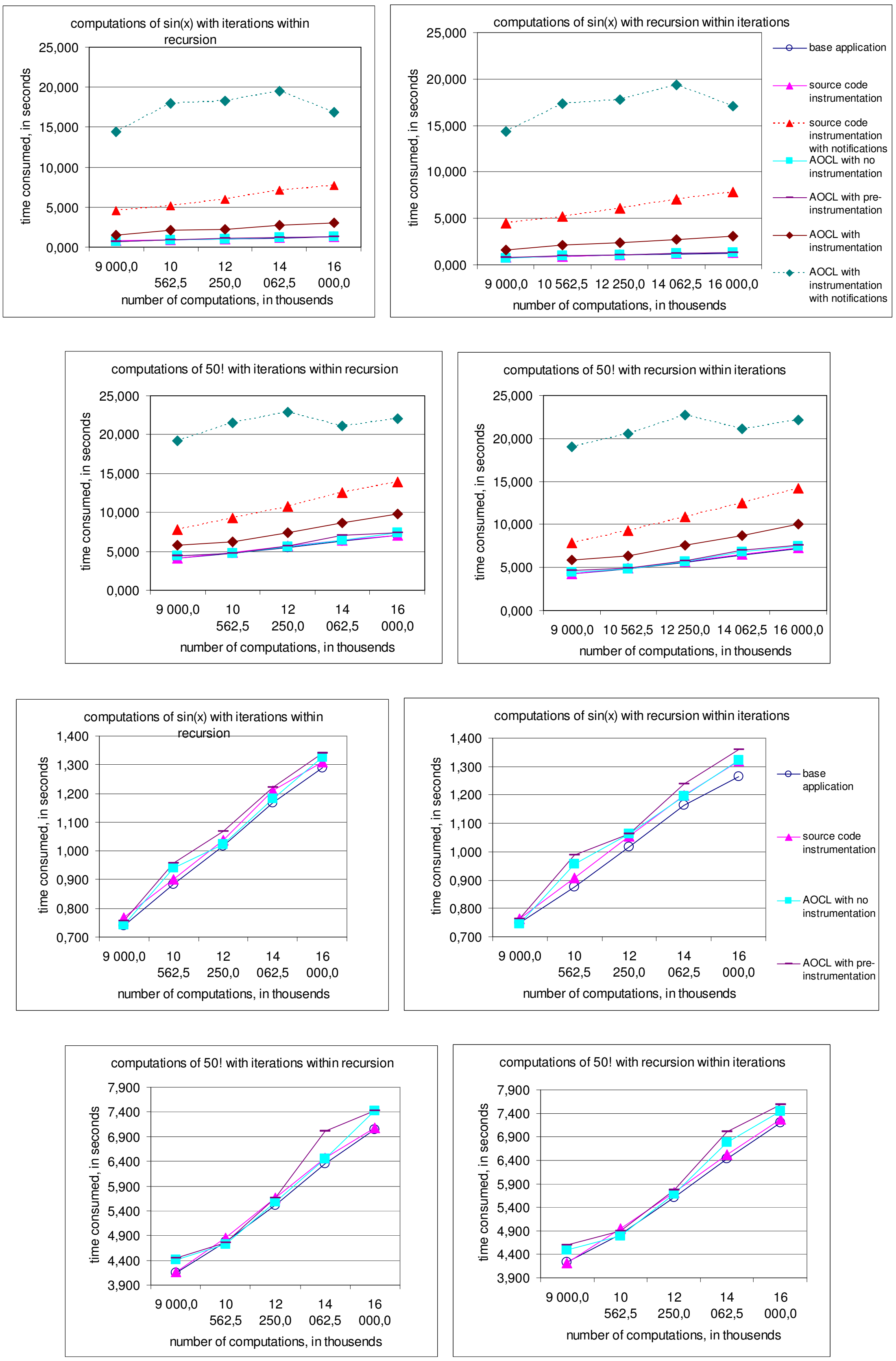


Figure 18. Sample test cases run results served in a form of charts: four upper charts plot results from each run (I-VI) for each computation type and stack characteristics, next four in below visualize the same results but are focused on runs that do not involve notifications (I-II, IV-VII) for the readability reasons.

6.5 *Analysis of results*

The results obtained in the previous section and collected in Table 5 gave a view of how leMonAdE affects performance of application. A difference between time consumed by instrumented application in runs II-VII and corresponding base application run I is considered as the overhead introduced by instrumentation. Given with results taken from Table 5 the differences were calculated and put into Table 6.

overhead regarding to computation type, computation stack characteristics, number of computations and instrumentation, in seconds				stack characteristics									
				iterations within recursion					recursion within iterations				
				number of computations, in thousends					number of computations, in thousends				
				9 000,0	10 562,5	12 250,0	14 062,5	16 000,0	9 000,0	10 562,5	12 250,0	14 062,5	16 000,0
computation	sin(x)	instrumentation	base application	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000
			source code instrumentation	0,028	0,016	0,019	0,041	0,019	0,012	0,031	0,035	0,032	0,053
			source code instrumentation with notifications	3,816	4,331	4,957	5,947	6,482	3,731	4,281	5,044	5,919	6,535
			AOCL with no instrumentation	0,003	0,056	0,004	0,013	0,035	-0,003	0,081	0,044	0,031	0,057
			AOCL with pre-instrumentation	0,016	0,075	0,050	0,054	0,050	0,012	0,112	0,044	0,075	0,094
			AOCL with instrumentation	0,803	1,228	1,253	1,538	1,741	0,793	1,228	1,329	1,525	1,772
			AOCL with instrumentation with notifications	13,728	17,066	17,235	18,388	15,609	13,625	16,509	16,735	18,160	15,810
	50!	instrumentation	base application	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000
			source code instrumentation	0,006	0,088	0,144	0,116	0,041	-0,025	0,118	0,113	0,094	0,062
			source code instrumentation with notifications	3,665	4,497	5,300	6,213	6,900	3,644	4,459	5,237	6,103	7,025
			AOCL with no instrumentation	0,253	-0,037	0,066	0,101	0,375	0,247	-0,041	0,075	0,340	0,241
			AOCL with pre-instrumentation	0,290	-0,009	0,147	0,672	0,378	0,366	0,063	0,159	0,575	0,385
			AOCL with instrumentation	1,600	1,494	1,906	2,335	2,803	1,616	1,525	2,000	2,231	2,828
			AOCL with instrumentation with notifications	15,081	16,753	17,425	14,769	14,967	14,753	15,678	17,106	14,716	14,908

Table 6. Overhead regarding to computation type, computation stack characteristics, number of computations and instrumentation, in seconds.

Assuming that the overhead of each application run is linear with respect to computation times, the *a* and b factors of *f(x)=ax+b* relation were estimated and contained in Table 7.

overhead regarding to computation type, computation stack characteristics, number of computations and instrumentation, in microseconds				stack characteristics			
				recursion within iteration		interations within recursion	
				linear approximation (y=ax+b)		linear approximation (y=ax+b)	
				b	a	b	a
computation	sin(x)	instrumentation	base application	0.000	**0.000**	0.000	**0.000**
			source code instrumentation	19.684	**0.000**	-25.913	**0.005**
			source code instrumentation with notifications	191.945	**0.397**	-18.975	**0.414**
			AOCL with no instrumentation	7.402	**0.001**	-5.092	**0.004**
			AOCL with pre-instrumentation	18.269	**0.002**	-20.659	**0.007**
			AOCL with instrumentation	-225.820	**0.124**	-255.281	**0.128**
			AOCL with instrumentation with notifications	**13100.108**	0.267	**12151.355**	0.325
	50!	instrumentation	base application	0.000	**0.000**	0.000	**0.000**
			source code instrumentation	19.857	**0.005**	-24.306	**0.008**
			source code instrumentation with notifications	-458.815	**0.467**	-647.661	**0.480**
			AOCL with no instrumentation	-144.770	**0.024**	-101.356	**0.022**
			AOCL with pre-instrumentation	-318.896	**0.050**	-92.400	**0.032**
			AOCL with instrumentation	-293.770	**0.188**	-197.950	**0.181**
			AOCL with instrumentation with notifications	**17518.630**	-0.139	**16025.645**	-0.048

Table 7. Linear approximation of a dependency between number of computations and overhead introduced, factor *a* is measured in seconds per thousand computations, factor b is measured in seconds.

The result obtained may be concluded as follows:

– In the case of instrumentation without notification the overhead introduced was expected to be independent of stack characteristics due to JVM feature of inliling of method calls introduced by aspects. It was confirmed by the measurement as can be seen in column 2 and 4 of Table 7.

– The overhead introduced by instrumentation was expected to be independent of the base application type due to fact that there is the same instrumentation in both cases. The accuracy of experiment does not allow to definitely confirm this expectation.

– Pre-instrumentation overhead is very similar to the overhead of source code instrumentation (again, due to inlining).

– Pre-instrumentation overhead varies between 0,002 and 0,05 microseconds per interception, and does not induce noticeable overhead to the application execution time.

– Instrumentation with aspects induces overhead which is 2 orders of magnitude greater than in the case of source code instrumentation. Additional time is consumed by AspectWerkz library when the advice is executed since it involves instantiation of JoinPoint object, calling advice method, which in turn calls instrumented method using refection API.

– In the case of aspect-based instrumentation with notifications the major part of overhead is introduced by JMX itself.

– The overhead introduced by aspect-based instrumentation with notifications is about 1,5-3 times grater than in the corresponding case of source code instrumentation with notification.

– Replacing ordinary class loader with AOCL induces negligible overhead

– Instrumentation with aspects with notifications in the function of computation times manifests large fluctuations so linear approximation does not apply in the measured range. It is more reasonable to extract an average of about 1 microsecond per interception as an approximation of the overhead. It may be implied by JMX policy in delivering notifications (e.g. buffering).

The results obtained proved that the pre-instrumentation overhead is insignificant and is optimized by JVM. In exchange the application is enabled to be dynamically instrumented with pluggable aspects. The overhead of simple instrumentation without notifications is 2 orders of magnitude greater than in the case of source code instrumentation, however does not require stopping application and recompilation as in the case of source code instrumentation.

Moreover, once deployed, aspect may be undeployed so it no longer causes the instrumentation overhead. Notification sent by aspect-based instrumentation causes overhead 1,5-2,5 times greater than in the case of source code instrumentation, although since aspect may store its state it is enabled to emit coarse-grained notifications, or since it is an MBean it may switch to the request-response on demand information exchange mode.

It is also important to note that the overhead introduced by aspect instrumentation with notifications remains at the order of microseconds which become negligible after introducing network layer where latency is at the order of milliseconds.

The test conducted can be regarded as the “worst case” since in the simple application all of operations were instrumented. In real life scenarios the ratio between instrumented code will be orders of magnitude lower resulting in significantly smaller overhead to the application execution time.

Chapter 7
Summary and Future Work

The main goal of this work which was development of a system for a monitoring of component-based application was successfully achieved. Since this thesis was set in some technological context it addressed Common Component Architecture, Java platform, H2O and Mocca frameworks and Moccaccino Manager tool. Nevertheless, a monitoring system turns out, in its major part, generic and applicable to the wider scope of Java-based application.

