Hochschule RheinMain
Fachbereich Design Informatik Medien
Studiengang Angewandte Informatik

Bachelor-Arbeit
zur Erlangung des akademischen Grades
Bachelor of Science - B.Sc.

Methodische Vorgaben für Performance-Monitoring im Kontext betrieblicher Java-Anwendungen

vorgelegt von
Johannes Mensing
Matrikel-Nummer 763013
am 15.09.2014

Betreuer: Oliver Hecker (Capgemini)
Referent: Prof. Dr. Sven Eric Panitz
Korreferent: Prof. Dr. Bodo. A. Igler

Ich versichere, dass ich die Bachelor-Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Ort, Datum Unterschrift Studierender

Hiermit erkläre ich mein Einverständnis mit den Folgenden aufgeführten Verbreitungsformen dieser Bachelor-Arbeit:

Verbreitungsformen	**ja**	**nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger		×
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	×	
Veröffentlichung des Titels der Arbeit im Internet	×	
Veröffentlichung der Arbeit im Internet		×

Ort, Datum Unterschrift Studierender

Thema der Bachelorarbeit

Methodische Vorgaben für Performance-Monitoring im Kontext betrieblicher Java-Anwendungen

Stichworte

Fachliches Performance-Monitoring, Performance, Monitoring, Java, Kieker, Leitfaden

Kurzzusammenfassung

Performance-Monitoring ist ein wichtiger Bestandteil der Qualitätssicherung in IT-Systemen. Dennoch wird es in der Praxis gerade bei kleineren Projekten häufig aus Kosten- und Zeitgründen vernachlässigt. Diese Arbeit beschäftigt sich mit dem Performance-Monitoring in betrieblichen Java-Anwendungen und beschreibt, wie ein Verfahren entwickelt wurde, um fachliches Performance-Monitoring unter der Verwendung des Kieker-Frameworks schnell und effizient in Java-Projekte zu integrieren. Der Nutzen des Monitorings soll dabei nicht erst zur Inbetriebnahme der Anwendung sichtbar werden, sondern vielmehr das Entwicklungsteam schon während den verschiedenen Entstehungsphasen der Software unterstützen. Das Verfahren zeigt eine Reihe von Best-Practices zur Code-Instrumentierung, Konfiguration und Erweiterung des Monitorings sowie zur Analyse der gesammelten Daten. Auch Schwierigkeiten und Fallstricke bei dieser Art des Performance-Monitorings werden betrachtet und Möglichkeiten zur individuellen Erweiterung aufgezeigt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Diese Arbeit beschreibt, wie in Zusammenarbeit mit Capgemini (Offenbach) ein Leitfaden für fachliches Performance-Monitoring in betrieblichen Java-Anwendungen entwickelt wurde. Da der Leitfaden insbesondere für kleine und mittelgroße Projekte gedacht ist, liegt sein Fokus vor allem auf der Simplizität seiner Anwendung. So soll er Entwickler befähigen ein Performance-Monitoring-System in eine Java-Anwendung zu integrieren, ohne zuvor lange Recherchen oder Einarbeitungszeiten in Kauf nehmen zu müssen. Die Integration kann schon während des Entstehungsprozesses der Software erfolgen und so schon früh zu einem qualitativ hochwertigen Ergebnis beitragen.

1.1 Gliederung der Arbeit

Die Arbeit gliedert sich in vier Kapitel, wobei das erste Kapitel die Einleitung darstellt. Hier wird ein kurzer Überblick über das Themengebiet gegeben und die Motivation sowie die Zielsetzung für diese Arbeit beschrieben.

Das zweite Kapitel beschäftigt sich mit den Grundlagen, die zum Verständnis dieser Arbeit notwendig sind. So werden hier einige themenspezifische Begriffe erklärt und das Programmierparadigma der *aspektorientierten Programmierung* sowie die grundlegende Funktionsweise von Monitoring-Tools erläutert.

In Kapitel drei wird ausführlich auf den Leitfaden für das Performance-Monitoring in Java-Anwendungen eingegangen, dessen Entstehung Kern dieser Arbeit ist. Hierbei wird das verwendete Framework *Kieker* genau betrachtet und seine Funktionsweise erklärt. Ferner wird erläutert, wie es für den Leitfaden verwendet und

erweitert wurde. Auch auf mögliche Stolpersteine bei der Nutzung des Leitfadens sowie seine Erweiterungsmöglichkeiten wird in diesem Kapitel eingegangen.

Das vierte und letzte Kapitel der Arbeit beinhalten die Reflexion, im Hinblick auf die zum Anfang der Arbeit definierte Zielsetzung. Des Weiteren wird ein Ausblick gegeben, bezüglich der in der Zukunft möglichen Erweiterungen.

Der Anhang dieser Arbeit beinhaltet den vollständigen Leitfaden für das Performance-Monitoring in betrieblichen Java-Anwendungen, der im Rahmen dieser Arbeit entstanden ist.

1.2 Motivation

Der Architekturleitfaden von Capgemini[1] besagt in Regel 136:

> *„Die Analyse der nichtfunktionalen Eigenschaften eines Softwaresystems mit technischem Monitoring ist sinnvoll und sollte in jedem Projekt zum Einsatz kommen."*

und weiterhin in Regel 139:

> *„Das fachliche Monitoring sollte in jedes Softwaresystem von Anfang an eingebaut werden."*

In der Praxis sieht es allerdings anders aus. In den meisten Fällen wird der zusätzliche Aufwand für ein Performance-Monitoring-System nur in größeren Projekten betrieben. Dabei wird häufig nicht auf erprobten Verfahren aufgebaut, sondern etwas spezifisches entwickelt, das sich nur schwer auf andere Projekte übertragen lässt. In kleinen und mittelgroßen Projekten wird im Normalfall überhaupt kein Monitoring-System eingebaut. Grund hierfür ist schlichtweg der zusätzliche Aufwand in Konstruktion, Realisierung und Betrieb. Beim bisherigen Stand der Monitoring-Techniken würde dieser in keinem Verhältnis zur Projektgröße stehen.

1 Vergleiche: *Architekturleitfaden Anwendungskonstruktion* 1.

Deshalb wird hierauf verzichtet. Konsequenzen daraus sind dann ggf. Ratlosigkeit und Ratespiele beim Support, wenn Ursachen für Performanceeinbrüche nicht gefunden werden können.[2]

Performance-Monitoring-Systeme können auch zum Nachweis bzw. zur Messbarkeit von nichtfunktionalen Anforderungen (NFAs) genutzt werden. Insbesondere bei der Überprüfung von NFAs, die die Effizienz und Zuverlässigkeit einer Anwendung beschreiben (wie z.B. durchschnittliche Antwortzeiten von bestimmten Use-Cases), kann das Performance-Monitoring zur Anforderungsbestätigung eingesetzt werden.

Zusätzlich zur Hilfe bei der Problemanalyse und zum Nachweis von NFAs kann Performance-Monitoring genutzt werden, um Nutzungsprofile zu ermitteln. D.h. anhand der produzierten Monitoring-Daten kann bestimmt werden, wie die Anwendung belastet wird, um ggf. darauf zu reagieren und sie entsprechend anzupassen.

2011 schrieb Stefan Eberlein seine Diplomarbeit[3] in Kooperation mit Capgemini. Darin beschreibt er u.a. grundlegende Techniken des Performance-Monitorings und der Analyse der erhobenen Daten, weißt auf typische Probleme hin und vergleicht drei verschiedene Frameworks, die zur Realisierung genutzt werden können. Ein direkt und praktisch anwendbares Verfahren wurde jedoch nicht beschrieben.

Genau an diesem Punkt setzt diese Arbeit an, indem sie Wege aufzeigt, wie ein Performance-Monitoring in Java-Projekten realisiert werden kann, ohne dass eine umfangreiche Einarbeitung in das Thema notwendig ist. Auch eine Methode zur Analyse der Monitoring-Daten wird vorgestellt, die „out of the box“ funktioniert und die erweitert werden kann, um Lösungen für individuelle Problemstellungen zu entwickeln. Die konkreten Ergebnisse sind in Anhang A in einem Leitfaden für die praktische Anwendung zusammengefasst.

2 Vergleiche: *Einfaches Performance-Monitoring von Java-Anwendungen* 2.
3 Vergleiche: *Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance–Monitoring* 3.

1.3 Zielsetzung

Ziel dieser Arbeit ist es, anhand von Forschungsergebnissen einen Leitfaden zu entwickeln, mit dessen Hilfe es möglich ist, ein Performance-Monitoring-System in kleineren und mittleren Java-Projekten zu integrieren. Dieses System soll es den Entwicklern ermöglichen, Antwortzeiten von Methoden zu ermitteln und deren Durchschnitts-, Minimal- und Maximalwerte innerhalb eines Zeitintervalls zu beobachten. Eine Visualisierung dieser Kennzahlen als Diagramm über einen Zeitraum ist eine weitere Anforderung. Des Weiteren soll es sich um ein fachliches Performance-Monitoring handeln, d.h. die erhobenen Daten sollen in einen fachlichen Kontext gesetzt werden, um Rückschlüsse über Use-Cases und Nutzerverhalten ziehen zu können. Hierzu muss auch ein entsprechendes Verfahren zur Analyse der gewonnenen Daten mitgeliefert werden.

Hauptaugenmerk bei der Entwicklung des Leitfadens soll auf der Simplizität seiner Anwendung liegen. Nutzer sollen also möglichst wenig Zeit aufwenden müssen, um das Performance-Monitoring-System zu integrieren. Darüber hinaus sollte das Verfahren so erweiterbar sein, dass die erhobenen Daten um projektspezifische Daten erweitert werden können. Dementsprechend sollte auch das Analyseverfahren erweiterbar sein.

2 Grundlagen

Im Folgenden werden die wichtigsten Grundlagen erläutert, die zum Verständnis dieser Arbeit notwendig sind. Hierbei wird auf einige Ergebnisse der Arbeit von Eberlein[1] eingegangen, da an vielen Stellen an diese angeknüpft wird.

2.1 Was ist Performance-Monitoring?

Wikipedia[2] definiert den Begriff des Monitorings folgendermaßen:

> *„Monitoring ist ein Überbegriff für alle Arten der unmittelbaren systematischen Erfassung (Protokollierung), Beobachtung oder Überwachung eines Vorgangs oder Prozesses mittels technischer Hilfsmittel (zum Beispiel Langzeit-EKG) oder anderer Beobachtungssysteme. Dabei ist die wiederholte regelmäßige Durchführung ein zentrales Element der jeweiligen Untersuchungsprogramme, um anhand von Ergebnisvergleichen Schlussfolgerungen ziehen zu können ...“*

Das *Performance-Monitoring* (im Folgenden auch *PM* oder nur *Monitoring*) ist eine Form des Monitorings in der IT, bei dem ein Software-System überwacht wird, um Schlüsse über Engpässe in der Datenverarbeitung (*Bottlenecks*), fehlerhafte Programmierung (*Bugs*) und Belastung eines Softwaresystems (und evtl. dessen zugrunde liegende Hardware) zu ziehen. Eberlein unterteilt es in technisches und fachliches PM[1].

Technisches PM umfasst die Überwachung technischer Ressourcen der Hardware (CPU-Auslastung, Speicherverbrauch, etc.) und feingranulare Zeitmessungen auf

1 Vergleiche: *Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance–Monitoring* 3.
2 Vergleiche: Wikipedia – *Monitoring* 4.

Funktionsebene. Auch die Ermittlung von sog. *Roundtrip-Zeiten*, d.h. die Messung der Dauer eines Aufrufs durch das System, ist diesem Aspekt des PM zuzuordnen. Das gleiche gilt für die Erstellung von Aufrufhierarchien (engl. *Traces*). Hierbei werden die durchlaufenen Methoden einer Transaktion geordnet aufgelistet. Die Auswertung der durch das technische PM erzeugten Daten setzt technisches Knowhow voraus. Deshalb ist das technische PM für die Lokalisierung und Behebung von Performance-Problemen nur bedingt geeignet.

Das fachliche PM setzt auf dem technischen PM auf und erweitert es, indem es die gewonnenen Daten in einen fachlichen Kontext setzt. Hierzu muss die Programmlogik des technisches PM um die Integration des fachlichen Kontextes erweitert werden. Ist dies erfolgreich durchgeführt worden, so lassen sich mit den ermittelten Daten Antworten auf konkrete fachliche Fragen finden, wie z.B.:

- Welchem Use-Case, welchem Nutzer oder welcher Nutzergruppe ist Transaktion *X* zuzuordnen?
- Wie lange dauert die Ausführung von Use-Case *X*?
- Wie sieht die Aufrufhierarchie von Use-Case *X* aus?
- Welche Use-Cases werden am häufigsten aufgerufen?

Auch der Nachweis von NFAs zum Thema Performance lassen sich mit fachlichem PM erbringen. Des Weiteren können die gewonnenen Daten genutzt werden, um Performance-Probleme und Fehlerquellen zu lokalisieren und dies bereits während der Entwicklung der Software, sofern es frühzeitig im System integriert wurde. Und schließlich dient es der Erstellung von Nutzungsprofilen, d.h. es ermöglicht eine genaue Analyse, wie die Software genutzt wird, um sie dann möglicherweise den Anforderungen entsprechend anzupassen.

2.2 Aspektorientierte Programmierung

Das im Rahmen dieser Arbeit entstandene Verfahren zum Performance-Monitoring von Java-Anwendungen bedient sich der Technik der *Aspektorientierten Programmierung* (AOP). Diese soll im Folgenden kurz vorgestellt werden.

Aspektorientierte Programmierung ist ein Programmierparadigma für die objektorientierte Programmierung, um generische Funktionalitäten über mehrere Klassen hinweg zu verwenden[3]. Der Teil eines Aspekts, der den einzufügenden Quellcode enthält nennt sich *Advice*. Die sog. *Pointcuts* eines Aspekts definieren an welchen Stellen der Advice in den bestehenden Programmcode integriert werden soll. Dies kann vor, nach oder um einen Methodenaufruf herum geschehen. Diese Form der Codeintegration wird auch als *Weaving* bezeichnet. Weaving geschieht meist zur Compilezeit, teilweise sogar erst zur Laufzeit. Auf diese Weise ist es möglich, Programmcode, der in verschiedenen Komponenten und Schichten ähnlich benötigt wird (*cross-cutting-concerns*) – wie es bspw. beim Logging der Fall ist –, zu implementieren, ohne ihn mit fachlichem Code der Anwendung zu vermischen (*Tangling*) oder ihn über mehrere funktionale Komponenten zu verstreuen (*Scattering*).[4]

2.3 Monitoring-Tools

Dieser Abschnitt erklärt kurz, wie Monitoring-Tools üblicherweise aufgebaut sind. So entspricht auch das in Abschnitt 3.1 genauer beschriebene Framework, welches im Rahmen dieser Arbeit eingesetzt wurde, diesem Muster.