The monitoring system concept emerged from analysis of current state of the art. The ideas were addressing specific issues and were realized by employing suitable libraries and techniques available. Design and implementation managed to realize the concept. The system accomplished the requirements previously identified:

– **Separation of business logic and monitoring concern** was achieved by taking advantage of AOP paradigm that is implemented by AspectWerkz library. As long as AspectWerkz employs bytecode instrumentation it overcomes inconvenience of source code instrumentation.

– **Adherence to existing standards and specifications** was achieved by addressing standardized Java platform extension points such as custom class loaders support or JMX specification. The bytecode instrumentation relies on JVM specification. Also the component-based application model CCA which is addressed by this thesis is a standard specification.

– **Introspective monitoring** is supported by aspect-based instrumentation that involves advices which are executed during application run in the places called join-points. Subsets of join points of the developer's choice may be selected using dedicated robust pointcut expression language that is tailored for Java programming language.

- **Monitoring of high-level business logic** was enabled thanks to aspect-based instrumentation. As long as aspects are plain Java objects and have access to the application state, they are stateful and may be programmed with some business logic that feeds monitoring tools with information at the higher level of abstraction.
- **Dynamic instrumentation** was achieved by taking advantage Java 5.0 capabilities of dynamic redefinition of bytecode at the runtime.
- **Agile adaptation** of the applications to a monitoring system came true, since neither derangement of application source code nor recompilation is needed. The monitoring concert is encapsulated in external code of aspects, and the join-points are specified either within aspect code or are specified programmatically through AOCL API.
- **Monitored application model** was developed. Moccaccino Manager defines its Architecture Description Language for Moccaccino that fully describes the architecture of a CCA-based application.
- **Minimizing overhead** was achieved to some extent, as measured in performance tests.
- **Security,** however not covered by this thesis actually was taken into consideration on the design stage of the monitoring system.

The test carried out proved usability of a system developed. The leMonAdE monitoring system has provided robust and universal solution on instrumentation, exposition and access layers of a reference architecture of a monitoring system.

As long as this work was not focused on providing efficient and scalable monitoring infrastructure, the integration with existing generic monitoring infrastructure such as Gemini [9] is scheduled for future work. Integration with an abstract data bus would make a solution complete and to significant extent universal.

Future plans include also addressing security issue of authorized access to monitoring information basing on Shibboleth [44] solution. Especially, since all employed techniques are enabled to be secured by some security policy. Shibboleth's decentralized security model seems to fit well in the distributed nature of a monitoring of component-based application.

The continuous development direction of the leMonAdE monitoring system involves creating of monitoring aspects organized in toolkits that will support developers and deployers of Mocca application and any other Java-based applications. The toolkits may be dedicated to developers at the stage of testing, verifying and debugging of applications as well as monitoring of application deployed in runtime environment.

Since applicability scope of the system developed enables it to apply to other Java-based application frameworks, another work left for future is to make frameworks such as Web Services and Web Services Resource Framework leMonAdE-enabled. leMonAdE monitoring system is to be applied in Virolab project as a system capable of monitoring Java-based middleware technologies supported by GridSpace [21] such as Mocca, Axis, XFire and WSRF.

Abbreviations

(in alphabetical order)

ADL – Architecture Description Language

ADLM – Architecture Description Language for Moccaccino

ADS – Aspect Deployment Scope

AISM – Application Instrumentation Specification for Moccaccino

AOCL – Aspect-Oriented Class Loader

AOP – Aspect-Oriented Programming

AST – Abstract Syntax Tree

BCEL – Bytecode Engineering Library

CCA – Common Component Architecture

CORBA – Common Object Request Broker Architecture

HTTP – Hypertext Transfer Protocol

IDE – Integrated Development Environment

JMS – Java Message Service

JMX – Java Management Extensions

JPDA – Java Platform Debugging Architecture

JPM – Join-Point Model

JVM –Java Virtual Machine

LAN – Local Area Network

leMonAdE – Agile Monitoring Adherence Environment

MBean – Managed Bean

MIR – Monitoring Instrumentation Request

MOM – Message-Oriented Middleware

OOP – Object-Oriented Programming

RMI – Remote Method Invocation

RMIX – Remote Method Invocation Extensions

RPC – Remote Procedure Call

SIR – Standard Intermediate Representation

SNMP – Simple Network Management Protocol

SOAP – originally Simple Object Access Protocol, lately also Service-Oriented Architecture Protocol

TLA – Three Letter Acronym

UML – Unified Modeling Language

WAN – Wide Area Network

XML – Extensible Markup Language

References

[1] Armstrong, R., Kumfert, G., McInnes, L.C., Parker, S., Allan, B., Sottile, M., Epperly, T., Dahlgren, T.: The CCA component model for high-performance scientific computing. Concurr. Comput. : Pract. Exper. 18(2) (2006) 215–229

[2] Kurzyniec, D., Wrzosek, T., Drzewiecki, D., Sunderam, V.: Towards self-organizing distributedccomputing frameworks: The H2O approach. Parallel Processing Letters 13(2) (2003) 273–290

[3] Malawski, M., Kurzyniec, D., Sunderam, V.S.: Mocca - towards a distributed CCA framework for metacomputing. [18]

[4] Malawski, M., Bartynski, T., Ciepiela, E., Kocot, J., Pelczar, P., Bubak, M.: An ADL-based support for CCA components on the grid. In: CoreGRIDWorkshop on Grid Systems, Tools and Environments in Conjunction with GRIDS@work: CoreGRID Conference, Grid Plugtests and Contest, Sophia-Antipolis, France (2006)

[5] Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. Technical report, France Telecom (2002)

[6] Marques, M.: Exploring eclipse's AST parser. IBM Developer Works (2005)

[7] Fahringer, T., Gerndt, M., Li, T., Mohr, B., Seragiotto, C., Truong, H.L.: (Standardized intermediate representation for fortran, java, c and c++ programs)

[8] Davies, J., Huismans, N., Slaney, R.,Whiting, S.,Webster, M., Berry, R.: An aspect-oriented performance analysis environment. Technical report, IBM Corporation, Hursley, UK (2003)

[9] Truong, H.L., Balis, B., Bubak, M., Dziwisz, J., Fahringer, T., Hoheisel, A.: Towards distributed monitoring and performance analysis services in the K-WF Grid project. [19] 156–163

[10] Kersten, M.: Aopwork: AOP tools comparison. IBM Developer Works (2005) http://www-128.ibm.com/developerworks/library/j-aopwork1/.

[11] Architecture description languages http://www.sei.cmu.edu/architecture/adl.html

[12] ASM – Java bytecode manipulation framework home page http://asm.objectweb.org

[13] AspectJ – seamless aspect-oriented extension to the Java programming language home page http://www.eclipse.org/aspectj

[14] AspectWerkz – Plain Java AOP home page http://aspectwerkz.codehaus.org

[15] Byte Code Engineering Library home page http://jakarta.apache.org/bcel

[16] Bodkin, R.: Aopwork: Performance monitoring with aspectj. IBM Developer Works (2005) http://www-128.ibm.com/developerworks/java/library/j-aopwork10

[17] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S.R., McInnes, L.C., Parker, S.R., Smolinski, B.A.: Toward a common component architecture for high-performance scientific computing. In: HPDC. (1999)

[18] 19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CA, USA. In: IPDPS, IEEE Computer Society (2005)

[19] 19. Wyrzykowski, R., Dongarra, J., Meyer, N., Wasniewski, J., eds.: Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005, Revised Selected Papers. In Wyrzykowski, R., Dongarra, J., Meyer, N., Wasniewski, J., eds.: PPAM. Volume 3911 of Lecture Notes in Computer Science., Springer (2006)

[20] Eclipse-JMX plugin-in for eclipse IDE http://code.google.com/p/eclipse-jmx

[21] Gubała, T., Bubak, M.: Gridspace - semantic programming environment for the grid. [19] 172–179

[22] Jurczyk, P., Golenia, M., Malawski, M., Kurzyniec, D., Bubak, M., Sunderam, V.S.: A system for distributed computing based on H2O and JXTA. In: Cracow Grid Workshop, CGW'04, December 13–15, 2004, Kraków, Poland (2005) 257–268

[23] Kurzyniec, D., Wrzosek, T., Sunderam, V., Slomiński, A.: RMIX: A multiprotocol RMI framework for java. In: Proc. of the Intl. Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France, IEEE Computer Society (2003) 140–146

[24] H2O home page http://dcl.mathcs.emory.edu/h2o

[25] Iordanov, B.: Improve application management with JMX. www.ftponline.com (2004) http://www.ftponline.com/special/opsmgmt/iordanov/default.asp

[26] Javassist home page http://www.csg.is.titech.ac.jp/~chiba/javassist

[27] JBoss AOP – Framework for Organizing Cross Cutting Concerns home page http://labs.jboss.com/jbossaop

[28] Eclipse Java Development Tools (JDT) home page http://www.eclipse.org/jdt

[29] JGroups - A Toolkit for Reliable Multicast Communication home page http://www.jgroups.org/javagroupsnew/docs/index.html

[30] Java Message Service (JMS) home page http://java.sun.com/products/jms

[31] Java Management Extensions (JMX) Technology home page http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement

[32] Java Platform Debugger Architecture home page http://java.sun.com/javase/technologies/core/toolsapis/jpda

[33] The Java Virtual Machine Specification - Second Edition http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

[34] CollabNet, Inc.: JXTA Home Page (2005) http://www.jxta.org

[35] Java Sun Developer Network home page http://java.sun.com

[36] Krishnamurthy, R.: Performance analysis of j2ee applications using aop techniques. ONJava.com (2004) http://www.onjava.com/pub/a/onjava/2004/05/12/aop.html

[37] Kuleshov, E.: Using the ASM framework to implement common java bytecode transformation patterns. In on Aspect-Oriented Software Development, A.S.I.C., ed.: AOSD07 Sixth International Conference on Aspect-Oriented Software Development. (2007)

[38] Apache log4j – Logging Service home page http://logging.apache.org/log4j

[39] Fahringer, T., Gerndt, M., Li, T., Mohr, B., Seragiotto, C., Truong, H.L.: (Monitoring and instrumentation requests for Fortran, Java, C and C++ programs)

[40] Mcmanis, C.: The basics of java class loaders: The fundamentals of this key component of the java architecture. javaworld.com (1996) http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html

[41] Penchikala, S.: Add object cache monitoring using JMX and aspects. www.devx.com (2005) http://www.devx.com/Java/Article/29526/0/page/1

[42] Sun Microsystems, I.: Java Remote Method Invocation (2005) http://java.sun.com/products/jdk/rmi/

[43] Serp – open source framework for manipulating Java bytecode home page http://serp.sourceforge.net

[44] Shibboleth - standards-based, open source middleware software which provides Web Single Sign On (SSO) across or within organizational boundaries home page http://shibboleth.internet2.edu

[45] Sosnoski, D.: Classworking toolkit: ASM classworking. IBM Developer Works (2005) http://www.springframework.org/docs/reference/aop.html

[46] Swarr, R.: Make your apps operations friendly with AOP. www.ftponline.com (2004) http://www.ftponline.com/special/opsmgmt/swarr/default.asp.