Monitoring-Tools sind grundsätzlich immer sehr ähnlich aufgebaut und können in drei Komponenten unterteilt werden[5]. In Abbildung 2.1 sind die drei Komponenten *Analyse*, *Datenspeicherung* und *Messsonden* dargestellt.

3 Vergleiche: Wikipedia – *Aspektorientierte Programmierung* 5.
4 Vergleiche: *Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance–Monitoring* 3.
5 Vergleiche:
Continuous Monitoring of Software Services: Design and Application of the Kieker Framework 6.

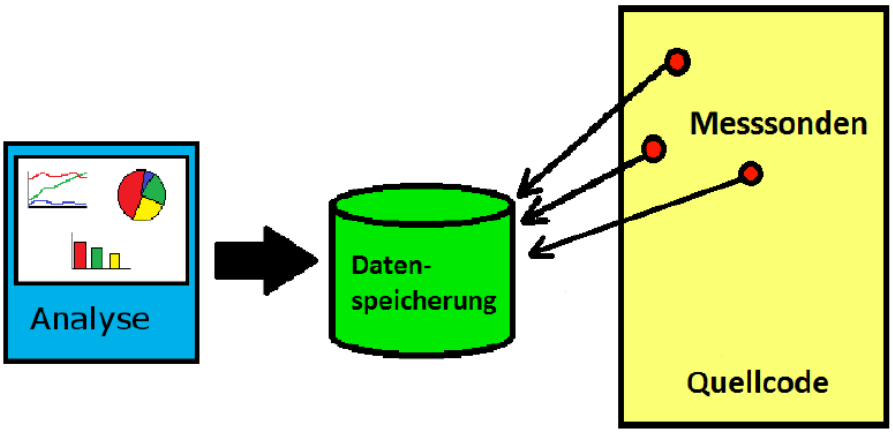


Abbildung 2.1: Aufbau von PM-Tools
(Quelle: *Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance–Monitoring*[3])

Die Messsonden (auch *Monitoring-Probes* oder kurz *Probes*) werden in den Quellcode der Software integriert, um die Daten des PM aufzuzeichnen. Man spricht hierbei auch von einer Instrumentierung des Codes[6]. Wie viele Probes eingesetzt werden und welche Daten sie festhalten ist von Tool zu Tool unterschiedlich und differiert auch je nach überwachter Anwendung. Jedoch werden sie immer an einzelnen Methoden platziert, sodass das wichtigste Datum – die Antwortzeit (engl. *Responsetime*) – gemessen werden kann. Dies geschieht durch das Festhalten der Zeitstempel vor und nach der überwachten Methode.

Die von den Monitoring-Probes gesammelten Daten werden dann an die zweite Komponente des PM-Tools – die Datenspeicherung – weitergereicht. Diese kann lokal oder zentral in einer Datenbank oder im Dateisystem stattfinden. In der Praxis haben sich die Persistierung in einer zentralen Datenbanken bei großen oder dezentral (lokal) im Dateisystem bei kleinen Projekten als die gängigsten Lösungen herausgestellt[7].

Die letzte Komponente – die Analyse – wird benötigt, um aus den aggregierten Daten sinnvolle Schlüsse zu ziehen. Die Möglichkeiten der Analyse sind mannigfaltig. So können hier Kennzahlen ermittelt, Diagramme produziert oder Auffälligkeiten

6 Vergleiche: *Dynamische Analyse mit dem Software-EKG* 7.
7 Vergleiche: *Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance–Monitoring* 3.

entdeckt werden. Auch die Vorteile des fachlichen Monitorings kommen hier zum tragen, da Daten entsprechend sortiert und gefiltert werden können, um Rückschlüsse über Use-Cases und Nutzerverhalten zu ziehen. Die Analyse-Komponente ist nicht zwangsläufig Teil eines Monitoring-Tools. In vielen Fällen können die produzierten Daten auch mit externen Programmen verarbeitet werden. Die meisten Monitoring-Tools – so auch das in dieser Arbeit verwendete Framework *Kieker* – enthalten jedoch auch diese Komponente, um ein abgeschlossenes Monitoring anzubieten.

3 Entwicklung des Leitfadens

Das nun folgende Kapitel beschreibt die Entstehung des in Anhang A aufgeführten Dokuments (im Folgenden auch *PM-Leitfaden* genannt). Dabei wird genauer auf Entscheidungsfindungen eingegangen und erläutert warum die beschrittenen Wege anderen Optionen vorgezogen wurden. Zusätzlich werden wichtige Fallstricke sowie Möglichkeiten zur Erweiterung und Individualisierung des PM aufgezeigt.

3.1 Das Kieker-Framework

Als Monitoring-Tool für den PM-Leitfaden wurde *Kieker*[1] ausgewählt, da es schon von Eberlein[2] im Vergleich zu *OpenARM* und *JAMon* mit Abstand am besten bewertet wurde (Bewertung 75% gegenüber 35% und 40%).

Kieker wurde an der *Christian-Albrechts-Universität zu Kiel* entwickelt und 2009 in einem Forschungsbericht[3] vorgestellt. Die in Java geschriebene Software wurde unter der *Apache License 2.0* veröffentlicht und darf frei verwendet, modifiziert und verteilt werden. Das letzte Release (1.9) erschien am 16.04.2014 und da bisherige Releases sehr zuverlässig in Abständen von etwa einem halben Jahr erschienen sind, ist mit kontinuierlicher Weiterentwicklung zu rechnen.

Das Werkzeug ist mit mittlerweile fast 500 dokumentierten Klassen in der API sehr umfangreich. Dennoch ist die Verwendung verhältnismäßig einfach, da ein

1 Vergleiche:
Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis 8.
2 Vergleiche: *Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance–Monitoring* 3.
3 Vergleiche:
Continuous Monitoring of Software Services: Design and Application of the Kieker Framework 6.

User-Guide[4] existiert, der die Anwendung an einfachen Beispielen erklärt, die mit den wenigen wichtigen Klassen auskommen. Zusätzlich stehen für alle wichtigen Komponenten Interfaces zur Verfügung, sodass bestehende Klassen leicht angepasst oder durch eigene ersetzt werden können.

Seit Eberlein Kieker 2011 evaluierte, wurde das Tool stetig weiterentwickelt und bietet mittlerweile u.a. folgende zusätzliche Features:

- Ein- und Ausschalten einzelner oder mehrerer Monitoring-Probes zur Laufzeit mittels regulärer Ausdrücke
- umfangreiche Konfiguration des Monitorings über eine Properties-Datei
- eine Auswahl von verschiedenen Monitoring-Probes
- eine WebGUI zur Konfiguration und Ausführung des plugin-basierten Analyse-Frameworks (s. Abschn. 3.6.2)

3.1.1 Aufbau des Tools

In Abbildung 3.1 sind die wesentlichen Komponenten des Kieker-Frameworks dargestellt.

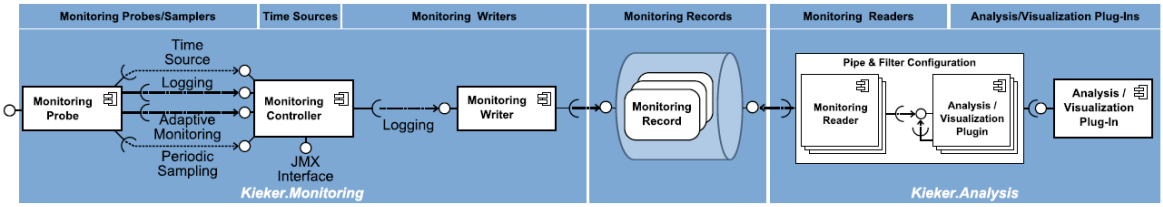


Abbildung 3.1: Die Komponenten des Frameworks
(Quelle: *Kieker User Guide*[9])

Wie bereits in Abschnitt 2.3 beschrieben, sind die Monitoring-Probes die Komponente, die in den Quellcode der überwachten Anwendung integriert wird. Im Betrieb kommunizieren sie mit dem *Monitoring-Controller*. Dieser stellt als Singleton-Instanz

4 Vergleiche: *Kieker User Guide* 9.

die Kernkomponente des Monitorings für die Datensammlung und -aufzeichnung dar. Er wird anhand von Properties in einer Konfigurationsdatei initialisiert. Diese Properties enthalten z.B. die Information welche Zeitquelle oder welche Klasse zum Schreiben der Monitoring-Daten verwendet werden soll (*Monitoring-Writer*). Ist das *adaptive Monitoring* eingeschaltet (ebenfalls über eine Property konfigurierbar), so kann in einer weiteren Textdatei mittels regulärer Ausdrücke festgelegt werden, welche Messsonden ausgeschaltet sind. Auch diese Information erhalten die Probes vom Monitoring-Controller. So können sie sogar zur Laufzeit ein- und ausgeschaltet werden.

Wann immer eine Messsonde Daten aufzeichnet, erstellt sie einen neuen *Monitoring-Record*. Monitoring-Records sind Objekte, die alle Monitoring-Daten einer Ausführung einer überwachten Methode enthalten. Die Monitoring-Probe reicht diese an den Monitoring-Controller weiter, der sie wiederum an den Monitoring-Writer leitet. Je nachdem welche Klasse für den Writer gewählt wurde, wird der Record dann in irgendeiner Form persistiert. In dieser Arbeit wird jedoch ausschließlich die Persistierung von Monitoring-Records in lokalen Textdateien betrachtet.

Analog zum Monitoring-Writer müssen die gespeicherten Records von einem *Monitoring-Reader* eingelesen werden, um analysiert werden zu können. Auch dieser muss für die entsprechende Form der Persistierung geeignet sein. Die eingelesenen Records können dann an die Plugins eines „Analyse-Netzwerkes“ weitergegeben werden, welches die Daten weiterverarbeitet. Eine ausführlichere Beschreibung der Analyse mit Kieker findet sich in Abschnitt 3.6.

3.1.2 Technik der Messsonden

Abbildung 3.2 zeigt die wesentliche Funktionsweise einer Monitoring-Probe von Kieker im Sequenzdiagramm.

Jede Messsonde umschließt eine Funktion, die sie überwacht (*überwachte Komponente*). So wird die Probe in einen Teil vor der überwachten Funktion und in einen danach geteilt.

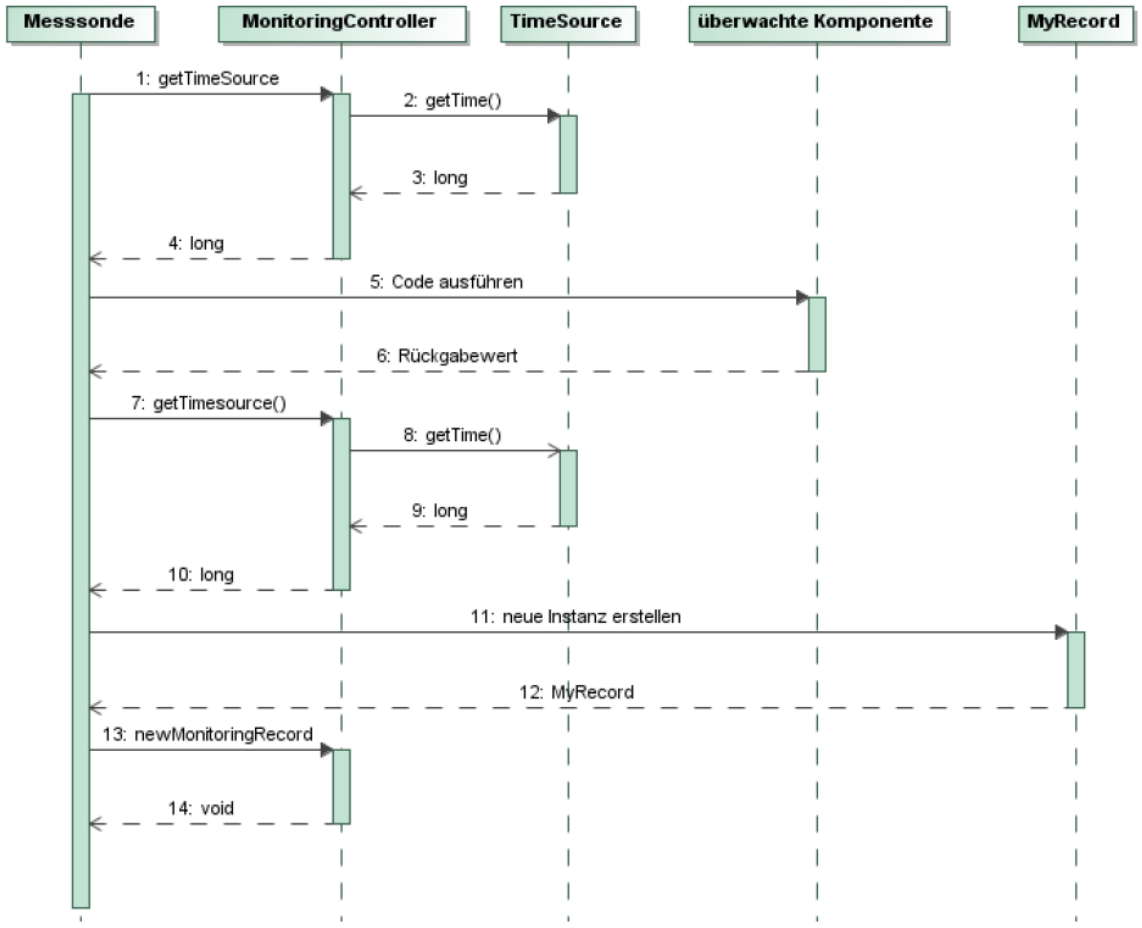


Abbildung 3.2: Sequenzdiagramm einer Kieker-Messsonde
(Quelle: *Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance–Monitoring*[3])

Der erste Teil zeichnet vor allem einen Zeitstempel (engl. *Timestamp*) *tin* auf. Hierzu kommuniziert sie mit dem Monitoring-Controller (*getTimeSource.getTime()*). Prinzipiell können zu diesem Zeitpunkt aber auch schon alle anderen Daten akquiriert werden, die vor Ausführung der überwachten Methode feststehen. So könnte hier bspw. schon die Methodensignatur oder der Host-Name aufgezeichnet werden.

Im zweiten Teil – nach Ausführung der überwachten Methode – wird ein zweiter Timestamp *tout* genommen (nochmals *getTimeSource.getTime()*) und somit die Dauer des Methodenaufrufs (*tout - tin*) festgehalten. Zusätzlich können jetzt alle

Daten, die erst nach Beendigung des Funktionsaufrufs feststehen, gespeichert werden; z.B. ein Indikator, ob die Funktion erfolgreich beendet wurde. Abschließend wird ein neuer Monitoring-Record erzeugt und an den Controller weitergereicht (*newMonitoringRecord()*).

3.2 Code-Instrumentierung

Für die Integration der Monitoring-Probes in den Quellcode stand schon früh die Verwendung von AOP fest. Die Vermeidung von Scattering und Tangling spricht eindeutig für diese Technik. Ansätze, die Codegenerierung voraussetzen oder empfehlen schieden aus, da nicht davon ausgegangen werden konnte, dass jedes Projekt, das vom PM-Leitfaden profitieren sollte, Codegenerierung verwendet. Die manuelle Instrumentierung ist zwar aufgrund von Fehleranfälligkeit, Scattering und Tangling nicht ideal, wurde aber als alternative Lösung gewählt, da sie sehr flexibel ist und somit immer als Notfallvariante dienen kann.

Wie die Code-Instrumentierung konkret durchgeführt wird, ist im PM-Leitfaden dokumentiert[5].

3.2.1 AspectJ oder Spring-AOP

Für die Realisierung der AOP-Variante stand neben *Spring-AOP* (die Umsetzung des AOP-Paradigmas im Spring-Framework) auch *AspectJ* zur Debatte. Für AspectJ sprach die Tatsache, dass Spring-AOP nur Klassen überwachen kann, die als Spring-Beans implementiert sind. Außerdem unterstützt AspectJ eine größere Zahl von Pointcut-Arten. So ist es mit Spring-AOP bspw. weder möglich, eine Methode, die nicht als *public* deklariert wurde, noch irgendeinen Konstruktor zu unterbrechen. Da das AOP-Paradigma in Spring mit Stellvertretern[6] (sog. *Proxies*) realisiert wird, ist es ebenfalls nicht möglich, Methoden zu überwachen, die direkt von anderen

5 Anhang A, Abschnitt 2.1. für die Instrumentierung mit Spring-AOP, Abschnitt 2.2 für die manuelle Variante

6 Eine Instanz, die zwischen Caller und Callee zwischengeschaltet wird um Unterbrechungsfunktionalität zu realisieren.

Methoden des selben Objekts aufgerufen werden. Die Vorteile von Spring-AOP liegen vor allem in der einfacherer Handhabung.[7]

Die Entscheidung fiel schließlich auf Spring-AOP. Grund hierfür war insbesondere die Tatsache, dass viele Projekte bei Capgemini das Spring-Framework bereits nutzen. So müssen keine zusätzlichen Bibliotheken importiert werden, die den „Technologie-Stack“ unnötig vergrößern. Weiterhin stellte sich heraus, dass wichtige fachliche Logik in den meisten Fällen in *public* Methoden von Spring-Beans implementiert ist, was die Notwendigkeit von AspectJ weiter schmälert. Da die Option der manuellen Instrumentierung ebenfalls bestehen sollte, war es zudem naheliegend, diese auch ergänzend zu nutzen, falls es doch einmal vorkommen sollte, dass eine Methode observiert werden soll, die von Spring-AOP nicht erreicht werden kann. In Extremfällen, bei denen die Möglichkeiten von Spring-AOP den Anforderungen nicht genügen sollten, wäre es sogar machbar, Spring-AOP um den AspectJ-Compiler zu erweitern[8].

3.2.2 Ein Stolperstein in Spring-AOP

Wie bereits erwähnt, verwendet Spring Proxies, um die Unterbrechungslogik für AOP umzusetzen. Hierbei gibt es zwei Möglichkeiten, diese Proxies von Spring erstellen zu lassen. Voreingestellt werden *JDK-Dynamic-Proxies* verwendet. Diese sind interface-basiert, d.h. sie implementieren ein aufgerufenes Interface. Mit ihnen ist es also nur möglich, Aufrufe zu unterbrechen, bei denen auf eine Klasse zugegriffen wird, die ein Interface implementiert. Alle anderen Klassen lassen sich mit dieser Methode nicht unterbrechen.

Da die Verwendung von Interfaces für das Monitoring mit Spring-AOP aber nicht zwingend erforderlich sein sollte, musste auf die zweite Möglichkeit zurückgegriffen werden: *CGLIB* (Code Generation Library) -*Proxies*. Bei dieser Art der Erzeugung von Stellvertretern werden diese als Unterklassen der Aufgerufenen Klasse erzeugt. So können diese auch unterbrochen werden, wenn sie kein Interface implementieren. Einziger Nachteil dieser Variante ist, dass sie nicht bei Methoden funktioniert, die

7 Vergleiche: *The Spring Framework - Reference Documentation* 10, Abschnitt 6.1.2.
8 Vergleiche: *The Spring Framework - Reference Documentation* 10, Abschnitt 6.8.

als *final* deklariert wurden, da diese nicht in einer Unterklasse überschrieben werden können.

Obwohl Spring laut Dokumentation[9] automatisch CGLIB verwendet, wenn JDK nicht ausreicht, stellte sich in der Praxis heraus, dass bei einigen Versionen von Spring die Verwendung von CGLIB-Proxies über ein Attribut in der entsprechenden Konfigurationsdatei erzwungen werden muss.

3.3 Die Monitoring-Daten

Wie schon in Abschnitt 3.1.1 beschrieben, werden die vom PM aufgezeichneten Daten von den Monitoring-Probes als Monitoring-Records aufgezeichnet. Die Auswahl der in den Records enthaltenen Monitoring-Daten für den PM-Leitfaden orientierte sich an den Vorgaben des Architekturleitfadens[10]. Auf ein Feld für den Rückgabewert der observierten Methode oder andere generische Felder musste jedoch verzichtet werden, da eine Umsetzung von generischen Feldern in den Monitoring-Records ohne eine komplexe Umstrukturierung der vorgegebenen Klassen von Kieker nicht möglich war. Wie sich der Inhalt der verwendeten Monitoring-Records im Detail zusammensetzt, ist in Anhang A (Abschn. 2.1. – Ende) beschrieben.

Obwohl Kieker in der Lage ist, die gesammelten Daten mittels verschiedener, austauschbarer Writer-Klassen in unterschiedlicher Form zu speichern (z.B. in einer Datenbank oder in Archiv-Dateien), wurden für die Persistierung einfache Text-Dateien im CSV-Format (*Comma-Separated-Values*) gewählt. Grund hierfür waren die einfache Handhabung und die damit verbundene Zeitersparnis. Die Konfiguration und Verwendung einer Datenbank hätten insbesondere bei der Analyse der Daten den Rahmen dieser Arbeit überschritten. Aus dem gleichen Grund wurde das Thema Datenreduktion – obwohl gerade im Betrieb unerlässlich – nicht behandelt. Zudem sollte es in den meisten Fällen keine Schwierigkeiten bereiten, dieses Problem unabhängig vom Rest des Verfahrens und individuell zu lösen.

9 Vergleiche: *The Spring Framework - Reference Documentation* 10, Abschnitt 7.5.3. ff.
10 Vergleiche: *Architekturleitfaden Anwendungskonstruktion* 1.

3.4 Vorstellung der vom Monitoring verwendeten Klassen

Im folgenden Abschnitt werden die eigens für den PM-Leitfaden entworfenen Klassen zur Erfassung der Monitoring-Daten betrachtet. Außerdem wird ein Blick auf die von Kieker bereitgestellten Klassen geworfen, die erweitert wurden, bzw. an denen sich bei der Implementierung orientiert wurde. Die für den PM-Leitfaden geschriebenen Klassen für die Analyse werden in Abschnitt 3.6.3 vorgestellt und sind auch im PM-Leitfaden ausführlich dokumentiert (s. Anhang A Abschn. 3.4.). Zusätzlich enthält das Javadoc-Dokument auf der dieser Arbeit beliegenden CD die Dokumentation aller Klassen der Packages `org.oasp.module.monitoring` und `org.oasp.module.monitoring.analysis`.

Die Klasse `kieker.monitoring.core.registry.`**`ControlFlowRegistry`**[11] wird von Kieker bereitgestellt, um Daten über mehrere Monitoring-Probes hinweg für jeden Thread separat (*thread-local*) zu speichern. Kieker nutzt diesen Mechanismus in der Klasse `kieker.monitoring.probe.spring.executions.`**`OperationExecutionMethodInvocationInterceptor`**, um *Trace-ID*[12], *EOI*[13] und *ESS*[14] jedes überwachten Threads für die Dauer eines Aufrufes festzuhalten. Die für den PM-Leitfaden entwickelte Klasse `org.oasp.module.monitoring.`**`CustomControlFlowRegistry`** ergänzt die **`ControlFlowRegistry`** von Kieker um Felder für *Action-ID*[15], *Session-ID* und *User-ID*. Dies sind die drei Felder für den fachlichen Kontext, die auch in den Klassen `org.oasp.module.monitoring.`**`CustomOperationExecutionMethodInvocationInterceptor`** und `org.oasp.module.monitoring.`**`CustomOperationExecutionRecord`** ergänzt wurden (s. Abschn. 3.4.1).

3.4.1 Klassen für die Spring-AOP-Instrumentierung

Die Klassen für Monitoring-Probe und Monitoring-Record müssen immer gut aufeinander abgestimmt sein, da die Records von den Probes erzeugt und „befüllt“ werden.

11 Für alle Klassen deren Domain-Name mit `kieker` beginnt, vergleiche: *Die Kieker-API* 11.
12 Eine Korrelations-ID, über die Monitoring-Records einem Aufruf zugeordnet werden können.
13 *Execution Order Index*: Ein Index, der bei jeder Messsonde eines Aufrufs inkrementiert wird.
14 *Execution Stack Size*: Gibt die Schachtelungstiefe eines Aufrufs an.
15 Enthält Use-Case-Name oder ID der Use-Case-Instanz.

D.h. innerhalb der Probe muss dafür gesorgt sein, dass alle Daten die im Record festgehalten werden sollen, auch akquiriert werden.

Kieker stellt mit den Klassen `kieker.common.record.controlflow.`**`OperationExecutionRecord`** und `kieker.monitoring.probe.spring.executions.`**`OperationExecutionMethodInvocationInterceptor`** ein „Record-Probe-Duo“ zur Verfügung, das mit Spring-AOP ohne weitere Anpassungen genutzt werden kann. Auch die Auswahl von Monitoring-Daten, die mit ihnen aufgezeichnet wird, ist sinnvoll und enthält die wichtigsten Datensätze wie Methodensignatur, *tin*, *tout*, *eoi*, *ess* und *Trace-ID*.

Für den PM-Leitfaden wurden diese Klassen um Felder für *Action-ID*, *Session-ID* und *User-ID* erweitert bzw. ergänzt. Die resultierenden Klassen sind `org.oasp.module.monitoring.`**`CustomOperationExecutionMethodInvocationInterceptor`** und `org.oasp.module.monitoring.`**`CustomOperationExecutionRecord`**, die für die bevorzugte Instrumentierungsvariante mit Spring-AOP entwickelt wurden.

Mit der Klasse `org.oasp.module.monitoring.`**`ContextProvider`** ist eine beispielhafte Lösung zur Initialisierung des fachlichen Kontextes gegeben, die ebenfalls Spring-AOP verwendet. Da diese Klasse die Felder *Action-ID*, *Session-ID* und *User-ID* an der **`CustomControlFlowRegistry`** registriert und deregistriert, muss die Methode `ContextProvider.`**`invoke()`** immer aufgerufen werden, bevor irgendeine Monitoring-Probe innerhalb des Aufrufes durchlaufen wird. Wie dies in der Praxis aussehen kann, ist im PM-Leitfaden erläutert (Anhang A Abschnitt 2.1. Punkt 5).

3.4.2 Klassen für die manuelle Instrumentierung

Für die manuelle Integration wurde mit der Klasse `org.oasp.module.monitoring.`**`ManualOperationExecutionProbe`** eine alternative Monitoring-Probe implementiert, für deren Verwendung Spring-AOP nicht notwendig ist. Stattdessen müssen die statischen Methoden `ManualOperationExecutionProbe.`**`before()`** und `ManualOperationExecutionProbe.`**`after()`** zu Beginn bzw. am Ende der überwachten Funktion per Hand eingefügt werden. (s. Anhang A Abschn. 2.2.)

Zusätzlich wurde die Beispielhafte Lösung zur Bereitstellung des fachlichen Kontextes (`org.oasp.module.monitoring.`**`ContextProvider`**) angepasst, so dass sie ebenfalls manu-

ell integriert wird und auf einen Aspekt verzichtet. Analog zur Messsonde wurde sie `org.oasp.module.monitoring.ManualContextProvider` genannt.

3.4.3 Erweiterung von Monitoring-Probes und -Records

Da Erweiterbarkeit eine wichtige Anforderung an den PM-Leitfaden war, erklärt er in Abschnitt 2.7. (s. Anhang A), wie die zur Verfügung gestellten Probes und Records erweitert bzw. ergänzt werden können. Der vollständige Quellcode aller entwickelten Klassen, sowie das Javadoc-Dokument (s. CD) tragen zusätzlich dazu bei, dass Java-Entwickler die Akquirierung der Monitoring-Daten individuell erweitern können.

3.5 Untersuchung des vom Monitoring verursachten Performance-Overheads

Um sicherzustellen, dass die Instrumentierung des Codes mittels Spring-AOP die Performance der überwachten Anwendung nicht nennenswert beeinflusst, wurden zwei Messreihen durchgeführt. Diese sollten Laufzeiten einer Anwendung mit und ohne Integration der Monitoring-Probes vergleichen. Bei der für die Messungen instrumentierten Software handelte es sich um eine von Capgemini bereitgestellte Beispiel-Anwendung, die betriebsintern zu Demonstrations- und Schulungszwecke genutzt wird. Diese stellt ein Restaurantverwaltungssystem dar, mit dem Mitarbeiter, angebotene Speisen und Getränke, Tische und Rechnungen verwaltet werden können.