[47] Autonomic Computing – IBM home page http://www-128.ibm.com/developerworks/autonomic

[48] Malawski, M., Bubak, M., Placek, M., Kurzyniec, D., Sunderam, V.: Experiments with distributed component computing across grid boundaries. In: Proceedings of HPCGECO/COMPFRAME Workshop in Conjunction with HPDC'06. (2006) 109–116

[49] Enterprise JavaBeans Technology home page http://java.sun.com/products/ejb

[50] JINI Specifications and API Archive home page http://java.sun.com/products/jini

[51] SOAP specification http://www.w3.org/TR/soap

[52] CORBA specification http://www.omg.org/technology/documents/formal/corba_2.htm

[53] Apache Commons Logging – Logging Services home page http://jakarta.apache.org/commons/logging

[54] Ruby home page http://www.ruby-lang.org/en

[55] JRuby – pure Java implementation of Ruby home page http://jruby.codehaus.org

[56] Hibernate - Relational Persistence for Java and .NET home page http://www.hibernate.org

[57] WS-Notification specification http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn

[58] Web Services Connector for JMX agents JSR-262
http://jcp.org/en/jsr/detail?id=262

[59] Moccaccino Manager project home site
(http://gforge.cyfronet.pl/projects/moccaccino)

Appendix A.
leMonAdE User Manual

This manual is dedicated to users of the leMonAdE monitoring system, namely software architects and programmers that want to incorporate it in large systems as well as in simple Java-based applications.

Sections in below are targeted to developers who are willing to take advantage of leMonAdE on whichever layer of a reference architecture of a monitoring system. This appendix is organized respectively to the leMonAdE layers and provides basis on how to employ it on each layer. The examples are presented and roughly explained. However, in order to get to the further details developers are supposed to refer to code documentation.

Thanks to leMonAdE's layered and modular architecture it is usable for variety of developers' aims. The foreseen application areas covered in this manual are:

– On **instrumentation layer** using Aspect-Oriented Class Loader

    – Dynamic deployment of bytecode

    – Aspect-Oriented Programming

        For Aspect-Oriented Class Loader manual please refer to the section **Using Aspect-Oriented Class Loader**.

– On **exposition layer** using JMX-enabled Aspect-Oriented Class Loader

    – Dynamic Programming

    – Monitoring and management of desktop applications

    – Remote monitoring and management

        For JMX-enabled Aspect-Oriented Class Loader manual please refer to the section **JMX-enabled Aspect-Oriented Class Loader**.

- On **access layer** using Mocca and Moccaccino leMonAdE editions
  - Monitoring and management of distributed applications
  - Supervising multiple simultaneous applications

For Mocca and Moccaccino leMonAdE editions manual please refer to the section **Using Mocca/Moccaccino leMonAdE-edition**.

Distribution

Before starting to work with leMonAdE the distribution has to be downloaded. The project site containing distribution and documentation is located at:

- http://gforge.cyfronet.pl/projects/lemonade

The current development version is available and is contained in SVN repository. Project can be checked out through anonymous access with the following command:

```
$ svn checkout https://gforge.cyfronet.pl/svn/lemonade
```

Moreover, a web interface of this repository is available though:

- http://gforge.cyfronet.pl/viewvc/?root=lemonade

Using Aspect-Oriented Class Loader

All the samples along with dedicated ant targets are incorporated into the distribution of leMonAdE AOCL. The package containing examples code is `pl.edu.agh.lemonade.aocl.sample`. Respective ant targets are included in `build.xml` script.

For presentation convenience, all of the examples attend extremely simple base application. It involves only one class, namely `pl.edu.agh.lemonade.aocl.sample.app.Clock` which code is shown in Code Snippet 10. It merely prints current date aligned with some given offset each second.

```
public class Clock implements Runnable {

    private long offset = 0;

    private long lastDate = 0;

    private int counter = 0;

    public void runInternal(){
           this.printDate();
           this.counter++;
           try {
                  Thread.sleep(1000);
           } catch (InterruptedException e) {
                  e.printStackTrace();
           }
    }

    public void run() {
           while (true) {
                  this.runInternal();
           }
    }

    public void addHoursOffset(int hours) {
           this.offset -= hours * 1000 * 60 * 60;
    }

    public long getTime() {
           this.lastDate = System.currentTimeMillis();
           return this.lastDate - this.offset;
    }

    private void printDate() {
           System.out.println(new Date(this.getTime()));
    }

    public static void main(String[] args) {
           new Clock().run();
    }

}
```

Code Snippet 10. Simple base application that the further examples attend.

The core of the instrumentation layer is `pl.edu.agh.lemonade.aocl.AspectOrientedClassLoader` class of leMonAdE AOCL project. It is also an entry point central class of instrumentation layer API. Following subsections are devoted to show the concrete examples involving AOCL and to present and explain its features.

Static Aspect Deployment Scope

Suppose we want to monitor base application by counting one of its method executions. For some purpose we also like to prevent from execution when some condition is fulfilled. For the example simplicity, let's assume, that every *n*-th execution is prevented.

The sample aspect below (Code Snippet 11) accomplish such a task. It is a pure AspectWerkz-like aspect code. The annotation-driven development was applied, thus `org.codehaus.aspectwerkz.annotation.Aspect` and `org.codehaus.aspectwerkz.annotation.Around` annotations are present. Therefore, no external aspect configuration is needed.

The aspect constructor takes special parameter of `org.codehaus.aspectwerkz.AspectContext` type however it is not necessary. Alternatively, the plain default constructor may be provided. `AspectContext` serves to pass some metadata information from `AspectOrientedClassLoader` that deploys aspects towards the aspect instance. In this example case the `blockFrequency` metadata object is passed. Moreover, it is supported to pass string key-value parameters to the aspect instance, e.g. aspect instance author.

The actual monitoring logic lies in advice method decorated with `Around` annotation, which denotes that such a method is a wrapper of target method execution (for annotations details refer to AspectWerkz documentation). Advice checks whether the execution is allowed and either let it run or not.

```
@Aspect
public class SampleAspect1 {

    private final int param;

    private final String author;

    private int count = 0;

    public SampleAspect1(AspectContext ctx) {
            this.param = (Integer) ctx.getMetaData("blockFrequency");
            this.author = ctx.getParameter("author");
    }

    @Around("execution(void
pl.edu.agh.lemonade.aocl.sample.app.Clock.printDate())")
    public Object monitor(StaticJoinPoint jp) throws Throwable {
            this.count++;
            System.out.println("before call #" + this.count + " by " + author);
            if (count % param != 0) {
                    return jp.proceed();
            } else {
                    System.out.println("call blocked");
                    return null;
            }
    }

}
```

Code Snippet 11. The sample aspect deployed onto base application that monitors method execution and prevents from unwished execution.

Having base application code as well as with aspect code, the `AspectOrientedClassLoader` has to be employed in order to reconcile those two.

The glue code (presented in Code Snippet 12) is responsible for instantiation of AOCL as a child of a given class loader (in most cases current class loader or system class loader) with a name and domain name provided. Then the locations of application code and aspect code (e.g. path to appropriate `jar` file or URL to `jar` file or local folder path) have to be added to AOCL. Since aspects exist in scope of Aspect Deployment Scope the proper has to be registered in AOCL. In this simple example we use static ADS named *ds* which pre-instruments all application class' methods execution. The static ADS defines some abstract callback methods, which developer is obliged to implement. It is worth noting that one of them prepares `AspectContext` instance which is to be passed to the aspect constructor, therefore, we can pass object to the aspect instance.

When all desired ADS are configured and added, AOCL has to be activated. From then on, every loaded class is pre-instrumented if it matches previously registered ADSs and new ADSs are no longer allowed to add.

Finally, the aspect itself may be deployed. If so, the string key-value parameters are specified, and along with aspect class name and aspect given name is passed as parameter of `deployAspect` method call. The last part of code is simply loading application main class by AOCL and calling its main method.

```
public class Sample1 {

    public static void main(String[] args) throws Exception {

            // AOCL instantiation with name MyAOCL and domain name org.foo
            AspectOrientedClassLoader aocl = new AspectOrientedClassLoader(
                           ClassLoader.getSystemClassLoader(), "MyAOCL",
"org.foo");

            // appending jar containing application code to the AOCL bytecode
            // provider
            aocl.addJar("./jar/lemonade-sample-app.jar");

            // appending jar containing aspect code to the AOCL bytecode provider
            aocl.addJar("./jar/lemonade-sample-aspect.jar");

            // static ADS instantiation with a pre-instrumentation pointcut
            // specified
            AspectDeploymentScope ads = new StaticAspectDeploymentScope("ds",
                           "execution(*
pl.edu.agh.lemonade.aocl.sample.app.Clock.*(..))") {

                    @Override
                    protected void prepareAspectDeploymentScope() {
                           // here the ADS preparation are to be performed
                           System.out.println("Preparing Aspect Deployment Scope");
                    }

                    @Override
                    public void prepareAspectContext(AspectContext aspectContext) {
                           // here the aspect context is to be prepared
                           // aspect context will be passed as an aspect
constructor
                           // parameter
                           System.out.println("Preparing Aspect Context");

                           // aspect context metadata enable passing objects to the
aspect
                           aspectContext.addMetaData("blockFrequency", 3);
                    }

                    @Override
                    protected void prepareAspect(Object aspectObj,
                                  AspectContext aspectContext) {
                           // here the aspect is to be prepared after its
instatiation
                           System.out.println("Preparing Aspect");
                    }

            };

            // all ADSs has to be added to the AOCL before its activation
            aocl.addAspectDeploymentScope(ads);

            // since AOCL is activated every loaded class is pre-intrumented if
            // needed
            aocl.activate();
            // parameters may be passed to the aspect while deploying
            Map<String, String> params = new HashMap<String, String>();
            params.put("author", "me");

            // deploying aspect with no pointcut definitions
            // with above constructed params

      ads.deployAspect("pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect1",
"myAspect",
                              null, params);

            // loading application main class and launching
            Runnable clock = (Runnable)
aocl.loadClass("pl.edu.agh.lemonade.aocl.sample.app.Clock")
                              .newInstance();
            new Thread(clock).start();
      }
}
```

Code Snippet 12. Sample usage of AOCL with simple desktop application involving static ADS and static aspect.

In order to run such a glue code it is indispensable to include external library dependencies in classpath. The required jar files are:

- `ant-1.5.2.jar`
- `asm-1.5.4-snapshot.jar`
- `asm-attrs-1.5.4-snapshot.jar`
- `asm-util-1.5.4-snapshot.jar`
- `concurrent-1.3.1.jar`
- `dom4j-1.4.jar`
- `jarjar-0.3.jar`
- `jrexx-1.1.1.jar`
- `managementapi-jrockit81.jar`
- `qdox-1.4.jar`
- `trove-1.0.2.jar`
- `tools.jar`

All the dependencies are inherited from AspectWerkz library. All of above listed files are delivered with distribution and are contained in its `lib` directory. The list of required libraries is also comprised in `build.xml` file as a `lemonade.aocl.run.classpath` classpath definition.