Für die Messreihen wurde das Tool *JMeter*[16] verwendet. Dieses versendete an die auf einem Tomcat-Server laufende Beispielanwendung über mehrere Stunden wiederholt eine zuvor festgelegte Abfolge von sieben http-Anfragen. Hierbei handelte es sich ausschließlich um lesende Zugriffe, um das Verhältnis von Performance-Overhead (verursacht durch das PM) zu Laufzeiten der Use-Cases nicht durch

16 Vergleiche: *Apache JMeter™* 12.

längere Schreibzugriffe zu verzerren. JMeter wurde so konfiguriert, dass vier Prozesse die Anfrage-Abfolgen gleichzeitig durchführten. So wurden bei jeder Messreihe in einem Zeitraum von etwa sieben Stunden über 1.330.000 http-Anfragen an den Server gesendet.

Bei der Messreihe mit instrumentiertem Code wurden pro http-Anfrage zwischen null und zehn Monitoring-Probes durchlaufen; pro Anfrage-Abfolge insgesamt 18 Probes. Der Vergleich der Messreihen ergab, dass eine http-Anfrage an die Anwendung ohne Messsonden im Durchschnitt eine Antwortzeit von 90 ms hatte, während die durchschnittliche Antwortzeit bei instrumentiertem Code 87 ms betrug. Auf den ersten Blick scheint die Anwendung mit Probes schneller zu sein. Wahrscheinlicher ist jedoch, dass der Performance-Overhead so gering ist, dass er aufgrund von Laufzeitschwankungen von Datenbankanbindung, Netzwerk und Java-Garbage-Collection auf diese Weise nicht messbar ist.

Da auch eine zweite Durchführung beider Messreihen keine befriedigendere Antwort lieferte (92 ms ohne und 86 ms mit Messsonden), wurden die Laufzeiten der Monitoring-Probes in einer dritten Messreihe direkter betrachtet. Hierzu wurde ein eigenständiges, kleines Java Programm geschrieben, in welchem eine leere Methode einer einzelnen Spring-Bean in einer Schleife wiederholt aufgerufen wird. Das Programm misst 100 mal die Zeit für jeweils 100.000 Aufrufe der Methode und schreibt sie in eine Datei. Da die Code-Instrumentierung mit Spring-AOP Messsonden nur an Methoden von Spring-Beans einfügt, konnte so sehr gut die Laufzeit der Probes beobachtet werden. Nach Vergleich der vom Programm festgehaltenen Messzeiten mit und ohne Monitoring-Probes, konnte eine Laufzeit von durchschnittlich 3,5 µs pro Probe-Durchlauf berechnet werden.

Dieses Ergebnis bestätigt auch die Interpretation der Resultate der ersten Messreihen. Die Monitoring-Probes wirken sich bei 3,5 µs pro Probe-Durchlauf nur mit durchschnittlich 9 µs auf jede http-Anfrage aus. Das entspricht bei einer Durchschnittsantwortzeit von 90 ms nur 0,01%.

Somit sollte auch eine umfangreiche Instrumentierung mittels Spring-AOP unter normalen Umständen keinen nennenswerten Performanceverlust mit sich bringen.

3.6 Analyse der Monitoring-Daten

Um sinnvollen Nutzen aus den vom PM gesammelten Daten zu ziehen, müssen diese analysiert werden. Prinzipiell kann dies mit Programmen wie *Microsoft-Excel*, *OpenOffice-Calc* oder anderen Tools, die CSV-Dateien verarbeiten können, geschehen. Für den PM-Leitfaden sollte jedoch ein Verfahren entwickelt werden, das – genau wie die Aggregation der Daten – einfach und ohne lange Einarbeitungszeit anzuwenden ist. Gleichzeitig sollten aber die wichtigsten Information aus den Monitoring-Daten gezogen und für den Nutzer zugänglich gemacht werden. Weitere Anforderungen waren eine Visualisierungsmöglichkeit der über die Zeit aufgetragenen Antwortzeiten sowie die Erweiterbarkeit des Verfahrens.

Zu diesem Zweck war es naheliegend auf das plugin-basierte *Pipe-and-Filter-Framework* von Kieker zurückzugreifen, da die zur Analyse vorliegenden Daten nicht zusätzlich aufbereitet werden müssen und direkt eingelesen werden können. Unter der Verwendung einiger selbstgeschriebener Plugins sowie der *Kieker-WebGUI* sollte diese Lösung die oben genannten Ansprüche erfüllen.

3.6.1 Das Pipe-and-Filter-Framework

Mit dem Pipe-and-Filter-Framework bietet Kieker ein Framework, um Analyse-Netzwerke aus Reader-, Filter- und Repository-Plugins zu erstellen. In Abbildung 3.3 ist ein Beispiel für ein solches Netzwerk dargestellt.

Das Reader-Plugin *Pipe reader* liest in diesem Fall Monitoring-Records aus einer Pipe und sendet diese über seinen Output-Port an den Input-Port des Reader-Plugins *Response time filter*. Dieses vergleicht die Antwortzeiten der eingehenden Objekte mit einem zuvor konfigurierten Schwellwert. Ist der Schwellwert nicht überschritten, so wird der betrachtete Record über den Output-Port *validResponseTimes* weitergeleitet. Andernfalls wird er am anderen Output-Port (*invalidResponseTimes*) versendet. In jedem Fall landen die Records schließlich bei einem der *Response time printer* und werden von diesen auf dem Standard-Output-Stream ausgegeben.

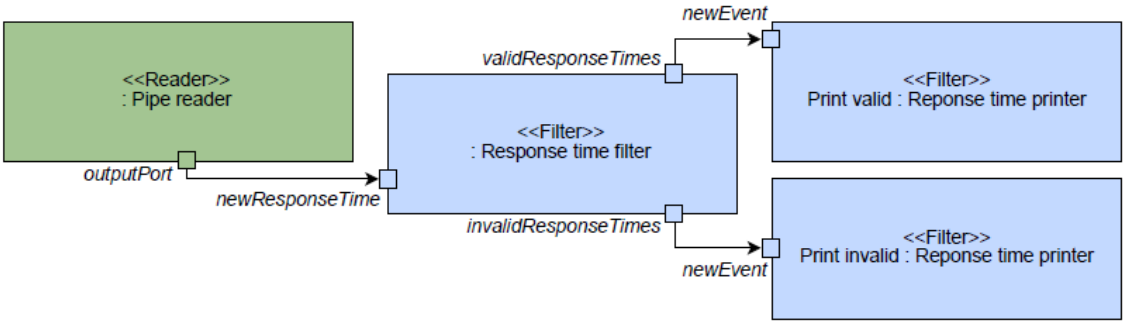


Abbildung 3.3: Beispiel Analyse-Netzwerk
(Quelle: *Kieker User Guide*[9])

Das Beispiel von Abbildung 3.3 verdeutlicht gut die grundlegende Funktionsweise der Plugin-Netzwerke. Zu Beginn der Struktur steht immer ein Reader-Plugin, das Monitoring-Records in irgendeiner Form[17] einliest. Dieses sendet die Record-Objekte dann weiter an ein oder mehrere Filter-Plugin(s). Innerhalb der Filter-Plugins können die Daten beliebig gefiltert, verarbeitet, aggregiert und an weitere Plugins verschickt werden. Hierbei muss auch die Klasse der versendeten Objekte nicht beibehalten werden. So ist es bspw. möglich die Records zuerst Anhand des fachlichen Kontextes (z.B. der *User-ID*) zu filtern und danach mit Objekten weiterzuarbeiten, die nur noch die Antwortzeiten enthalten. Lediglich die Klassen, die von *direkt* verbundenen Input- und Output-Ports empfangen bzw. versendet werden, müssen übereinstimmen. Der Funktionalität einzelner Plugins und insbesondere ihrer Kombination sind damit kaum Grenzen gesetzt.

17 Da im PM-Leitfaden die Records in Textdateien persistiert werden, wird ein Reader verwendet, der sie in dieser Form einlesen kann. Vergleiche Anhang A Abschn. 3.4.

3.6.2 Die Kieker-WebGUI

Die Plugin-Netzwerke des Pipe-and-Filter-Frameworks von Kieker können über eine Java-API erstellt und konfiguriert werden. Auch die Durchführung der Analyse lässt sich über diese API starten. Da aber insbesondere die Verknüpfung der verschiedenen Plugins über ihre In- und Output-Ports mühsam und fehleranfällig ist, entwickelt Kieker derzeit eine WebGUI, die für all dies eine grafische Oberfläche bietet. Abbildung 3.4 zeigt einen Screenshot der Kieker-WebGUI, in dem ein Plugin-Netzwerk erstellt wird.

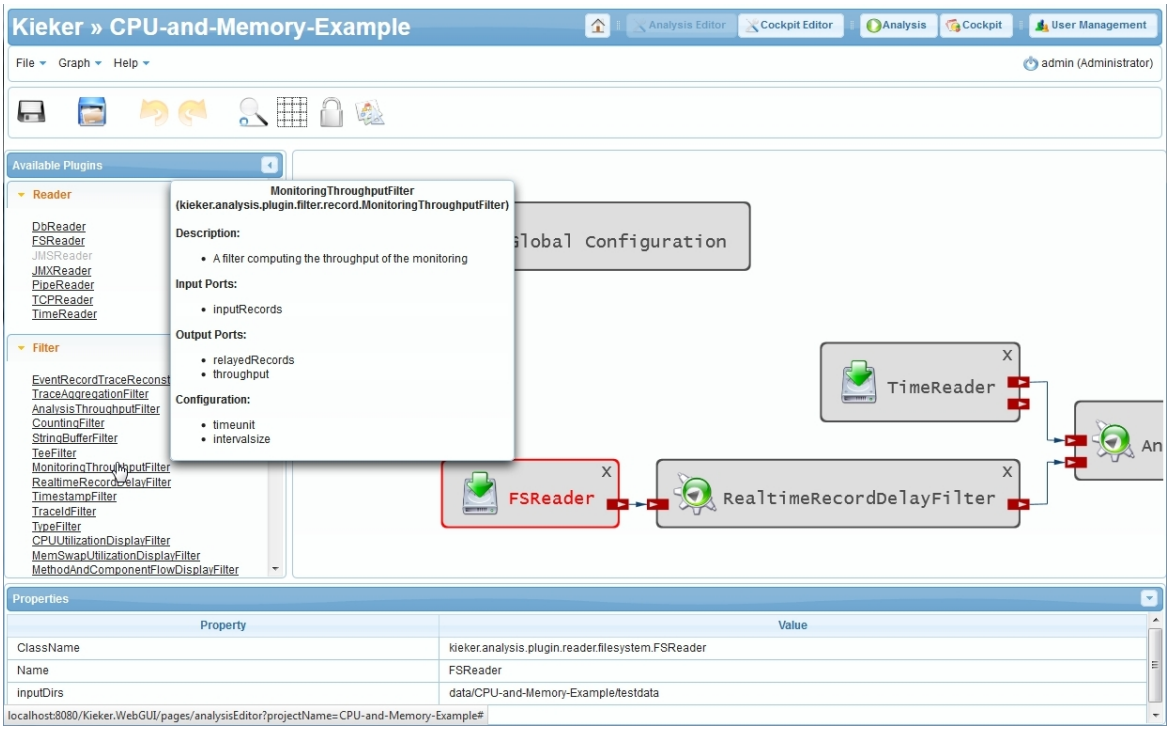


Abbildung 3.4: Screenshot aus der Kieker-WebGUI
(Quelle: *Everything in Sight: Kieker's WebGUI in Action*[13])

Eine Beta-Version dieser GUI, die für den PM-Leitfaden (s. Anhang A Abschn. 3.) verwendet wurde, bietet bereits folgende Features:

- Plugins können aus einer Liste ausgewählt und ihre Ports per Mausklick verbunden werden.

- Die Analyse kann konfiguriert werden, indem Properties für einzelne Plugins gesetzt werden.
- Analyse-Netzwerke können innerhalb der WebGUI ausgeführt werden.
- Analyse-Netzwerke können in spezifischen XML-Konfigurationsdateien gespeichert oder aus ihnen geladen werden.
- Einige speziell für die WebGUI entwickelte Plugins bieten Visualisierungsmöglichkeiten wie bspw. die Darstellung von Liniendiagrammen.

3.6.3 Erweiterung der Analyse mit Kieker durch selbstgeschriebene Plugins

Zum Zeitpunkt der Entstehung dieser Arbeit reichten die bereitgestellten Plugins von Kieker nicht aus, um die Anforderungen an die Analyse zu realisieren. Zusätzlich war die Dokumentation der Plugins[18] nicht ausreichend, um diese ohne Schwierigkeiten ihrem Potential entsprechend zu nutzen. Aus diesen Gründen war es notwendig, für den PM-Leitfaden eigene Plugins zu entwickeln. Im folgenden werden diese aufgelistet und kurz beschrieben. Eine ausführliche Dokumentation findet sich in Anhang A Abschnitt 3.4.

1. `org.oasp.module.monitoring.analysis.`**`OnlineFSReader`** – Dieses Reader-Plugin wurde entwickelt um eine Online-Analyse zu ermöglichen.
2. `org.oasp.module.monitoring.analysis.`**`RecordContentFilter`** – Ein Filter-Plugin, dass Monitoring-Records nach beliebigem Inhalt filtern kann.
3. `org.oasp.module.monitoring.analysis.`**`RecordResponseTimeProcessFilter`** – Dieses Plugin berechnet Kennzahlen aus den Antwortzeiten eingehender Monitoring-Records.
4. `org.oasp.module.monitoring.analysis.`**`TimeSeriesDisplayFilter`** – Mit Hilfe dieses Plugins ist es möglich Kennzahlen als Liniendiagramme über die Zeit grafisch darzustellen und zu vergleichen.

18 Vergleiche: *Die Kieker-API* 11.

5. `org.oasp.module.monitoring.analysis.`**`JMXDataProviderFilter`** – Ein Plugin, dass Kennzahlen über eine JMX-Schnittstelle bereitstellt, damit diese dann mit externen Tools weiterverarbeitet werden können.

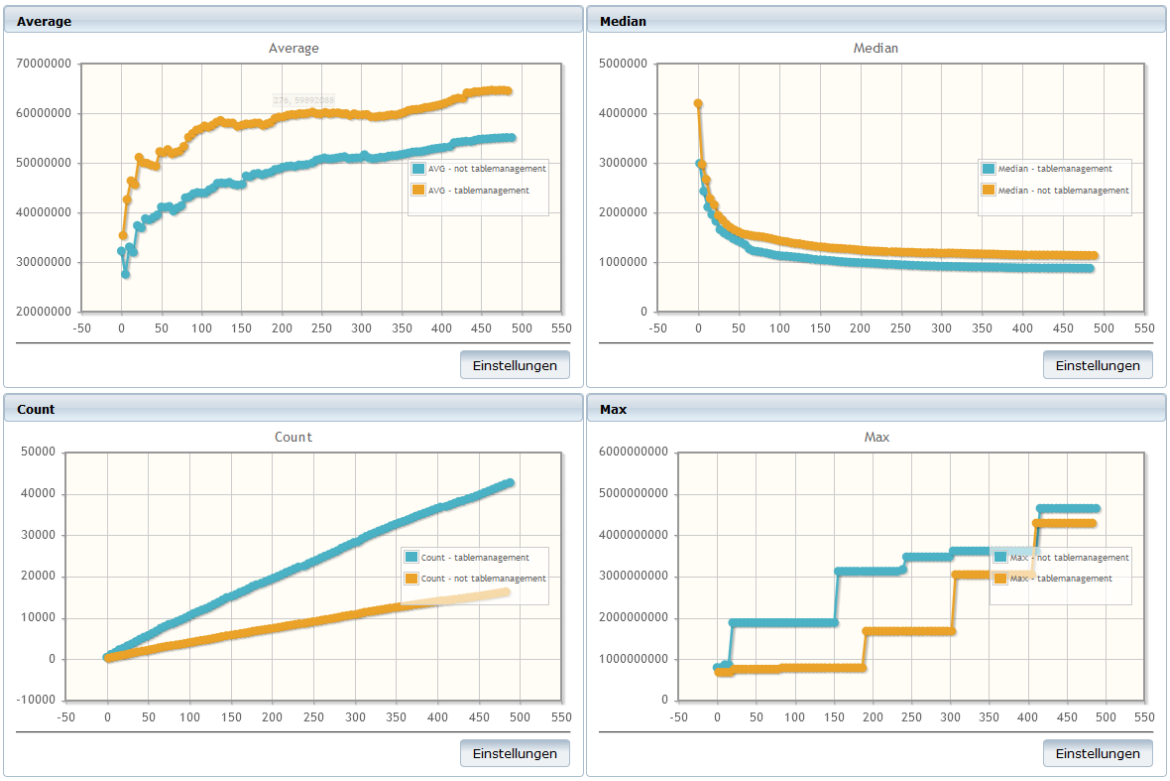


Abbildung 3.5: Visualisierung von Kennzahlen mit der Kieker-WebGUI (Quelle: *Screenshot aus der WebGUI*)

Unter Verwendung dieser zusätzlichen Klassen bietet die Analyse der Monitoring-Daten nach dem Verfahren des PM-Leitfadens folgende Features an:

- Alle Plugin-Klassen, um grundlegende Analyse der vom PM produzierten Daten durchzuführen. Dies beinhaltet die Visualisierung von Kennzahlen (Minimum, Maximum, Median und Durchschnitt) der Antwortzeiten der Monitoring-Probes. Hierbei können die Probes nach beliebigem Inhalt gefiltert werden. Eine beispielhafte Visualisierung ist in Abbildung 3.5 zu sehen.
- Online-Analyse, d.h. die Monitoring-Daten können während des laufenden Betriebs der überwachten Anwendung schon analysiert werden.

- Bereitstellung und periodische Aktualisierung der oben genannten Kennzahlen über eine JMX-Schnittstelle. So können die Kennzahlen eines online-überwachten Systems an externe Monitoring-Tools weitergeleitet und in andere Überwachungssysteme integriert werden.
- Eine Webanwendung mit grafischer Benutzeroberfläche, über die die Analyse einfach konfiguriert, gespeichert und gestartet werden kann.
- Ein Analysesystem, für das schnell und einfach weitere Plugin-Klassen geschrieben werden können, um weitere Features zu realisieren und Lösungen für individuelle Probleme umzusetzen.

Da die Erweiterbarkeit der Analyse ebenfalls eine wichtige Anforderung war, wurden auch diese Klassen mit vollständigem Quellcode bereitgestellt. Zusätzlich besteht ein Javadoc-Dokument (s. CD), das alle genannten Klassen enthält. Mit Hilfe dieser Dateien und dem Abschnitt *3.5. Individuelle Lösungen* des PM-Leitfadens (s. Anhang A), sollte es Java-Entwicklern möglich sein, ohne lange Einarbeitungszeiten die Analyse um eigene Plugins zu erweitern, um spezifische Probleme zu lösen.

4 Resümee

Im Rahmen dieser Arbeit sollte ein Leitfaden entstehen, der es Entwicklern ermöglicht, Java-Anwendungen ohne lange Einarbeitungszeit mit einem System zum fachlichen Performance-Monitoring zu versehen. Das Ergebnis ist in Anhang A in gedruckter Form oder auf der dieser Arbeit beiliegenden CD als HTML-Datei einzusehen.

Die Anforderung der Simplizität der Anwendung ist soweit möglich erfüllt worden. So geht der PM-Leitfaden auf alle zur Anwendung nötigen Aspekte ein, ohne sich dabei in Details zu verlieren. Zudem erfordert seine Verwendung keine spezifischen Kenntnisse von verwendeten Techniken oder Tools und kann ohne zusätzliche Recherchen eingesetzt werden. Diese Einfachheit steht im Kontrast zur Vielseitigkeit der Einsetzungsgebiete. Da ein Großteil der Projekte bei Capgemini das Spring-Framework bereits verwendet, wurde mit der Nutzung desselben zur Code-Instrumentierung jedoch ein Kompromiss gefunden, der diese Gegensätze bestmöglich vereint. Die Möglichkeit der manuellen Instrumentierung sollte die Zahl potentieller Nutzer noch erhöhen.

Ein Verfahren zur Analyse, um u.a. Kennzahlen wie Minimal-, Maximal- und Durchschnittswerte der Laufzeiten von beliebigen Methoden zu ermitteln, ist ebenfalls gegeben. Dieses beinhaltet wie gefordert die Möglichkeit zur Visualisierung der Kennzahlen innerhalb eines Zeitintervalls. Über diese Anforderung hinaus wurden Analyse-Plugins entwickelt, um die Analyse schon während des laufenden Betriebs parallel durchzuführen. So können zusätzlich zur Visualisierung die Kennzahlen auch über eine JMX-Schnittstelle bereitgestellt werden, um in etwaige externe Monitoring-Tools gespeist zu werden.

Durch die Integration von *User- Session-* und *Action-ID* in das PM-System ist ein fachlicher Kontext gegeben, zu dem die im Monitoring erhobenen Daten in Be-

zug gesetzt werden können. So ist es wie gewünscht möglich, Rückschlüsse über Nutzerverhalten und Use-Cases zu ziehen.

Leider sind nicht alle Klassen von Kieker zur direkten Erweiterung entwickelt worden. Die Klasse `kieker.monitoring.core.registry.`**`ControlFlowRegistry`** ist bspw. als Enum implementiert. Dennoch liefert der PM-Leitfaden mit vollständigem Quellcode und ausführlicher Javadoc-Dokumentation genug Informationen, damit ein Java-Entwickler die bestehenden Monitoring-Records und -Probes um zusätzliche Monitoring-Daten erweitern kann. Auch die Implementierung neuer Analyse-Plugins sollte mit diesen Informationen und evtl. Zuhilfenahme des Kieker-User-Guides[1] Programmierer vor keine allzu großen Probleme stellen.

4.1 Ausblick

Während der Entstehung dieser Arbeit kristallisierten sich einige Aspekte des Performance-Monitoring heraus, um die das vorgestellte Verfahren noch erweitert werden könnte, deren Realisierung im Rahmen dieser Arbeit aber nicht mehr durchführbar war. In diesem abschließenden Abschnitt wird kurz auf diese Aspekte eingegangen.

Der wohl wichtigste Aspekt des PM, für den im PM-Leitfaden kein Lösungsansatz geliefert wird, ist der der Datenreduktion. Ein im Betrieb laufendes PM-System kann große Mengen von Monitoring-Daten produzieren, die monoton wachsen. Daher sollte die Menge der Daten periodisch reduziert werden. Die einfachste Variante stellt hier das Löschen der ältesten Daten dar. Allerdings gehen bei dieser Variante Informationen verloren. Eine elegantere Methode besteht in der Aggregation der Daten. Hierbei werden Mittelwerte o.ä. über einen bestimmten Zeitraum berechnet. Statt allen Daten werden nur diese aggregierten Daten über lange Zeiträume festgehalten. Dies kann auch in verschiedenen Stufen geschehen. Je älter die Daten, desto größer die Zeiträume über die aggregiert wird. Eine weitere Alternative ist die Archivierung der Daten. Hierbei werden sie in einen Sekundärspeicher ausgelagert,

1 Vergleiche: *Kieker User Guide* 9.

um so die Datenmenge im Produktivsystem zu reduzieren. Sowohl bei der Aggregation als auch bei der Archivierung ist zu beachten, dass diese Methoden die Analyse der Daten einschränken bzw. verkomplizieren.[2]

Da der PM-Leitfaden für die Anwendung in kleinen und mittelgroßen Projekten gedacht ist, beschränkt er sich auf lokale Systeme. D.h. er liefert keinen direkten Lösungsweg, um ein PM auf verteilten Systemen durchzuführen. Zwar bieten Probes und Records mit Feldern für *Action-ID* und *Trace-ID* die Möglichkeit einer Korrelation der erhobenen Daten über mehrere Systeme hinweg, doch wird keine Lösung vorgestellt, um den fachlichen Kontext von einem System zum nächsten zu übertragen. Hierzu müsste bspw. beim Aufruf entfernter Methoden (Java-RMI) der fachliche Kontext zusammen mit dem zu übertragenen Objekt in ein Stellvertreter-Objekt verpackt werden, um auf Empfängerseite wieder entpackt zu werden[3]. Um das Verfahren um diese Funktionalität zu erweitern, müsste außerdem geklärt werden, ob die Daten weiterhin lokal gespeichert werden. Gängigere Praxis bei verteilten Systemen ist die Speicherung in einer zentralen Datenbank.

Zu guter Letzt könnte man den PM-Leitfaden um Module erweitern, die ihn für bestimmte Projektgrößen oder -typen abstimmen. So könnten vorgefertigte Record-Probe-Duos, die unterschiedliche Datensätze erheben, angefertigt werden oder Analyse-Plugins entwickelt werden, die bei bestimmte Arten von Projekten häufig gebraucht werden.

2 Vergleiche: *Architekturleitfaden Anwendungskonstruktion* 1.
3 Vergleiche: *Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance–Monitoring* 3.

Anhang

A Der Leitfaden für das Performance-Monitoring

Der folgende Anhang beinhaltet den PM-Leitfaden, der im Rahmen dieser Arbeit in Zusammenarbeit mit Capgemini entstanden ist. Die dieser Arbeit beiliegende CD beinhaltet u.a. das Original in HTML-Form. Bei der hier abgedruckten Version konnten die Hyperlink-Referenzen leider nicht übertragen werden.

Anleitung zum Performance-Monitoring von Java-Anwendungen mit dem Kieker-Framework

Johannes Mensing
version 1.0, August 2014

Inhaltsverzeichnis

1. Einleitung

Diese Anleitung ist entstanden, um einfaches Performance-Monitoring in kleinen und mittelgroßen Java-Projekten zu ermöglichen, ohne dabei einen unverhältnismäßig großen Aufwand betreiben zu müssen. Um einen genaueren Einblick in Kontext und Motivation dieser Arbeit zu bekommen, sieh Dir bitte die Powerpoint-Präsentation *Einfaches Performance-Monitoring von Java-Anwendungen* von Thomas Maier an.

Für die Entwicklung dieser Anleitung wurde Kieker-1.9 verwendet. Es ist möglich, dass zukünftige Releases kompatibel sind, dies ist aber nicht garantiert und muss getestet werden. Bei allen Fragen bezüglich Kieker ist immer ein Blick in den Kieker-User-Guide empfehlenswert.

Kieker ist unter der Apache License, Version 2.0 veröffentlicht und kann dementsprechend verwendet werden.

1.1. Was diese Anleitung bietet

Das hier vorgestellte Verfahren zum Monitoring erlaubt eine kontinuierliche Erfassung von Monitoring-Daten wie Responsetime, Methodensignatur, Thread-ID oder Host-Name. Außerdem werden einige Daten — wie etwa User-ID oder Session-ID — erfasst, mit denen sich die gewonnenen Informationen in Bezug zu einem fachlichen Kontext setzen lassen. Die Daten werden von Messsonden erfasst — sogenannte Monitoring-Probes — die normalerweise per Spring-AOP in den Code integriert werden. Alternativ wird die Möglichkeit der manuellen Codeinstrumentierung vorgestellt. Die Monitoring-Probes können gezielt an kritischen Punkten platziert werden, doch haben Performance-Messungen gezeigt, dass auch großzügige bis vollständige Codeinstrumentierungen kaum messbaren Performance-Overhead produzieren. Zudem sind die Monitoring-Probes auch zur Laufzeit an- und abschaltbar. Die produzierten Daten werden als Textdateien im CSV-Format persistiert, die von Tools wie z.B. Excel ausgewertet werden können.

Des Weiteren wird eine Möglichkeit zur Analyse der produzierten Daten unter Verwendung der Kieker-WebGUI aufgezeigt. Hiermit können die Daten beliebig gefiltert und Kennzahlen der Responsetimes — Minimum, Maximum, Median und Durchschnitt — ermittelt, visualisiert und bspw. für externe Tools über JMX bereitgestellt werden. Auch eine kontinuierliche Online-Analyse ist möglich. Da es sich bei der Analyse um ein plugin-basiertes Framework handelt, ist auch eine Erweiterung mit eigenen Plugins für individuelle Lösungen leicht umsetzbar.

1.2. Was diese Anleitung nicht bietet

Diese Anleitung liefert keine Konzept, wie die vom Monitoring erfassten Daten komprimiert oder anderweitig zusammengefasst werden können. D.h. es kann u.U. zu erheblichen Datenmengen kommen. Bei jeder in der Produktion laufenden Anwendung, die das hier vorgestellte Verfahren verwendet, sollte unbedingt darauf geachtet werden, die entstehenden Daten periodisch zu komprimieren und/oder zu löschen.