As long as AspectWerkz uses `java.lang.instrument` API in order to perform bytecode manipulation it provides its own instrumentation agent, that is introduced by `-javaagent:`*`path-to-lemonade-aocl.jar`* JVM option.

`Ant's build.xml` file contains dedicated targets for all the samples to run. In this sample such a target may look like:

```
    <target name="sample1" depends="jar">
           <java classname="pl.edu.agh.lemonade.aocl.sample.start.Sample1"
fork="true">
                  <classpath refid="lemonade.aocl.run.classpath" />
                  <jvmarg value="-javaagent:${lemonade.aocl.jar.name}" />
           </java>
    </target>
```

Therefore, in order to run above discussed sample simply type:

```
[lemonade-aocl-project-dir]$ ant sample1
```

ADS-aware aspect with assignable pointcut definition

The following example is similar to the previous one, although more robust since it take advantage of advanced AOCL features such as ADS-aware aspects and assignable pointcut definitions. The aspect code is slightly altered as it is depicted in Code Snippet 13.

First, aspect realizes `pl.edu.agh.lemonade.aocl.DeploymentScopeAwareAspect` interface which contains sole method – `setAspectDeploymentScope`. This method is used to provide aspect instance with additional context information related to encompassing ADS. Every aspect that realizes such the interface has this method called just after aspect instantiation. Second, the pointcut definition is no longer hard-coded in `Around` annotation, instead the symbolic name is used.

```
@Aspect
public class SampleAspect2 implements DeploymentScopeAwareAspect {

    // skipped

    @Around("pointcut")
    public Object monitor(StaticJoinPoint jp) throws Throwable {
            this.count++;
            System.out.println("before call #" + this.count + " by " + author);
            if (count % param != 0) {
                    return jp.proceed();
            } else {
                    System.out.println("call blocked");
                    return null;
            }
    }

    // skipped

    public void setAspectDeploymentScope(AspectDeploymentScope ads) {
            this.ads = ads;
            System.out.println("sample aspect is given with encompassing ADS: "
                            + this.ads.getName());
    }

}
```

Code Snippet 13. Static, ADS-aware aspect code with assignable pointcut definitions. The skipped parts are the same as in the previous example.

Such an aspect has to be appropriately deployed by explicitly providing pointcut definition that will bind pointcut symbolic name *pointcut* with actual pointcut expression. The rest of code remains unchanged as shown in Code Snippet 14.

```
public class Sample2 {

    public static void main(String[] args) throws Exception {

            // skipped

            aocl.activate();

            // pointcut expression may be defined at aspect deploy time
            Map<String, String> pointcutDefs = new HashMap<String, String>();
            pointcutDefs.put("pointcut",
                           "execution(void
pl.edu.agh.lemonade.aocl.sample.app.Clock.printDate())");

            // parameters may be passed to the aspect while deploying
            Map<String, String> params = new HashMap<String, String>();
            params.put("author", "me");

            // deploying aspect with previously defined pointcut definitions
            // and params

    ads.deployAspect("pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect2",
"myAspect",
                           pointcutDefs, params);

            // skipped
    }
}
```

Code Snippet 14. Sample usage of AOCL with simple desktop application involving static ADS and static ADS-aware aspect with assignable pointcut definition.

The way the glue code presented in Code Snippet 14 is launched remains unchanged. You might use again dedicated ant target:

```
[lemonade-aocl-project-dir]$ ant sample2
```

Parameterized Aspect Deployment Scope

Suppose we need a generic aspect which performs some well defined additional behavior with reference to abstract join-point, but we also want to avoid using explicitly pointcut expressions that may seem cumbersome and inconvenient to aspect developer or deployer. Instead, we want to provide high-level parameters values, which will imply actual pointcut expression resolved at aspect deployment time.

The practical example to consider may be tracing of method calls: we want to monitor each call from within some method (*caller method*) body to the other method (*callee method*) and let's call it *outer call*. Moreover, we want to monitor other calls the *callee method* performs within its body and let's call them *internal calls* if they call methods from the same *package scope*, or *inner calls* if they call method from outside of such *package scope*. What is more we want to trace not only *internal calls* and *inner calls* but as well accessing fields within *callee method* body. Such an abstract aspect may look like the one presented in Code Snippet 15.

```
@Aspect
public abstract class MethodTracerAspect implements MethodTracerAspectMBean,
            DeploymentScopeAwareAspect {

    private int depth = 0;

    private String id;

    protected final String packageScope;

    protected final String calleeMethodName;

    protected final String callerMethodName;

    protected AspectDeploymentScope scope;

    public MethodTracerAspect(AspectContext context) {
            this.packageScope = context.getParameter("packageScope");
            this.calleeMethodName = context.getParameter("calleeMethodName");
            this.callerMethodName = context.getParameter("callerMethodName");
            this.id = context.getParameter("_id");
    }

    @Around("outerCall")
    public final Object outerCall(StaticJoinPoint jp) throws Throwable {
            this.depth++;
            this.beforeOuterCall(jp);
            long time = System.currentTimeMillis();
            Object result = jp.proceed();
            time = System.currentTimeMillis() - time;
            this.afterOuterCall(jp, time);
            this.depth--;
            return result;
    }

    @Around("innerCall")
    public final Object innerCall(StaticJoinPoint jp) throws Throwable {
            this.depth++;
            this.beforeInnerCall(jp);
            long time = System.currentTimeMillis();
            Object result = jp.proceed();
            time = System.currentTimeMillis() - time;
            this.afterInnerCall(jp, time);
            this.depth--;
            return result;
    }

    @Around("internalCall")
    public final Object internalCall(StaticJoinPoint jp) throws Throwable {
            this.depth++;
            this.beforeInternalCall(jp);
            long time = System.currentTimeMillis();
            Object result = jp.proceed();
            time = System.currentTimeMillis() - time;
            this.afterInternalCall(jp, time);
            this.depth--;
            return result;
    }

    @Around("setting")
    public final Object fieldSet(StaticJoinPoint jp) throws Throwable {
            this.depth++;
            this.beforeFieldSet(jp);
            long time = System.currentTimeMillis();
            Object result = jp.proceed();
            time = System.currentTimeMillis() - time;
            this.afterFieldSet(jp, time);
            this.depth--;
            return result;
    }

    @Around("getting")
    public final Object fieldGet(StaticJoinPoint jp) throws Throwable {
            this.depth++;
            this.beforeFieldGet(jp);
            long time = System.currentTimeMillis();
            Object result = jp.proceed();
            time = System.currentTimeMillis() - time;
            this.afterFieldGet(jp, time);
            this.depth--;
            return result;
    }

    // skipped

}
```

Code Snippet 15. Abstract aspect performing method calls monitoring and accessing fields monitoring from within given method.

The resolution of pointcut expression, that has to be bound with pointcut names at aspect deployment time, is delegated to parameterized ADS. Such an ADS is both to resolve ADS pointcut expression basing on ADS pointcut parameters and to resolve aspect pointcuts expressions basing on aspect pointcut parameters. The method responsible for the former is `resolveADSPointcutExpression` while for the latter – `resolveAspectPointcutExpressions`. The implementation of parameterized ADS dedicated to our sample is presented in Code Snippet 16.

```
public class MethodTracerAspectDeploymentScope extends
            JMXParameterizedAspectDeploymentScope implements
            MethodTracerAspectDeploymentScopeMBean {

    /**
     * Here the package name we wanted to monitor is stored
     */
    private String packageScope;

    /**
     * @param name
     *            arbitrary ADS name
     * @param properties
     *            properties that parameterize this ADS instance
     * @throws RequiredParameterNotFoundException
     */
    public MethodTracerAspectDeploymentScope(String name, Properties properties)
                throws RequiredParameterNotFoundException {
            super(name, properties);

            // the only one parameter is the package name within which we want to
            // monitor method calls
            this.packageScope = properties.getProperty("packageScope");

            if (this.packageScope == null) {
                    // if some required parameter is not given the exception is
raised
                    throw new RequiredParameterNotFoundException("packageScope");
            }
    }

    public String getPackageScope() {
            return packageScope;
    }

    /*
     * implementation of
pl.edu.agh.lemonade.aocl.AspectDeploymentScope#resolveADSPointcutExpression()
     */
    public String resolveADSPointcutExpression() {
            // below it is resolved the pointcut expression of this ADS instance
            // basing on given ADS parameters (the only one is packageScope)
            return "(call(* " + this.packageScope + "..*(..)) && within("
                            + this.packageScope + "..)) || "
                            + "(call(* ..*(..)) && !call(* " + this.packageScope
                            + "..*(..)) && within(" + this.packageScope + "..)) || "
                            + "(set(* " + this.packageScope + "..*) && within("
                            + this.packageScope + "..)) || " + "(get(* "
                            + this.packageScope + "..*) && within(" +
this.packageScope
                            + "..))";
    }

    public String traceMethod(String aspectName, String callerMethodName,
                String calleeMethodName, String aspectClassName)
                throws ClassNotFoundException,
RequiredParameterNotFoundException {
```

```
            // below aspect pointcut parameter are collected in a map
            HashMap<String, String> pointcutParams = new HashMap<String,
String>();
            pointcutParams.put("calleeMethodName", calleeMethodName);
            pointcutParams.put("callerMethodName", callerMethodName);
            pointcutParams.put("packageScope", this.packageScope);

            // below the same parameters are passed as the ordinary string
            // key-values
            // pairs that are to be passed to the aspect instance
            HashMap<String, String> parameters = new HashMap<String, String>();
            parameters.put("calleeMethodName", calleeMethodName);
            parameters.put("callerMethodName", callerMethodName);

            // this inherited method is responsible for such configured aspect
            // deployment
            return this.deployParameterizedAspect(aspectClassName, aspectName,
                            pointcutParams, parameters);
     }

     @Override
     public void prepareAspectContext(AspectContext aspectContext) {

     }

     /*
      * implementation of
pl.edu.agh.lemonade.aocl.ParameterizedAspectDeploymentScope#resolveAspectPointcutE
xpressions(java.util.Map)
      */
     @Override
     protected Map<String, String> resolveAspectPointcutExpressions(
                   Map<String, String> pointcutParameters)
                   throws RequiredParameterNotFoundException {

            // below the pointcut parameters are read...
            String calleeMethodName = pointcutParameters.get("calleeMethodName");
            String callerMethodName = pointcutParameters.get("callerMethodName");

            if (callerMethodName == null) {
                   // if some required parameter is not given the exception is
raised
                   throw new
RequiredParameterNotFoundException("callerMethodName");
            }
            if (calleeMethodName == null) {
                   // if some required parameter is not given the exception is
raised
                   throw new
RequiredParameterNotFoundException("calleeMethodName");
            }

            // ... in order to resolve and return actual pointcut expressions
            HashMap<String, String> pointcutDefs = new HashMap<String, String>();
            pointcutDefs.put("outerCall", "call(" + calleeMethodName
                            + ") &amp;&amp; withincode(" + callerMethodName + ")");
            pointcutDefs.put("innerCall", "call(* ..*(..)) &amp;&amp; !call(* "
                            + packageScope + "..*(..)) &amp;&amp; withincode("
                            + calleeMethodName + ")");
            pointcutDefs.put("internalCall", "call(* " + this.packageScope
                           + "..*(..)) &amp;&amp; withincode(" + calleeMethodName +
")");
            pointcutDefs.put("setting", "set(* " + this.packageScope
                           + "..*) &amp;&amp; withincode(" + calleeMethodName +
")");
            pointcutDefs.put("getting", "get(* " + this.packageScope
                           + "..*) &amp;&amp; withincode(" + calleeMethodName +
")");

            return pointcutDefs;
    }

}
```

Code Snippet 16. Parameterized ADS dedicated for the abstract aspect depicted in Code Snippet 15.