Zur Bereitstellung des fachlichen Kontextes wird keine universelle Lösung geliefert, da dies in Anbetracht der Unterschiedlichkeit der möglichen Kontexte und Softwareprojekte nicht umzusetzen war. Stattdessen werden Beispielhafte Lösungen vorgestellt, die in vielen Fällen aber mit kleinen Anpassungen übernommen werden können. (Vergleiche Abschnitt 2.1. Punkt 5.)

Das hier beschriebene Verfahren wurde zur Anwendung in kleinen und mittelgroßen Projekten entwickelt. Es beschränkt sich deshalb auf lokale Systeme und bietet „out of the Box“ keine Lösung für ein Monitoring von verteilten Systemen.

1.3. Voraussetzungen

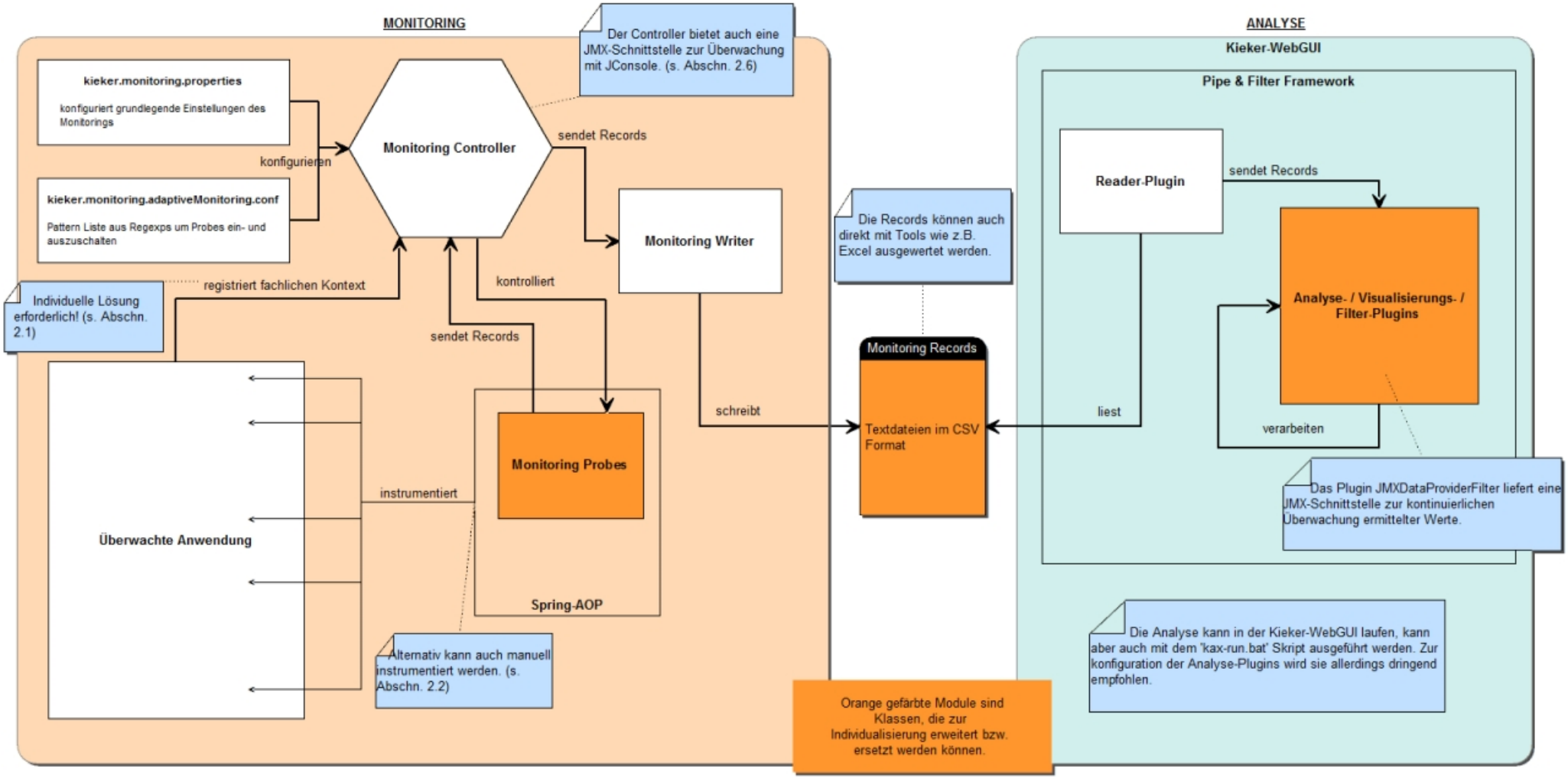
Um die in dieser Anleitung beschriebenen Verfahren einsetzen zu können, müssen die folgenden Voraussetzungen erfüllt sein:

- Java 1.7 oder höher
- Kieker 1.9 (wird, bei Verwendung von Maven, transitiv über die Datei `performance-monitoring-1.0.jar` eingebunden)
- Spring-AOP (ist keine Voraussetzung für die manuelle Instrumentierung)

Die Beispielklassen zur Bereitstellung des fachlichen Kontexts (`org.oasp.module.monitoring.ContextProvider` und `org.oasp.module.monitoring.ManualContextProvider`; s. Abschnitt 2.1. Punkt 5) verwenden außerdem:

- Spring 3.2.1
- javax.servlet 2.3

1.4. Übersichtsdiagramm


MONITORING
ANALYSE
Kieker-WebGUI
Pipe & Filter Framework
kieker.monitoring.properties
konfiguriert grundlegende Einstellungen des Monitorings
konfigurieren
kieker.monitoring.adaptiveMonitoring.conf
Pattern Liste aus Regexps um Probes ein- und auszuschalten
Monitoring Controller
Der Controller bietet auch eine JMX-Schnittstelle zur Überwachung mit JConsole. (s. Abschn. 2.6)
sendet Records
Monitoring Writer
Individuelle Lösung erforderlich! (s. Abschn. 2.1)
registriert fachlichen Kontext
kontrolliert
sendet Records
schreibt
Überwachte Anwendung
instrumentiert
Monitoring Probes
Spring-AOP
Alternativ kann auch manuell instrumentiert werden. (s. Abschn. 2.2)
Die Records können auch direkt mit Tools wie z.B. Excel ausgewertet werden.
Monitoring Records
Textdateien im CSV Format
liest
Reader-Plugin
sendet Records
Analyse- / Visualisierungs- / Filter-Plugins
verarbeiten
Das Plugin JMXDataProviderFilter liefert eine JMX-Schnittstelle zur kontinuierlichen Überwachung ermittelter Werte.
Die Analyse kann in der Kieker-WebGUI laufen, kann aber auch mit dem 'kax-run.bat' Skript ausgeführt werden. Zur konfiguration der Analyse-Plugins wird sie allerdings dringend empfohlen.
Orange gefärbte Module sind Klassen, die zur Individualisierung erweitert bzw. ersetzt werden können.

2. Monitoring

Das folgende Kapitel gibt in Abschnitt 2.1. eine Schritt-für-Schritt-Anleitung, wie ein grundlegendes Performance-Monitoring eingerichtet werden kann. In Abschnitt 2.2. wird eine alternative Möglichkeit der Codeinstrumentierung aufgezeigt. Die Abschnitte 2.3. - 2.6. befassen sich mit der Konfiguration sowie zusätzlichen Features des Monitorings.

2.1. Einrichten des Basic Performance-Monitorings

Im Folgenden wird in wenigen Schritten erklärt wie ein grundlegendes Performance-Monitoring erreicht werden kann. Dies ist ein guter Einstieg in das Monitoring mit Kieker und kann später individualisiert und erweitert werden.

1. Füge folgende Zeilen der `pom.xml` Deines Java-Projekts hinzu:

```
<dependency>
  <groupId>org.oasp.module.monitoring</groupId>
  <artifactId>performance-monitoring</artifactId>
  <version>1.0</version>
</dependency>
```

   Solltest Du kein Maven verwenden, musst Du das jar-File `performance-monitoring-1.0.jar` anderweitig dem Build-Path Deines Projekts hinzufügen.

2. Die Datei `resources\samples\kieker.monitoring.properties` (aus der `performance-monitoring-1.0.jar`) muss in das `META-INF` Verzeichnis Deines Projektes kopiert werden. In dieser Datei ist außerdem die Property `kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath` auf den Verzeichnispfad zu setzten, in dem die Monitoring-Logs angelegt werden sollen.
3. Um den Programmcode mittels Spring-AOP mit Monitoring-Probes zu instrumentieren, muss die Datei `resources\samples\performance-monitoring.xml` in alle Spring-„Application Contexts“ importiert werden, in denen das Monitoring verwendet werden soll. Im Fall einer Standardanwendung nach Vorgaben der OASP könnte das folgendermaßen aussehen:
   a. Kopieren der Datei `performance-monitoring.xml` in das Verzeichnis `<Dein_Projekt>\src\main\resources\resources\spring\crosscutting`.
   b. Importieren der Datei `performance-monitoring.xml` in beide vorhandenen „Application Contexts“ durch Einfügen folgender Zeile ...

```
<import resource="classpath:/resources/spring/crosscutting/performance-monitoring.xml" />
```

      ... in die Dateien `<Dein_Projekt>\src\main\resources\resources\spring\crosscutting.xml` und `<Dein_Projekt>\src\main\resources\resources\spring\gui-flows.xml`.

> **Tipp** Alternativ (oder ergänzend) kann der Programmcode auch manuell instrumentiert werden. Siehe dazu Abschnitt 2.2. Manuelle Instrumentierung.

4. Um zu bestimmen, welche Teile des Programmcodes grundsätzlich mit Monitoring-Probes versehen werden sollen, muss folgende Zeile aus der Datei `performance-monitoring.xml` angepasst werden:

```
<aop:advisor order="1" advice-ref="monitoringProbe"
pointcut="execution(public * com.capgemini.gastronomy.restaurant..*.*(..))" />
```

   Das Attribut `pointcut` bestimmt mit einem regulären Ausdruck, an welchen Methoden welcher Spring-Beans eine Monitoring-Probe gesetzt wird.

Wichtig Die Zeile `<aop:config proxy-target-class="true">` in dieser Datei sorgt dafür, dass CGLib-Proxies anstelle von JDK-Dynamic-Proxies verwendet werden. Dies ist wichtig, da JDK-Dynamic-Proxies nur für Interfaces erstellt werden können.

Anmerkung Zum genaueren Verständnis der regulären Ausdrücke, siehe Abschnitt 2.5. Pointcut-Expressions.

Anmerkung Für eine detailliertere Bestimmung, welche Monitoring-Probes ein- und welche ausgeschaltet werden sollen siehe Abschnitt 2.4. Aktivieren und Deaktivieren von Monitoring-Probes.

5. Als letzter Schritt muss nun noch ein Weg gefunden werden, die Daten des fachlichen Kontexts an einer Kontrollinstanz (`org.oasp.module.monitoring.CustomControlFlowRegistry`) zu registrieren. Dies muss in jedem Aufruf geschehen, bevor er die erste Monitoring-Probe durchläuft. Nur so ist gewährleistet, dass alle Probes immer auf den fachlichen Kontext des aktuellen Aufrufs zugreifen und die Messdaten später fachlich aggregiert und analysiert werden können. Zusätzlich muss auch am Ende jedes Aufrufs der fachliche Kontext wieder „abgemeldet“ werden, um eine eindeutige Zuordnung der Daten sicherzustellen.

   Eine Beispiellösung für dieses Problem liefert die Klasse `org.oasp.module.monitoring.ContextProvider`. Diese kann über einen zweiten AOP-Advisor in der Datei `performance-monitoring.xml` in den Code integriert werden:

```
<aop:advisor order="0" advice-ref="contextProvider"
pointcut="execution(public *
com.capgemini.gastronomy.restaurant.gui..*.*Controller.*(..))" />
```

   Auch hier ist der reguläre Ausdruck der Pointcut-Expression anzupassen.

   Wichtig Das Attribut `order=0` sorgt dafür, dass an einem Pointcut, an dem sowohl Monitoring-Probe als auch `ContextProvider` ausgeführt werden, letzterer zuerst durchlaufen wird. Dennoch musst Du mit der Pointcut-Expression selber dafür sorgen, dass er am Entrypoint jedes Aufrufs, der fachlich gemonitort werden soll, aufgerufen wird.

Das Basic Performance-Monitoring ist nun eingerichtet und produziert während der Ausführung des überwachten Programms Monitoring-Records. Hierzu erstellt Kieker `.dat`-Dateien und legt diese in das von Dir in der Property `kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath` angegebene Verzeichnis. In der `.dat`-Datei wird jeder Record im CSV(Comma Separated Values)-Format in einer Zeile dargestellt. Ein Beispiel könnte so aussehen (hier aus Platzgründen auf mehrere Zeilen verteilt)...

```
$1;1407402241982072647;public void
com.capgemini.gastronomy.restaurant.persistence.common.AbstractRestaurantDao.
delete(java.lang.Object);B263CD5C792468124DFB90166E35BD8A;3;1407402241974333720;
1407402241982061097;CE08039;5;4;deleteTable@1407402241879;65078309;44;true
```

... und ist folgendermaßen zu lesen:
Der erste Wert (`$0` oder `$1`) gibt die Klasse des Monitoring-Records an. Diese werden in der parallel angelegten Datei `kieker.map` festgehalten. Normalerweise steht `$1` für den `kieker.common.record.misc.KiekerMetadataRecord`. Dieser kann in der Kieker-API genauer betrachtet werden. `$0` steht für den `org.oasp.module.monitoring.CustomOperationExecutionRecord`; wie auch in diesem Beispiel.

Seine Werte können wie folgt interpretiert werden:

```
<RT>;<LT>;<MS>;<SID>;<TrID>;<Tin>;<Tout>;<HN>;<EOI>;<ESS>;<AID>;<UID>;<ThID>;<Suc>
```

Kürzel	Fachlicher Kontext	Beschreibung
<RT>	nein	*Record-Type* (Klassenname des Records in `kieker.map` wie oben beschrieben)
<LT>	nein	*Logging-Timestamp*
<MS>	nein	*Method-Signature* (Signatur der Methode an der die Monitoring-Probe angebracht ist)
<SID>	**ja**	*Session-ID*
<TrId>	nein	*Trace-ID* (eine Korrelations-ID, über die Records einem Aufruf zugeordnet werden können)
<Tin>	nein	*Time-IN* (Timestamp **vor** dem Methodeaufruf)
<Tout>	nein	*Time-OUT* (Timestamp **nach** dem Methodenaufruf)
<HN>	nein	*Host-Name*
<EOI>	nein	*Execution Order Index* (Index, der bei jeder Messsonde eines Aufrufs um eins erhöht wird)
<ESS>	nein	*Execution Stack Size* (gibt die Schachtelungstiefe des Aufrufs an)
<AID>	**ja**	*Action-ID* (Use-Case-Name oder ID der Use-Case-Instanz)
<UID>	**ja**	*User-ID*
<ThID>	nein	*Thread-ID*
<Suc>	nein	*Success* (Indikator, ob die Methode fehlerlos beendet wurde)

Achtung Die ersten beiden Daten (<RT> & <LT>) kommen nicht aus der `toArray()`-Methode des Records. So würde man mit `<record>.toArray()[0]` das Datum <MS> adressieren.