What is needed now is a concrete aspect extending abstract aspect class that will actually perform monitoring logic. Suppose we want to count method call and overall time consumed by these calls. The code will look like in Code Snippet 17.

```
public class SampleAspect3 extends MethodTracerAspect {

    private int callCount = 0;

    private long overallTimeConsumed = 0;

    public SampleAspect3(AspectContext context) {
            super(context);
    }

    @Override
    protected void afterOuterCall(StaticJoinPoint jp, long time) {
            super.afterOuterCall(jp, time);

            // the counters has to be updated
            this.callCount++;
            this.overallTimeConsumed += time;

            // a proper message has to be displayed
            System.out.println("SampleAspect3: overall time consumed in "
                           + this.calleeMethodName + " in " + this.callCount
                           + " calls is " + this.overallTimeConsumed);
    }

}
```

Code Snippet 17. Concrete aspect based on MethodTracerAspect (see: Code Snippet 15) that counts given method calls and overall time consumed in such a method.

Note that above presented concrete aspect is not bound with any particular method until it is deployed. Deployment is performed in a glue code presented in Code Snippet 18.

```
public class Sample3 {

    public static void main(String[] args) throws Exception {

            // AOCL instantiation with name MyAOCL and domain name org.foo
            AspectOrientedClassLoader aocl = new AspectOrientedClassLoader(
                            ClassLoader.getSystemClassLoader(), "MyAOCL",
"org.foo");

            // appending jar containing application code to the AOCL classpath
            aocl.addJar("./jar/lemonade-sample-app.jar");

            // since our ADL is parameterized we need to prepare structure storing
            // parameters values
            Properties adsParameters = new Properties();

            // we want to pre-instrument all calls withing a given package
            adsParameters
                            .put("packageScope",
"pl.edu.agh.lemonade.aocl.sample.app");

            // instatinating parameterized ADS
            MethodTracerAspectDeploymentScope methodTracerScope = new
MethodTracerAspectDeploymentScope(
                            "methodTracerADS", adsParameters);

            // all ADSs has to be added to the AOCL before its activation
            aocl.addAspectDeploymentScope(methodTracerScope);

            // since AOCL is activated every loaded class is pre-intrumented if
            // needed
            aocl.activate();

            // deploying aspect tracing calls within:
            // public void pl.edu.agh.lemonade.aocl.sample.app.Clock.run()
            // targeted at:
            // public void pl.edu.agh.lemonade.aocl.sample.app.Clock.runInternal()
            aocl.addJar("./jar/lemonade-sample-aspect.jar");
            methodTracerScope
                            .traceMethod(
                                            "myAspect",
                                            "public void
pl.edu.agh.lemonade.aocl.sample.app.Clock.run()",
                                            "public void
pl.edu.agh.lemonade.aocl.sample.app.Clock.runInternal()",

    "pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect3");

            // loading application main class and launching
            Runnable clock = (Runnable) aocl.loadClass(

    "pl.edu.agh.lemonade.aocl.sample.app.Clock").newInstance();
            new Thread(clock).start();
    }

}
```

Code Snippet 18. Glue code instrumenting sample application with sample parameterized ADS.

Now the sample is ready to run and you can use dedicated ant target.

```
[lemonade-aocl-project-dir]$ ant sample3
```

Interactive configuration of Aspect-Oriented Class Loader

The following sample is equivalent to the first one, however the way how the AOCL is configured is significantly altered.

Suppose that we are dealing with two simultaneous threads – the first one which is actual application thread and the second one that manages application instrumentation in an interactive way.

In order to support interactive AOCL configuration it is provided dedicated AOCL's method `waitForActivation` which suspends application thread until instrumentation management thread allows it to resume. While application thread is suspended instrumentation management thread is enabled to perform necessary AOCL configuration that has to be done before loading application bytecode (mainly ADS configuration as long as aspects may be deployed dynamically at the application runtime). Having configuration completed, such a thread is expected to call `activate` AOCL's method.

Code Snippet 19 is equivalent to the first sample from this manual and shows how those thread interacts with each other.

```
public class Sample4 {

    public static void main(String[] args) throws Exception {

            // AOCL instantiation with name MyAOCL and domain name org.foo
            final AspectOrientedClassLoader aocl = new AspectOrientedClassLoader(
                           ClassLoader.getSystemClassLoader(), "MyAOCL",
"org.foo");

            // this thread will perform aocl configuration
            new Thread(new Runnable() {

                    public void run() {

try {

    // appending jar containing aspect code to the AOCL
    // classpath
    aocl.addJar("./jar/lemonade-sample-aspect.jar");
    // static ADS instantiation with a pre-instrumentation
    // pointcut
    // specified
    AspectDeploymentScope ads = new StaticAspectDeploymentScope(
                    "ds",
                    "execution(* pl.edu.agh.lemonade.aocl.sample.app.Clock.*(..))")
{

                    @Override
                    protected void prepareAspectDeploymentScope() {
                           // here the ADS preparation are to be performed
                           System.out
                                           .println("Preparing Aspect Deployment
Scope");
                    }

                    @Override
                    public void prepareAspectContext(
                           AspectContext aspectContext) {
                           // here the aspect context is to be prepared
                           // aspect context will be passed in aspect
                           // constructor
                           System.out.println("Preparing Aspect Context");
                           // aspect context metadata enable passing objects to
                           // the
                           // aspect
                           aspectContext.addMetaData("blockFrequency", 3);
                    }

                    @Override
                    protected void prepareAspect(Object aspectObj,
                                   AspectContext aspectContext) {
                           // here the aspect is to be prepared after its
                           // instatiation
                           System.out.println("Preparing Aspect");
                    }

            };
            // all ADSs has to be added to the AOCL before its
            // activation
            aocl.addAspectDeploymentScope(ads);

            // parameters may be passed to the aspect while deploying
            Map<String, String> params = new HashMap<String, String>();
            params.put("author", "me");

            // deploying aspect with no pointcut definitions
            // with above constructed params
            ads.deployAspect(
                   "pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect1",
                   "myAspect", null, params);

            // since AOCL is activated every loaded class is
            // pre-intrumented if needed
            // the main thread is resumed
            aocl.activate();
     } catch (InstantiationException e) {
                   e.printStackTrace();
     } catch (IllegalAccessException e) {
            e.printStackTrace();
     } catch (ClassNotFoundException e) {
            e.printStackTrace();
     } catch (ClassLoaderAlreadyActivatedException e) {
            e.printStackTrace();
     } catch (IOException e) {
            e.printStackTrace();
     }

}}).start();

            // this thread will wait after the aocl configuration is completed
            aocl.waitForActivation();

            // loading application main class and launching
            aocl.addJar("./jar/lemonade-sample-app.jar");
            Runnable clock = (Runnable) aocl.loadClass(

     "pl.edu.agh.lemonade.aocl.sample.app.Clock").newInstance();
            new Thread(clock).start();

     }

}
```

Code Snippet 19. Interactive configuration of AOCL involving two threads: actual application thread and application instrumentation management thread.

To run the sample you can use dedicated ant target.

```
[lemonade-aocl-project-dir]$ ant sample4
```

Using JMX-enabled Aspect-Oriented Class Loader

JMX AOCL is used as an AOCL overlay, thus, JMX-enabled AOCL API is composed of plain AOCL classes derivatives. JMX-enabled AOCL class inherits from AOCL class, therefore, it is painless to replace the latter with the former. However developer is expected to bear in mind that all ADSs have to be valid MBeans in order to use with JMX-enabled AOCL, so the proper JMX-enabled equivalent classes have to be used.

JMX-enabled AOCL constitutes exposition layer, enabling among others dynamic programming as well as remote monitoring and management of desktop applications.

JMX-enabled Static Aspect Deployment Scope

The next sample bases on the first one from section 0. However, since sample aspect is to be a standard MBean is has to realize dedicated MBean interface named similarly as an aspect class but with `MBean` suffix such as the one presented in Code Snippet 20.

```
public interface SampleAspect5MBean extends AbstractAspectMBean {

    // 'count' this will be accessible as a read-only attribute of this MBean
    public int getCount();

}
```

Code Snippet 20. JMX-enabled aspect as a valid MBean.

For the aspect developer convenience the `AbstractAspectMBean` interface along with its `AbstactAspect` realization are provided in distribution. They encapsulate and expose base attributes of aspects such as aspect name, aspect class name, ADS name, AOCL name etc.

As standard MBean convention states, MBean interface with corresponding name with 'MBean' suffix determines which attributes and methods of a given class that will be exposed. If we are dealing with a class hierarchy the corresponding MBean interfaces are expected to follow this hierarchy as well. That means that if an arbitrary MyAspect class inherits from AbstractAspect class, corresponding MyAspectMBean has to inherits form

AbstractAspectMBean interface as well just as it is depicted in Figure 19. Therefore, the aspect code will look like in Code Snippet 21.

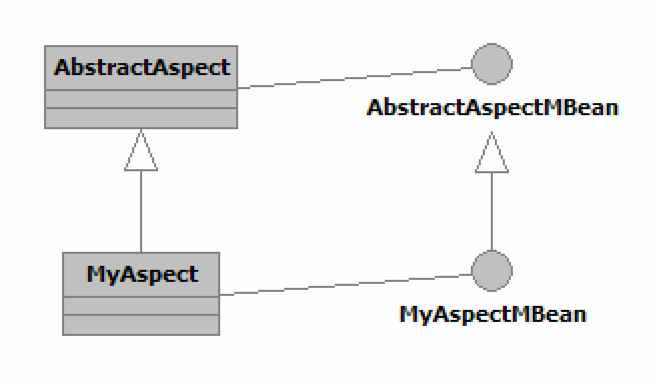


Figure 19. The standard MBeans class hierarchy as implied by a convention.

```
@Aspect
public class SampleAspect5 extends AbstractAspect implements SampleAspect5MBean {

    private final int param;

    private final String author;

    private int count = 0;

    public SampleAspect5(AspectContext ctx) {
        this.param = 3;
        this.author = ctx.getParameter("author");
    }

    @Around("execution(void
pl.edu.agh.lemonade.aocl.sample.app.Clock.printDate())")
    public Object monitor(StaticJoinPoint jp) throws Throwable {
        this.count++;
        System.out.println("before call #" + this.count + " by " + author);
        if (count % param != 0) {
            return jp.proceed();
        } else {
            System.out.println("call blocked");
            return null;
        }
    }

    // implements SampleAspect5MBean
    public int getCount() {
        return this.count;
    }
}
```

Code Snippet 21. Sample JMX-enabled aspect that inherits from `AbstractAspect` helper class.