2.2. Manuelle Instrumentierung

Ist die Instrumentierung mit Spring-AOP nicht möglich oder unerwünscht, so können alternativ auch manuell Monitoring-Probes im Code integriert werden. Für diesen Zweck wurde die Klasse `org.oasp.module.monitoring.ManualOperationExecutionProbe` implementiert. Wie diese Klasse zu verwenden ist, zeigt das folgenden Beispiel:

Diese beispielhafte Methode soll überwacht werden:

```
1 public boolean createTable(long id) throws ValidationException {
2
3     this.tableManagement.createTable(id);
4     this.LOG.debug("Restaurant table with id '" + id + "' will be created.");
5     return true;
6 }
```

Hierzu wird zu Beginn der Methode die Methode `ManualOperationExecutionProbe.before(String)` aufgerufen, um den Monitoring-Vorgang einzuleiten. Als Parameter benötigt sie die Signatur der überwachten Methode. Der ursprüngliche Code der überwachten Methode sollte von einem `try`-Block umschlossen werden, damit auch im Falle einer Exception der Monitoring-Vorgang abgeschlossen werden kann. Dies geschieht mit Hilfe der Methode `ManualOperationExecutionProbe.after(MonitoringInfo)`, die als Parameter das Ergebnis der `ManualOperationExecutionProbe.before(String)`-Methode benötigt. Zusätzlich muss

ein Boolean übergeben werden, der angibt ob die Methode fehlerlos beendet wurde.

Eine manuelle Instrumentierung der Methode könnte also folgendermaßen aussehen:

```
1    String sigName = "public boolean createTable(long)";
2     try {
3       sigName = this.getClass().
4                 getDeclaredMethod("createTable", long.class).toString();
5     } catch (NoSuchMethodException | SecurityException e) {
6       this.LOG.warn("[MONITORING ISSUE] Method signature of class "
7                 + this.getClass().toString()
8                 + " could not be obtained. Using shortened verion: '"
9                 + sigName + "'");
10    }
11    MonitoringInfo mInfo = ManualOperationExecutionProbe.before(sigName);
12    boolean success = false;
13    try {
14      this.tableManagement.createTable(id);
15      this.LOG.debug("Restaurant table with id '" + id
16                + "' will be created.");
17      success = true;
18    } finally {
19      ManualOperationExecutionProbe.after(mInfo, success);
20    }
21    return true;
22  }
```

Achtung

Den String-Parameter für die Methode `ManualOperationExecutionProbe.before(String)` musst Du selber besorgen. Er muss einem String entsprechen, wie ihn die Methode `java.lang.reflect.Method.toString()` produzieren würde. D.h. er muss folgende Syntax berücksichtigen:

> <Zugriffsmodifikator> <Rückgabetyp> <vollqualifizierte MethodenName> (<ParameterTypA>, <ParameterTypB, ...) [throws <throwable>]

Andernfalls funktioniert das Ein- und Ausschalten von Monitoring-Probes nicht. Die Zeilen 1 - 7 zeigen eine beispielhafte Möglichkeit, diesen String zusammenzustellen.

Tipp

Die manuelle Instrumentierung kann auch ergänzend zur Instrumentierung mit Spring-AOP genutzt werden, um z.B. Klassen, die keine Spring-Beans sind, zu monitoren.

2.3. Konfiguration des Monitorings

Zur Konfiguration des Monitorings wird die Datei `kieker.monitoring.properties` verwendet, die per Default im `META-INF` Verzeichnis des Projekts liegen sollte. Obwohl sie ausführlich dokumentiert ist, soll hier kurz auf die wichtigsten Properties eingegangen werden:

- `kieker.monitoring.enabled` — Sollte normalerweise `true` sein. Andernfalls wird das Monitoring direkt nach dem Start der Anwendung abgeschaltet.
- `kieker.monitoring.timer` — Sollte `kieker.monitoring.timer.SystemNanoTimer` sein, da der `SystemMilliTimer` in den meisten Fällen nicht präzise genug ist.
- `kieker.monitoring.timer.SystemNanoTimer.unit` — Sollte immer `0` sein, da einige Analyse-Plugins nur mit Nanosekunden arbeiten können.
- `kieker.monitoring.writer` — Sollte `kieker.monitoring.writer.filesystem.AsyncFsWriter` sein, da die Analyse auf diesen Writer abgestimmt ist.

- `kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize` — Produziert das Monitoring zeitweise zu viele Monitoring-Records, kann über diese Property die Größe des Buffers angepasst werden.
- `kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior` — Normalerweise `0`. Sollte der interne Buffer aufgrund von zu vielen Records überlaufen, kann diese Property auf `2` gesetzt werde. Dann werden allerdings die überzähligen Records verworfen. Diese Property sollte nicht auf `1` gesetzt werden, da der Writer sonst nicht mehr asynchron arbeitet und die Anwendung erheblich verlangsamen kann.

Anmerkung

Bei Datei `resources\samples\kieker.monitoring.properties` handelt es sich um eine gekürzte Form des von Kieker mitgelieferten Originals. Solltest Du Dich dazu entscheiden, die Analyse selber in die Hand zu nehmen und den `kieker.monitoring.writer.filesystem.AsyncFsWriter` durch einen anderen zu ersetzen, empfehle ich das Original heranzuziehen, das in jedem Release von Kieker enthalten ist.

2.4. Aktivieren und Deaktivieren von Monitoring-Probes

Um Monitoring-Probes zur Laufzeit zu aktivieren oder zu deaktivieren, kann eine Pattern-List in der Datei `kieker.monitoring.adaptiveMonitoring.conf` verwendet werden. Hierzu müssen folgende Properties in der Datei `kieker.monitoring.properties` gesetzt werden:

```
kieker.monitoring.adaptiveMonitoring.enabled=true
...
kieker.monitoring.adaptiveMonitoring.configFile=<Pfad zur Datei>
...
kieker.monitoring.adaptiveMonitoring.readInterval=30
```

Die erste Property schaltet das adaptive Monitoring ein. In der Zweiten ist der Pfad zur Datei `kieker.monitoring.adaptiveMonitoring.conf` anzugeben. Mit der Dritten kann das Poll-Intervall in Sekunden festgelegt werden.

Die Syntax der Pattern-List ist in der Datei `kieker.monitoring.adaptiveMonitoring.conf` mit einigen Beispielen erklärt.

Alternativ können Monitoring-Probes auch über den Zugriff auf den Monitoring-Controller via JMX ein- und ausgeschaltet werden.

2.5. Pointcut-Expressions

In der Datei `performance-monitoring.xml` wird mittels Pointcut-Expressions bestimmt, welche Methoden welcher Spring-Beans durch das Monitoring überwacht werden. Diese Pointcut-Expressions verwenden die AspectJ-Pointcut-Expression-Language, die im AspectJ Programmers Guide näher beschrieben ist.

2.6. Zugriff auf den Monitoring-Controller via JMX

Auf den Monitoring-Controller von Kieker kann als MBean über JMX zugegriffen werden. Auf diese Weise kann beispielsweise zur Laufzeit das gesamte Monitoring oder einzelne Probes ein- und ausgeschaltet werden. Hierzu müssen die folgenden Properties in der Datei `kieker.monitoring.properties` gesetzt werden:

```
kieker.monitoring.jmx=true
...
kieker.monitoring.jmx.MonitoringController=true
```

Nun kann z.B. JConsole verwendet werden, um auf den Monitoring-Controller zuzugreifen. Dabei ist zu beachten, dass Kieker zum Classpath von JConsole hinzugefügt wird. Eine detailliertere Erläuterung findest Du im Kieker-User-Guide.

2.7. Individuelle Anpassung des Monitorings

Sollten Dir die vom Monitoring festgehaltenen Daten nicht umfangreich genug sein, so kannst Du es um zusätzliche Daten erweitern. Hierzu sollten zunächst die Klassen `org.oasp.module.monitoring.CustomOperationExecutionRecord` und `org.oasp.module.monitoring.CustomOperationExecutionMethodInvocationInterceptor` (bzw. `org.oasp.module.monitoring.ManualOperationExecutionProbe` im Falle von manueller Instrumentierung) betrachtet und erweitert werden.

Sollte es sich bei den neuen Daten um fachlichen Kontext handeln, so muss außerdem die Klasse `org.oasp.module.monitoring.CustomControlFlowRegistry` erweitert werden und auch bei der Registrierung des fachlichen Kontexts an der Instanz dieser Klasse müssen die neuen Daten berücksichtigt werden (s. Abschn. 2.1. Punkt 5).

> **Anmerkung** Einige der genannten Klassen lassen sich nicht oder nur sehr umständlich im Sinne von "`extend`" erweitern. In diesen Fällen muss leider auf das Prinzip des Kopierens und Ergänzens zurückgegriffen werden.

3. Analyse

Die vom Monitoring produzierten Monitoring-Records im CSV Format können mit Tools wie z.B. Excel ausgewertet werden. Diese Anleitung beschreibt in Abschnitt 3.1. - 3.3. zusätzlich, wie eine grundlegende Analyse mit Hilfe des plugin-basierten „Pipe-and-Filter Frameworks“ und der WebGUI von Kieker durchgeführt werden kann. Hierzu werden eine Reihe von fertigen Plugins bereitgestellt, die in Abschnitt 3.4 dokumentiert sind. In Abschnitt 3.5. wird außerdem beschrieben, wie das Framework um selbstgeschriebene Plugins schnell erweitert werden kann, um individuelle Lösungen zu ermöglichen. Abschnitt 3.6. erläutert wie eine Analyse ohne die WebGUI durchgeführt werden kann.

3.1. Das Pipe-and-Filter Framework

Das Package `Kieker.Analysis` stellt ein plugin-basiertes Framework zur Verfügung, das eine individuell konfigurier- und erweiterbare Analyse ermöglicht. Hierbei speist immer ein Reader-Plugin — das Monitoring-Records in irgendeiner Form einliest — Records in ein Netzwerk aus mehreren Filter-Plugins, die die Records dann weiterverarbeiten, aggregieren, analysieren, visualisieren oder auch für externe Schnittstellen bereitstellen. Allen Filter-Plugins ist gemein, dass sie immer über mindestens einen Input-Port verfügen, über den sie Records oder andere Objekte, die bei der Weiterverarbeitung der Records entstehen, empfangen können. Die meisten verfügen außerdem über mindestens einen Output-Port, der zur Weiterleitung von Daten an andere Plugins dient.

Für eine detailliertere Beschreibung ziehe bitte den Kieker-User-Guide zu Rate.

3.2. Die Kieker-WebGUI

Das Erstellen, Konfigurieren und Ausführen der Plugin-Netzwerke kann zwar unter Verwendung einer Java-API erfolgen, doch ist dies mühsam und fehleranfällig. Deshalb soll in diesem Guide die Kieker-WebGUI zur Analyse verwendet werden, die zu eben diesem Zweck von Kieker entwickelt wird. Zum derzeitigen Stand (Kieker-1.9) befindet sich die WebGUI noch in der Beta-Phase, doch sind die wichtigen Funktionalitäten schon ausreichend implementiert, um brauchbare Ergebnisse zu erzielen und mit einer ausführlichen Weiterentwicklung ist zu rechnen.

Im Kieker Blog kann ein ausführlicherer Überblick über die WebGUI gewonnen werden.

3.3. Erstellung eines beispielhaften Plugin-Netzwerks

Im folgenden wird unter Verwendung der Kieker-WebGUI ein Beispielhaftes Plugin-Netzwerk erstellt, um die Verwendung der WebGUI zu verdeutlichen.

1. Entpacke das `kieker-webgui-1.9_binaries.zip`-File in ein beliebiges Verzeichnis.
2. Führe das Skript `kieker-webgui-1.9_binaries\bin\Kieker.WebGUI.bat` aus, um die WebGUI zu starten. (Dies kann einige Sekunden dauern.)
3. Gib folgende URL http://localhost:8080/Kieker.WebGUI/login in Deinen Webbrowser ein.
4. Logge Dich mit dem Benutzernamen „admin“ und dem Passwort „kieker“ ein.

5. Klicke oben links auf „Datei“ → „Neues Projekt“, um ein Projekt mit beliebigem Namen zu erstellen; im Folgenden als `<Dein_Projekt>` bezeichnet.
   Es wurde nun ein Verzeichnis `kieker-webgui-1.9_binaries\bin\data\<Dein_Projekt>` erstellt.
6. Klicke nun unter http://localhost:8080/Kieker.WebGUI/pages/ auf <Dein_Projekt> → „Analyse Editor“. Nun siehst Du die Seite, auf der Plugin-Netzwerke zusammengestellt werden können.
7. Importiere jetzt die Datei `performance-monitoring-1.0.jar` in Dein Projekt. Hierzu klickst Du oben links auf „Datei“ → „Bibliotheken Verwalten“ und im daraufhin erscheinenden Popup auf „Durchsuchen“. Wähle jetzt die eben genannte Datei aus. Damit ist gewährleistet, dass Dein Projekt über die eigens für diese Anleitung geschriebenen Klassen verfügt.

> **Tipp** Alternativ können .jar-Datein, die Klassen zur Analyse enthalten, auch in das Verzeichnis `kieker-webgui-1.9_binaries\bin\data\<Dein_Projekt>\lib` kopiert werden.

8. Durch klicken auf Plugins im Bereich „Verfügbare Plugins“ auf der Linken Seite, fügst Du sie Deinem Plugin-Netzwerk hinzu. Klicke zunächst auf `FSReader`. Dieser erscheint nun als graues Rechteck im Haupt-Widget und lässt sich durch Ziehen mit der Maus positionieren.

> **Anmerkung** Über das Kästchen mit der Aufschrift „Globale Konfiguration“ lässt sich die Zeiteinheit der Monitoring-Records angeben; in fast allen Fällen kann es beim Default `NANOSECONDS` belassen werden.