Moreover, the glue code (as shown in Code Snippet 22) has to be slightly modified simply by replacing `AspectOrientedClassLoader` instance with `JMXAspectOrientedClassLoader` instance.

```
public class Sample5 {

    public static void main(String[] args) throws Exception {

            // JMXAOCL instantiation with name MyAOCL and domain name org.foo
            JMXAspectOrientedClassLoader aocl = new JMXAspectOrientedClassLoader(
                           ClassLoader.getSystemClassLoader(), "MyAOCL",
"org.foo");
            // from now on JMX AOCL is exported and registered in MBeanServer
            // thus, it is configurable via JMX API

            // appending jar containing application code to the AOCL classpath
            aocl.addJar("./jar/lemonade-sample-app.jar");

            // appending jar containing aspect code to the AOCL classpath
            aocl.addJar("./jar/lemonade-sample-aspect.jar");

            // static ADS instantiation with a pre-instrumentation pointcut
            // specified
            JMXStaticAspectDeploymentScope jmxStaticAds = new
JMXStaticAspectDeploymentScope(
                           "ds",
                           "execution(*
pl.edu.agh.lemonade.aocl.sample.app.Clock.*(..))") {

                   @Override
                   public void prepareAspectContext(AspectContext aspectContext) {
                           // here the aspect context is to be prepared
                           // aspect context will be passed in aspect constructor
                           System.out.println("Preparing Aspect Context");

                           // aspect context metadata enable passing objects to the
aspect
                           aspectContext.addMetaData("blockFrequency", 3);
                   }

            };

            // all ADSs has to be added to the AOCL before its activation
            aocl.addAspectDeploymentScope(jmxStaticAds);

            // since AOCL is activated every loaded class is pre-intrumented if
            // needed
            aocl.activate();

            // parameters may be passed to the aspect while deploying
            Map<String, String> params = new HashMap<String, String>();
            params.put("author", "me");

            // deploying aspect with no pointcut definitions
            // with above constructed params
            jmxStaticAds.deployAspect(
                           "pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect5",
                           "myAspect", null, params);

            // loading application main class and launching
            Runnable clock = (Runnable) aocl.loadClass(
      "pl.edu.agh.lemonade.aocl.sample.app.Clock").newInstance();
              new Thread(clock).start();
      }

}
```

Code Snippet 22. A glue code that involves JMX-enabled AOCL and runs sample application.

Nevertheless, the way how the glue code is executed has to be altered since some JMX related JVM arguments are required. As long as security wasn't addressed concern in this sample we use neither password authentication nor SSL. Therefore, additional JXM related JVM arguments used are:

- `com.sun.management.jmxremote.port`
  – sets the port number through which you want to enable JMX/RMI connections
- `com.sun.management.jmxremote.authenticate`
  – enables/disables authentication, it has to be set to true or false
- `com.sun.management.jmxremote.ssl`
  – enables/disables SSL, it has to be set to true or false

Hereby, appropriate ant target is presented in Code Snippet 23.

```
      <target name="sample5" depends="jar">
              <java classname="pl.edu.agh.lemonade.aocl.sample.start.Sample5"
fork="true">
                      <classpath refid="lemonade.aocl.run.classpath" />
                      <jvmarg value="-javaagent:${lemonade.aocl.jar.name}" />
                      <jvmarg value="-Dcom.sun.management.jmxremote.port=1234" />
                      <jvmarg value="-
Dcom.sun.management.jmxremote.authenticate=false" />
                      <jvmarg value="-Dcom.sun.management.jmxremote.ssl=false" />
              </java>
      </target>
```

Code Snippet 23. Ant target (a part of `build.xml` file) that runs provided application main class and takes care about a proper configuration of classpath and JVM arguments required by JMX-enabled AOCL.

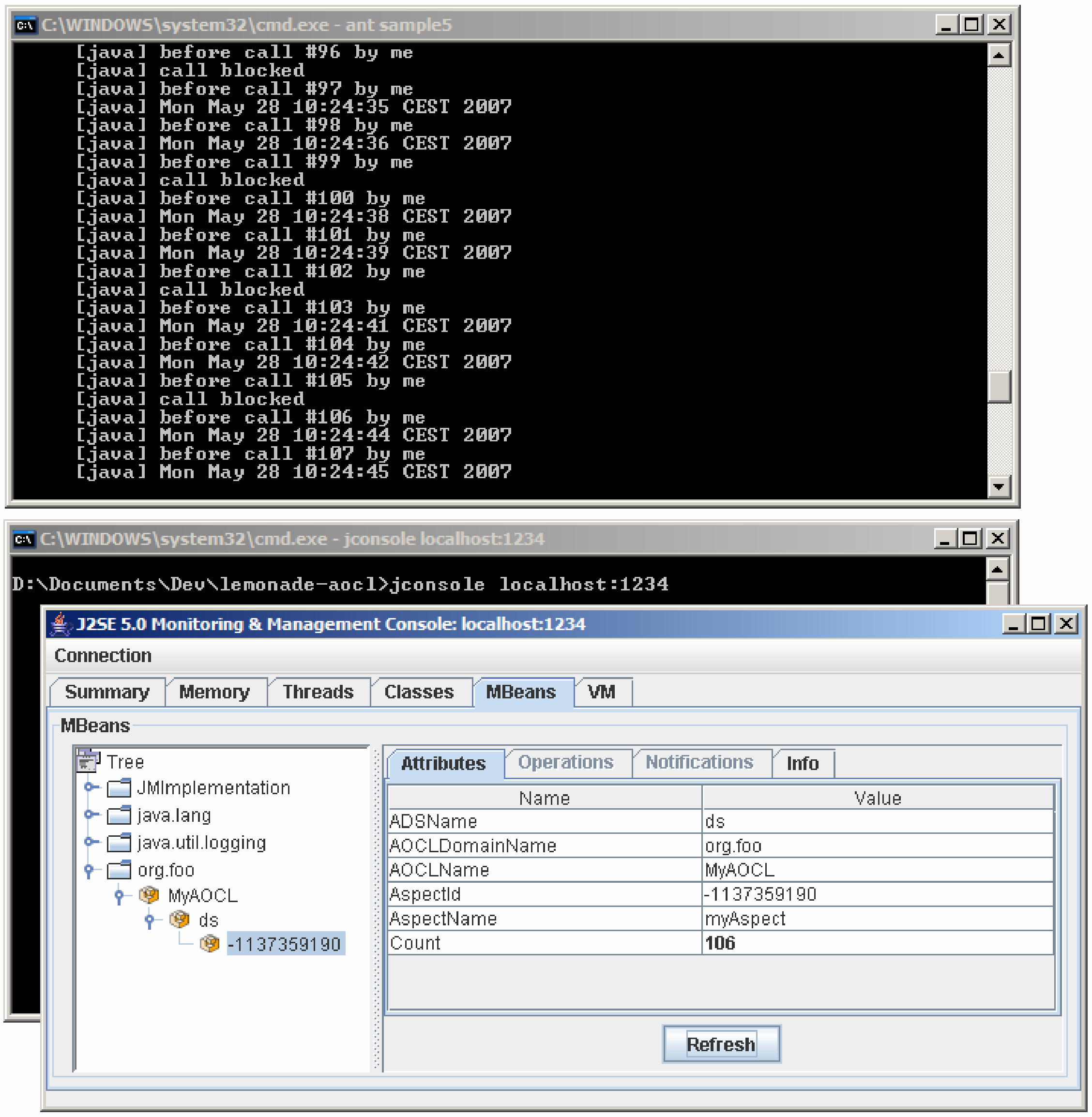
Such a target allows us running the sample by typing:

```
[lemonade-aocl-project-dir]$ ant sample5
```

The output console is verbose and prints out monitoring events, however we are not compelled to track console any more. Since AOCL, ADSs and aspects are registered MBeans we may use any of variety of JMX-compliant consoles and tools to monitor them. So, we may use e.g. the standard tool that comes with JDK - `jconsole`, and provide host name and port number for RMI connector. In our sample we run it as follows:

```
[lemonade-aocl-project-dir]$ jconsole localhost:1234
```

In a JMX console we find domain named after AOCL's domain name (in this case: `org.foo`) where AOCL is registered with its name. Then we should see a view that is presented in Figure 20. AOCL MBean constitutes a root for tree-like structure of nested ADSs' MBeans and aspects' MBeans. The ADS is registered with its name as it was provided in a glue code while aspect unique names are aids (aspect ids) that are generated just before aspect deployment. The MBeans structure is depicted in left-hand tree panel of JMX console depicted in Figure 20. By choosing the aspect MBean we are given with its operations, attributes and notifications available. In this sample the `Count` attribute may be monitored among others.



```
C:\WINDOWS\system32\cmd.exe - ant sample5
     [java] before call #96 by me
     [java] call blocked
     [java] before call #97 by me
     [java] Mon May 28 10:24:35 CEST 2007
     [java] before call #98 by me
     [java] Mon May 28 10:24:36 CEST 2007
     [java] before call #99 by me
     [java] call blocked
     [java] before call #100 by me
     [java] Mon May 28 10:24:38 CEST 2007
     [java] before call #101 by me
     [java] Mon May 28 10:24:39 CEST 2007
     [java] before call #102 by me
     [java] call blocked
     [java] before call #103 by me
     [java] Mon May 28 10:24:41 CEST 2007
     [java] before call #104 by me
     [java] Mon May 28 10:24:42 CEST 2007
     [java] before call #105 by me
     [java] call blocked
     [java] before call #106 by me
     [java] Mon May 28 10:24:44 CEST 2007
     [java] before call #107 by me
     [java] Mon May 28 10:24:45 CEST 2007
```

```
C:\WINDOWS\system32\cmd.exe - jconsole localhost:1234

D:\Documents\Dev\lemonade-aocl>jconsole localhost:1234
```

Name	Value
ADSName	ds
AOCLDomainName	org.foo
AOCLName	MyAOCL
AspectId	-1137359190
AspectName	myAspect
Count	106

Figure 20. JMX-enabled AOCL sample. The application console (upper), JMX console (lower) with AOCL, ADSs and aspects arranged in a tree-like structure (left-hand panel) and attributes panel of sample aspect MBean (right-hand panel).

Configuration of JMX-enabled Aspect-Oriented Class Loader

In order to facilitate configuration management of JMX-enabled AOCLs dedicated `JMXAspectOrientedClassLoaderConfiguration` class is introduced. It is used to store, serialize and deserialize JMX-enabled AOCLs configuration and to enable configuration management without explicit manipulation on `JMXAspectOrientedClassLoader` instance.

This is alternative way to configure JMX-enabled AOCLs that is especially suitable for remote monitoring as long as configuration may be stored in `JMXAspectOrientedClassLoaderConfiguration` instance, serialized, passed though network, deserialized and finally applied to JMX-enabled AOCL instance.

Code Snippet 24 shows a glue code equivalent to `Sample1` class that takes advantage of `JMXAspectOrientedClassLoaderConfiguration`.

```
public class Sample6 {

    public static void main(String[] args) throws Exception {

            // instantiation of empty AOCL configuration
            JMXAspectOrientedClassLoaderConfiguration config =
JMXAspectOrientedClassLoaderConfiguration
                            .newEmpty("MyAOCL", "org.foo");

            // appending jar containing application code to the AOCL classpath
            config.addJar("./jar/lemonade-sample-app.jar");

            // static ADS configuration with a pre-instrumentation pointcut
            // specified
            JMXAspectOrientedClassLoaderConfiguration.StaticADSConfiguration
staticADSConfig = new
JMXAspectOrientedClassLoaderConfiguration.StaticADSConfiguration(
                            "staticADS",
                            "execution(*
pl.edu.agh.lemonade.aocl.sample.app.Clock.*(..))");

            // static aspect configuration
            JMXAspectOrientedClassLoaderConfiguration.StaticAspectConfiguration
staticAspectConfig = new
JMXAspectOrientedClassLoaderConfiguration.StaticAspectConfiguration(
                            "staticAspect",
                            "pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect8",
                            "file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-
sample-aspect.jar");
            // pointcut definition has to be provided...
            staticAspectConfig.addPointuctDef("myPointcut",
                            "execution(* java.lang.Runnable+.*(..))");
            // ... as well as parameters
            staticAspectConfig.addParameter("author", "me");
            staticAspectConfig.addParameter("date", "today");
            staticADSConfig.addStaticAspectConfiguration(staticAspectConfig);
            config.addAspectDeploymentScopeConfiguration(staticADSConfig);

            // configuration of parameterized ADS which is provided as a jar file

    JMXAspectOrientedClassLoaderConfiguration.ParameterizedADSConfiguration
parameterizedADSConfig = new
JMXAspectOrientedClassLoaderConfiguration.ParameterizedADSConfiguration(
                            "parameterizedADS",
                            "file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-
sample-scope.jar");
            parameterizedADSConfig.addPointcutParameter("packageScope",
                            "pl.edu.agh.lemonade.aocl.sample.app");
            config.addAspectDeploymentScopeConfiguration(parameterizedADSConfig);

            // parameterized aspect configuration

    JMXAspectOrientedClassLoaderConfiguration.ParameterizedAspectConfiguration
parameterizedAspectConfig = new
JMXAspectOrientedClassLoaderConfiguration.ParameterizedAspectConfiguration(
                            "parameterizedAspect",
                            "pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect7",
                            "file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-
sample-aspect.jar");
            // pointcut params has to be provided
            parameterizedAspectConfig.addPointcutParam("callerMethodName",
                              "public void
pl.edu.agh.lemonade.aocl.sample.app.Clock.run()");
            parameterizedAspectConfig
                              .addPointcutParam("calleeMethodName",
                                             "public void
pl.edu.agh.lemonade.aocl.sample.app.Clock.runInternal()");
            parameterizedADSConfig

    .addParameterizedAspectConfiguration(parameterizedAspectConfig);

            // JMXAOCL instantiation with above-specified confguration
            JMXAspectOrientedClassLoader aocl = new JMXAspectOrientedClassLoader(
                           ClassLoader.getSystemClassLoader(), config);

            // from now on JMX AOCL is exported and registered in MBeanServer
            // thus, it is configurable via JMX API
            // since AOCL is activated every loaded class is pre-intrumented if
            // needed
            aocl.activate();

            // loading application main class and launching
            Runnable clock = (Runnable) aocl.loadClass(

    "pl.edu.agh.lemonade.aocl.sample.app.Clock").newInstance();
            new Thread(clock).start();
    }
}
```

Code Snippet 24. A glue code that configures JMX-enabled AOCL by taking advantage of `JMXAspectOrientedClassLoaderConfiguration` class, and launches the sample application.

This sample is launched by dedicated ant target:

```
[lemonade-aocl-project-dir]$ ant sample6
```

The launched sample may be monitored by any JMX tool.

XML-based configuration of JMX-enabled Aspect-Oriented Class Loader

The following section is devoted to a sample that is equivalent to the previous one (in section 0), however it is characterized by alternative way for configuring JMX-enabled AOCL. As mentioned recently `JMXAspectOrientedClassLoaderConfiguration` is serializable, although not in terms of `java.io.Serializable` but it may be represented as XML document.

Verbose glue code from previous section shown in Code Snippet 24 may be replaced by more concise code presented in Code Snippet 25.

```
public class Sample7 {

    public static void main(String[] args) throws Exception {

            JMXAspectOrientedClassLoaderConfiguration config =
JMXAspectOrientedClassLoaderConfiguration
                            .parse(new FileInputStream(new File("./sample-aocl-
config.xml")));

            JMXAspectOrientedClassLoader aocl = new JMXAspectOrientedClassLoader(
                            ClassLoader.getSystemClassLoader(), config);
            aocl.activate();

            Runnable clock = (Runnable)
aocl.loadClass("pl.edu.agh.lemonade.aocl.sample.app.Clock")
                            .newInstance();
            new Thread(clock).start();
    }
}
```

Code Snippet 25. Concise glue code that reads JMX-enabled AOCL configuration from XML-based document and launches the sample application.

In such a code whole configuration is read from external XML file. In our example `sample-aocl-config.xml` file is used which is included in the distribution and presented in Code Snippet 26. Such a file exactly corresponds to the configuration made by code shown in Code Snippet 24.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<jmxaocl name="myapp" domainName="org.foo.myapp">

    <jarUrl>
            file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-sample-app.jar
    </jarUrl>

    <staticADS name="staticADS"
            pointcutExpression="execution(* java.lang.Runnable+.*(..))">

            <staticAspect name="staticAspect"
                    class="pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect8"
                    jar="file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-
sample-aspect.jar">
                    <pointcut name="myPointcut"
                            expression="execution(* java.lang.Runnable+.*(..))" />
                    <parameter name="author" value="me" />
                    <parameter name="date" value="today" />
            </staticAspect>

    </staticADS>

    <parameterizedADS name="parameterizedADS"
            class="ec.aocl.sample.scope.MethodTracerAspectDeploymentScope"
            jar="file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-sample-
scope.jar">
            <pointcutParameter name="packageScope"
                    value="pl.edu.agh.lemonade.aocl.sample.app" />

            <parameterizedAspect name="parameterizedAspect"
                    class="pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect7"
                    jar="file:///D:/Documents/Dev/lemonade-aocl/jar/lemonade-
sample-aspect.jar">
                    <pointcutParameter name="callerMethodName"
                            value="void
pl.edu.agh.lemonade.aocl.sample.app.Clock.run()" />
                    <pointcutParameter name="calleeMethodName"
                            value="void
pl.edu.agh.lemonade.aocl.sample.app.Clock.runInternal()" />
                    <parameter name="author" value="me" />
                    <parameter name="date" value="today" />
            </parameterizedAspect>

    </parameterizedADS>

</jmxaocl>
```

Code Snippet 26. Configuration of JMX-enabled AOCL serialized as an XML-based form that is bidirectionally transformable to/from `JMXAspectOrientedClassLoaderConfiguration` instance.

In order to start sample we can use as usual ant target:

```
[lemonade-aocl-project-dir]$ ant sample7
```

Using Mocca/Moccaccino leMonAdE Edition

The following instructions has to be followed in order to setup a local host-scoped testbed as well as to setup fully valid distributed execution environment.

- Get the distribution of H2O. Version 2.1 that was used in tests is recommended.
- In **lemonade-aocl** project create appropriately `h2o.properties` file basing on `h2o.properties.example` file.
- Build **lemonade-aocl** and deploy it to the h2o library directory by executing:

```
[lemonade-aocl-dir]$ ant deploy
```

- In **mocca-lemonade** project create appropriately `h2o.properties` file basing on `h2o.properties.example` file and `jmx.properties` file basing on `jmx.properties.example` file.
- Build **mocca-lemonade** and deploy it file to the h2o services directory by executing:

```
[mocca-lemonade-project-dir]$ ant deploy
```

- In **moccaccino-lemonade** project create appropriately `h2o.properties` file basing on `h2o.properties.example`, `aocl.properties` file basing on `aocl.properties.example` file and `mocca-lemonade.properties` file basing on `mocca-lemonade.properties.example` file.
- Build **moccaccino-lemonade** and deploy it file to the h2o services directory by executing:

```
[moccaccino-lemonade-project-dir]$ ant deploy
```

- Now you can easily start h2o testbed consisting of one locally launched kernel according to its configuration written in `\etc\KernelConfig-0.xml` just by executing:

```
[mocca-lemonade-project-dir]$ ant h2o
```

- Sample application is run by executing (n is a sample number – 1 or 2):

```
[moccaccino-lemonade-project-dir]$ ant sample-n
```

- Custom application may be submitted with assistance of lemonade-enabled Moccaccino Manager e.g. by running the following (Code Snippet 27) ant target:

```
<target name="runMyApplication">
            <java classname="edu.agh.moccaccino.manager.impl.StartManager"
fork="true">
                    <classpath refid="moccaccino.run.classpath" />
                    <arg value="[URL to ADLM file]" />
                    <arg value="[URL to AISM file]" />
                    <arg value="[First kernel endpoint address]" />
                    <arg value="[Second kernel endpoint adress]" />
                    [another kernels endpoints]
            </java>
      </target>
```

Code Snippet 27. Generic ant target for submission application by Moccaccino Manager.

Sample Application

Sample application that may be found in the Moccaccino leMonAdeE Edition distribution is assembled of component instances arranged in a tree-like structure that delegates calls top-down. Application architecture that corresponds to ADLM description presented in Code Snippet 28. is depicted in Figure 21. For further details on ADLM refer to the Moccaccino documentation [59].

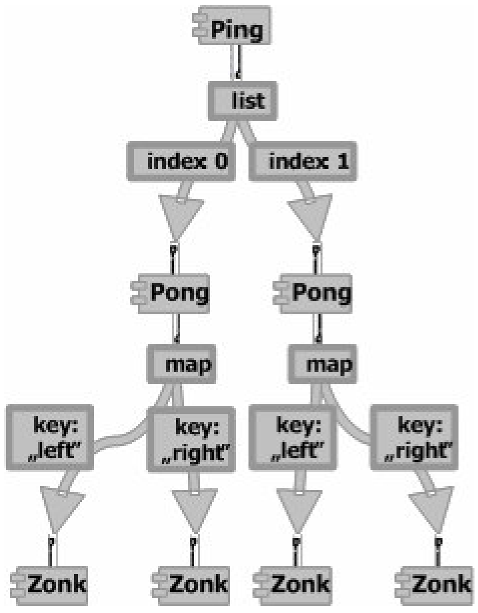


Figure 21. Architecture of sample application contained in Moccaccino Manager distribution. Component instances are arranged in a tree-like structure; components refer to themselves via qualifiers (lists, maps).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "moccaccino.adl" "../../adl-schemas/adl-schema.dtd" >
<application name="PingPongZonkApplication">
    <component-classes
            codebase="file:///D:/Documents/Dev/moccaccino-lemonade/jar/moccaccino-
lemonade-samples.jar">
            <component-class name="Ping"
                    classname="edu.agh.moccaccino.sampleappa.Ping">
                    <provides>
                            <port name="MyGoPort"
    classname="edu.agh.moccaccino.sampleappa.PingPort"/>
                    </provides>
                    <uses qualifier="list">
                            <port name="pongs"
    classname="edu.agh.moccaccino.sampleappa.PongPort"/>
                    </uses>
            </component-class>

            <component-class name="Pong"
                    classname="edu.agh.moccaccino.sampleappa.Pong">
                    <provides>
                            <port name="PongPort"
    classname="edu.agh.moccaccino.sampleappa.PongPort"/>
                    </provides>
                    <uses qualifier="map">
                            <port name="zonks"
    classname="edu.agh.moccaccino.sampleappa.ZonkPort"/>
                    </uses>
            </component-class>

            <component-class name="Zonk"
                    classname="edu.agh.moccaccino.sampleappa.Zonk">
                    <provides>
                            <port name="ZonkPort"
    classname="edu.agh.moccaccino.sampleappa.ZonkPort"/>
                    </provides>
            </component-class>
    </component-classes>

    <assembly>
            <component-group name="ping" component-class="Ping" weight="5">
                    <connection usesPort="pongs" qualifier-attribs="length=2"
                            providesPort="PongPort" weight="5" shared="false">
                            <component-group name="pongs" component-class="Pong"
weight="5">
                                    <connection usesPort="zonks"
                                            qualifier-attribs="keys=one;two"
providesPort="ZonkPort" weight="5" shared="false">
                                            <component-group name="zonks" component-
class="Zonk" weight="5"/>
                                    </connection>
                            </component-group>
                    </connection>
            </component-group>
    </assembly>

    <deployment/>
    <configuration-port-type-maps>
           <!—skipped-->
    </configuration-port-type-maps>

    <execution>
           <component path="."/>
    </execution>
</application>
```

Code Snippet 28. Architecture description of sample application contained in Moccaccino Manager distribution. Expressed in XML-based ADLM language. This document describes the architecture depicted in Figure 21.

In Moccaccino leMonAdE Edition the new concept of Application Instrumentation Specification for Moccaccino (AISM) is introduced. This is a XML-based language that contain configuration of all JMX-enabled AOCLs that are involved in application. AISM refers to component groups defined in ADLM. The possible instrumentation of sample application is presented in Code Snippet 29.

The main element of AISM document is `instrumentation` with attributes `userName` and `password` used as authentication credentials while connecting to the MBeanServers. Instrumentation element consists of zero or more `componentGroupInstrumentation` elements that bind component groups with a configuration of AOCLs used by components instances of these groups. In fact, `componentGroupInstrumentation` has the same syntax as `jmxaocl` element (see: Code Snippet 26), although accepts additional nested `componentGroup` elements that specify component groups the `componentGroupInstrumentation` configuration is to be applied to.

```
<?xml version="1.0" encoding="UTF-8"?>

<instrumentation userName="" password="">
    <componentGroupInstrumentation>
           <componentGroup name="ping" />
           <staticADS name="staticADS"
                  pointcutExpression="execution(private void
edu.agh.moccaccino.sampleappa.Ping.printDate())">
                  <staticAspect name="staticAspect"

    class="pl.edu.agh.lemonade.aocl.sample.aspect.SampleAspect8"
                         jar="file:///D:/Documents/Dev/lemonade-
aocl/jar/lemonade-sample-aspect.jar">
                         <pointcut name="myPointcut"
                                expression="execution(private void
edu.agh.moccaccino.sampleappa.Ping.printDate())" />
                         <parameter name="author" value="me" />
                         <parameter name="date" value="today" />
                  </staticAspect>
           </staticADS>
    </componentGroupInstrumentation>
</instrumentation>
```

Code Snippet 29. AISM document for sample application. It refers to the application ADLM description from Code Snippet 28.

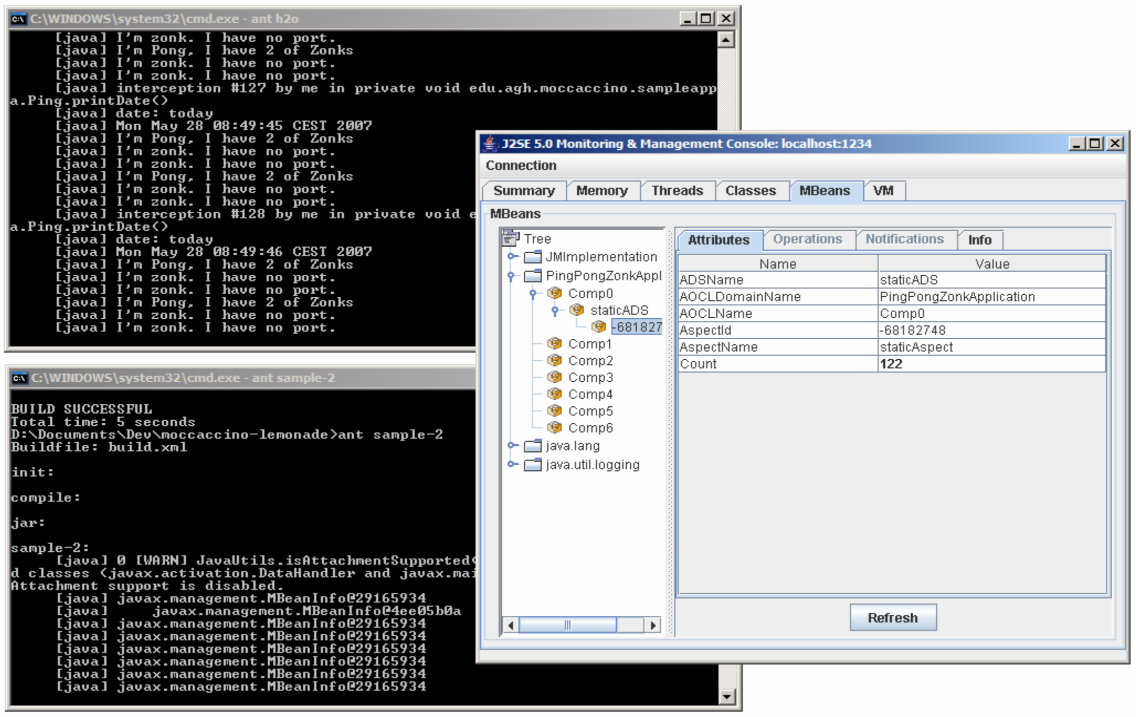
In order to start testbed the following ant target may be applied:

```
[mocca-lemonade-project-dir]$ ant h2o
```

Having local H2O kernel up the sample application may be run by dedicated ant target:

```
[moccaccino-lemonade-project-dir]$ ant sample-2
```

Since this example involves single one H2O kernel, all involved AOCLs are registered in the sample MBeanServer. Therefore, whole application may be monitored by single standard JMX console. The screenshot presented in Figure 22 shows the output consoles of local H2O kernel and Moccaccino Manager along with JConsole connected to the kernel's JVM. In JConsole, note that one AOCL is per one component instance and all AOCLs are contained in the application name domain.



```
     [java] I'm zonk. I have no port.
     [java] I'm Pong, I have 2 of Zonks
     [java] I'm zonk. I have no port.
     [java] I'm zonk. I have no port.
     [java] interception #127 by me in private void edu.agh.moccaccino.sampleapp
a.Ping.printDate()
     [java] date: today
     [java] Mon May 28 08:49:45 CEST 2007
     [java] I'm Pong, I have 2 of Zonks
     [java] I'm zonk. I have no port.
     [java] I'm zonk. I have no port.
     [java] I'm Pong, I have 2 of Zonks
     [java] I'm zonk. I have no port.
     [java] I'm zonk. I have no port.
     [java] interception #128 by me in private void e
a.Ping.printDate()
     [java] date: today
     [java] Mon May 28 08:49:46 CEST 2007
     [java] I'm Pong, I have 2 of Zonks
     [java] I'm zonk. I have no port.
     [java] I'm zonk. I have no port.
     [java] I'm Pong, I have 2 of Zonks
     [java] I'm zonk. I have no port.
     [java] I'm zonk. I have no port.
```

```
BUILD SUCCESSFUL
Total time: 5 seconds
D:\Documents\Dev\moccaccino-lemonade>ant sample-2
Buildfile: build.xml

init:

compile:

jar:

sample-2:
     [java] 0 [WARN] JavaUtils.isAttachmentSupported(
d classes (javax.activation.DataHandler and javax.mai
Attachment support is disabled.
     [java] javax.management.MBeanInfo@29165934
     [java]     javax.management.MBeanInfo@4ee05b0a
     [java] javax.management.MBeanInfo@29165934
     [java] javax.management.MBeanInfo@29165934
     [java] javax.management.MBeanInfo@29165934
     [java] javax.management.MBeanInfo@29165934
     [java] javax.management.MBeanInfo@29165934
     [java] javax.management.MBeanInfo@29165934
```

Name	Value
ADSName	staticADS
AOCLDomainName	PingPongZonkApplication
AOCLName	Comp0
AspectId	-68182748
AspectName	staticAspect
Count	122

Figure 22. Sample Moccaccino application that runs within local testbed: local H2O kernel console (upper), Moccaccino Manager console (lower), JMX JConsole (right) with tree-like structure of MBeans involved in application run (left panel).

Additional information

For the detailed information and documentation user is expected to view the source code and the source code documentation. For deepened view of samples code the user should refer to the distribution. Additional information and support may be received by contacting the author.

Appendix B. Papers Relevant to the Thesis

The following appendix provides the papers co-authored by the author of this thesis that are relevant to this work, respectively:

1. Eryk Ciepiela, Maciej Malawski, Bartosz Baliś, Marian Bubak: **System for Monitoring of Component-based Applications**, submitted to Seventh International Conference on Parallel Processing and Applied Mathematics PPAM, Gdańsk, Poland 2007

2. Maciej Malawski, Tomasz Bartyński, Eryk Ciepiela, Joanna Kocot, Przemysław Pelczar and Marian Bubak: **An ADL-based Support for CCA Components**, presented in CoreGRIDWorkshop on Grid Systems, Tools and Environments in Conjunction with GRIDS@work: CoreGRID Conference, Grid Plugtests and Contest, Sophia-Antipolis, France, October 2006

3. Maciej Malawski, Tomasz Bartyński, Eryk Ciepiela, Joanna Kocot, Przemysław Pelczar, Marian Bubak: **A new Approach to Supporting Component Applications on Grid**, Cracow Grid Workshop Proceedings, Kraków, Poland, October 2006 (in press)