9. Klicke auf den `FSReader` im Haupt-Widget, um im unteren Bereich seine Properties einzublenden. Hier kann jedes Plugin konfiguriert werden. Gib für die Property `inputDirs` des `FSReader` den Pfad zu dem Verzeichnis an, in dem sich die zu analysierenden Monitoring-Records befinden.

> **Wichtig** Es muss das Verzeichnis sein, in dem sich die `.map` und `.dat` Dateien befinden. Nicht etwa ein Parent-Verzeichnis!

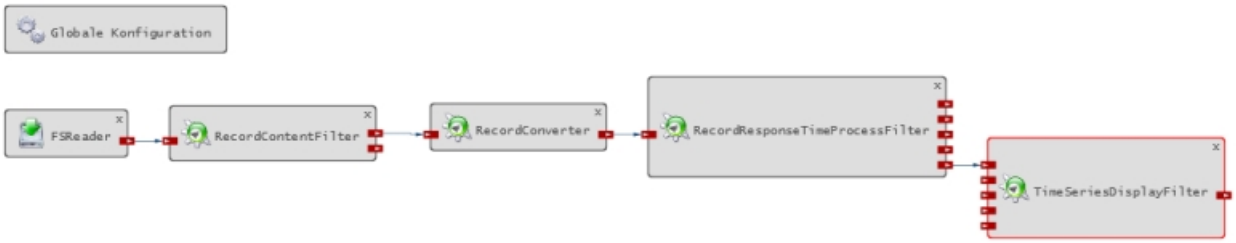
10. Füge nun einen `RecordContentFilter` Deinem Plugin-Netzwerk hinzu. Dieser filtert `OperationExecutionRecords` anhand ihres Inhalts; z.B. der Methodensignatur.
11. Verbinde durch Anklicken den Output-Port des `FSReader` mit dem Input-Port des `RecordContentFilter`.
12. Setze nun die Properties des `RecordContentFilter` wie folgt:
    - type: `STRING`
    - compare: `GE` (dies entspricht für Strings einem *contains*)
    - arrayIndex: `0` (am Index 0 befindet sich in der Array-Darstellung des Records normalerweise die Methodensignatur)
    - filterKey: <Ein Teil der Methodensignatur, die alle Records enthalten sollen, die Du herausfiltern möchtest. Z.B. `com.capgemini.gastronomy.restaurant.`>

    > **Tipp** Die Symbolleiste des Analyse Editors verfügt über einen Button zur automatischen Anordnung der Plugins. („Layoutet den Graphen“)

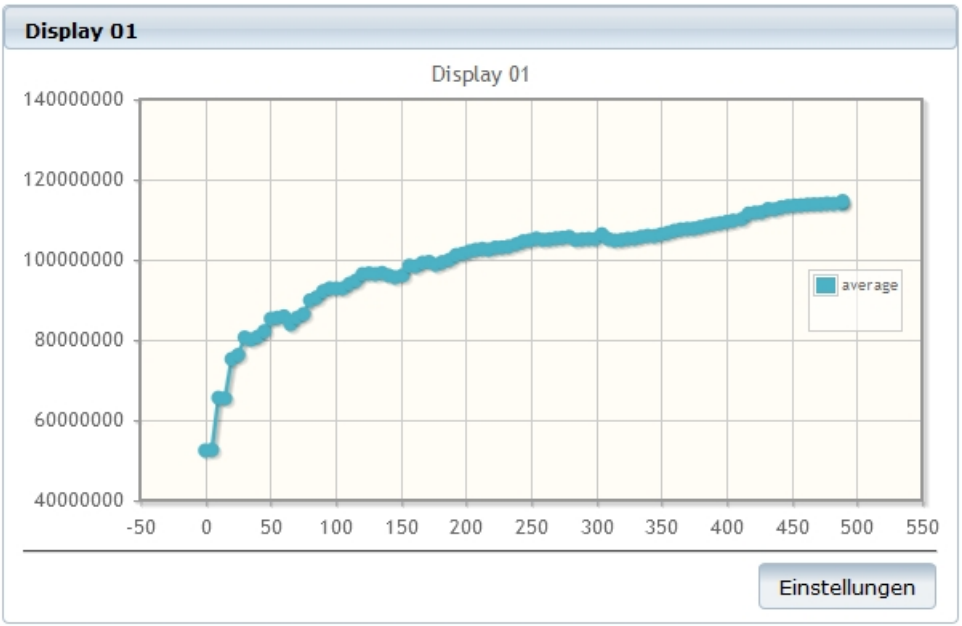
13. Füge als nächstes einen `RecordConverter` hinzu und verbinde seinen Input-Port mit dem Output-Port `recordsOutTrue` des `RecordContentFilter`. Dieser wandelt die Records in Objekte der Klasse `kieker.tools.opad.record.NamedDoubleRecord` um — eine abstraktere Klasse, die nur noch Timestamp und Responsetime des Records enthält. Hierzu müssen keine Properties gesetzt werden.
14. Als nächstes muss dem Netzwerk ein `RecordResponseTimeProcessFilter` hinzugefügt werden. Verbinde seinen Input-Port mit dem Output-Port des `RecordConverter` und belasse seine Properties wie voreingestellt. Der `RecordResponseTimeProcessFilter` aggregiert nun die ankommenden `NamedDoubleRecords` und berechnet aus diesen Anzahl, Minimum, Maximum, Median und Durchschnitt der Antwortzeiten. Die Ergebnisse sendet er alle fünf Sekunde (Property

„deliverPeriodInMS“ = `5000`) über die fünf Output-Ports weiter.

15. Die vom `RecordResponseTimeProcessFilter` berechneten Werte lassen sich mit einem `TimeSeriesDisplayFilter` in der WebGUI visualisieren. Füge einen solchen Deinem Plugin-Netzwerk hinzu und verbinde den Output-Port `outputAvg` des `RecordResponseTimeProcessFilter` (dieser überträgt den Durchschnittswert) mit einem der Input-Ports des `TimeSeriesDisplayFilter`. Setze nun die Property des entsprechenden Input-Ports des `TimeSeriesDisplayFilter` auf „average“, um zu benennen **was** diese Serie darstellt. D.h. wenn Du die Verbindungslinie z.B. zum Input-Port `inputSeriesA` gezogen hast, solltest Du die für die Property `seriesA` „average“ eingeben.
16. Dein erstes Plugin-Netzwerk ist nun fertig und betriebsbereit. Klicke auf das Diskettensymbol oben links, um Dein Projekt zu speichern. Die folgende Grafik zeigt, wie das ganze in etwa aussehen sollte:



17. Um eine Visualisierung der `TimeSeriesDisplayFilter` zu ermöglichen, muss diese noch im „Cockpit Editor“ eingerichtet werden (Button oben rechts).
18. Der Cockpit Editor dient der Konfiguration der Visualisierung, wie sie später im Cockpit-Bereich zu sehen ist. In unserem Fall verfügt nur der `TimeSeriesDisplayFilter` über eine Visualisierung. Klicke zunächst oben links auf „Datei“ → „Neue Ansicht“, um eine neue Ansicht zu erstellen. Klicke dann links auf den Namen der gerade erstellten Ansicht und auf „Ansicht Selektieren“, um sie auszuwählen.
19. Füge nun das „XYPlot Display“ des `TimeSeriesDisplayFilter` der Ansicht hinzu, indem Du rechts auf `TimeSeriesDisplayFilter` → „XYPlot Display“ klickst. Optional kannst Du dem Display noch einen Namen geben, durch setzen der Property „Name“ im unteren Bereich.
20. Speichere Dein Projekt erneut durch Anklicken des Diskettensymbols.
21. Wechsle zur Analyse-Ansicht mit dem Button „Analyse“ oben rechts.
22. Klicke unten auf „Analyse Instanziieren“ gefolgt von „Analyse Starten“, um die Analyse zu starten.
23. Wechsle nun in die „Cockpit“-Ansicht und wähle die von Dir erstellte Ansicht aus, um das Ergebnis Deiner Analyse zu betrachten. Je nach verwendeten Monitoring-Daten sollte Deine Visualisierung der folgenden Grafik mehr oder weniger ähneln:



Anmerkung Du solltest die Analyse in der Analyse-Ansicht immer zurücksetzen, bevor Du eine neue konfigurierst oder startest.

3.4. Übersicht der bereitgestellten Plugin-Klassen

Im folgenden findest Du eine Übersicht der wichtigsten Filter-Plugins, die für die Analyse mit dem „Pipe-and-Filter Framework“ empfohlen werden. Mit Hilfe dieser Plugins ist eine grundlegende Analyse bereits möglich. So können Records, die von der Klasse `kieker.common.record.controlflow.OperationExecutionRecord` erben, mit dem **RecordContentFilter** nach jeglichem Inhalt gefiltert werden. Mit dem **RecordResponseTimeProcessFilter** lassen sich die gefilterten Daten dann aggregieren und zu Kennzahlen zusammenfassen. Diese Kennzahlen können dann mit dem **TimeSeriesDisplayFilter** in der WebGUI visualisiert und/oder mit dem **JMXDataProviderFilter** über JMX bereitgestellt werden, um mit externen Monitoring-Tools darauf zuzugreifen.

Kieker stellt noch eine Reihe weiterer Plugins — die in der WebGUI auch aufgelistet sind — zur Verfügung. Diese können in der Kieker-API nachgeschlagen werden, sind aber zum derzeitigen Stand (Kieker-1.9) nur spärlich dokumentiert.

FSReader (`kieker.analysis.plugin.reader.filesystem.FSReader`)

Dies ist der von Kieker bereitgestellte Standardreader für die offline Analyse. Er sollte verwendet werden, wenn während der Analyse keinen neuen Monitoring-Records zu erwarten sind; diese würde er einfach ignorieren. Er wandelt die in der .dat-Datei gefundenen Records wieder in Java-Objekte um und sendet sie weiter.

Input-Ports

keine — Das Verzeichnis zum Lesen wird in der Property `inputDirs` angegeben.

Output-Ports

`monitoringRecords`

Die gelesenen Records werden über diesen Output-Port weitergeleitet.

Properties

`inputDirs`

Der Pfad zum Verzeichnis, das die .dat-Datei mit den Monitoring-Records im CSV-Form enthält.

`ignoreUnknownRecordTypes`

Ist diese Property auf `false` gesetzt, so bricht die Analyse ab, wenn ein Record von unbekanntem Typ gelesen wird.
Default: `false`

RecordConverter (`org.oasp.module.monitoring.analysis.RecordConverter`)

Dieses Plugin wandelt eingehende Objekte vom Typ `kieker.common.record.controlflow.`**`OperationExecutionRecord`** (oder abgeleitete Klassen) in Objekte vom Typ `kieker.tools.opad.record.`**`NamedDoubleRecord`** um. Diese enthalten nur noch Timestamp und Responsetime und können vom **RecordResponseTimeProcessFilter** verarbeitet werden.

Input-Ports

`oer`
: Empfängt Objekte vom Typ `kieker.common.record.controlflow.`**`OperationExecutionRecord`**.

Output-Ports

`ndr`
: Leitet Objekte vom Typ `kieker.tools.opad.record.`**`NamedDoubleRecord`** weiter.

Properties

keine

Tipp Die folgenden Filter-Plugins sind eigens für diese Anleitung geschrieben worden und sind zusätzlich ausführlich im JavaDoc der Datei `performance-monitoring-1.0.jar` dokumentiert.

OnlineFSReader (`org.oasp.module.monitoring.analysis.OnlineFSReader`)

Dieser Reader ist eine Erweiterung des **FSReader** und ist dafür konzipiert, eine kontinuierliche Analyse während des Monitorings zu ermöglichen. Findet er keine weiteren Record-Einträge in einer .dat-Datei, so wartet er auf neue und überprüft auch das Verzeichnis auf neue .dat-Dateien, um diese ggf. einzulesen.

Input-Ports

keine — Das Verzeichnis zum Lesen wird in der Property `inputDirs` angegeben.

Output-Ports

`monitoringRecords`
: Die gelesenen Records werden über diesen Output-Port weitergeleitet.

Properties

`inputDirs`
: Der Pfad zum Verzeichnis, das die .dat-Datei mit den Monitoring-Records im CSV-Form enthält.

`ignoreUnknownRecordTypes`
: Ist diese Property auf `false` gesetzt, so bricht die Analyse ab, wenn ein Record von unbekanntem Typ gelesen wird.
Default: `false`

RecordContentFilter (`org.oasp.module.monitoring.analysis.RecordContentFilter`)

Dieses Plugin dient der Filterung von `kieker.common.record.controlflow.`**`OperationExecutionRecords`** anhand ihres Inhalts. Hierzu werden diese in der Array-Darstellung betrachtet, wie sie die `toArray()`-Methode produziert.

Input-Ports

`recordsIn`
: Empfängt Objekte vom Typ `kieker.common.record.controlflow.`**`OperationExecutionRecord`**.

Output-Ports

`recordsOutTrue`
: Leitet jene Objekte weiter, für die die Filterbedingung zutrifft. (Die Vergleichsoperation ergibt `true`.)

`recordsOutFalse`
: Leitet jene Objekte weiter, für die die Filterbedingung nicht zutrifft. (Die Vergleichsoperation ergibt `false`.)

Properties

`type`
: Datentyp der Daten anhand derer gefiltert werden soll. Nur `NUMBER`, `BOOLEAN` und `STRING` sind gültig.
Default: `STRING`

`compare`
: Einer der sechs Vergleichsoperatoren. `GT` (>), `LT` (<), `GE` (>=), `LE` (<=), `EQ` (==) oder `NE` (!=).
Für `type` = `NUMBER` sind alle Operatoren gültig.
Für `type` = `BOOLEAN` sind nur `EQ` und `NE` gültig.
Für `type` = `STRING` sind die Operatoren `GT` und `LE` ungültig; `EQ` bedeutet, dass der String des Records und der `filterKey` exakt gleich sein müssen; `NE` bedeutet, dass der String des Records und der `filterKey` nicht exakt gleich sein dürfen; `GE` bedeutet, dass der String des Records den `filterKey` enthalten muss; `LT` bedeutet, dass der String des Records den `filterKey` nicht enthalten darf.
Default: `EQ`

`arrayIndex`
: Der Index, der angibt, an welcher Stelle der Array-Darstellung des Records das Datum liegt, das beim Filtern verglichen wird.
Default: `0`

`filterKey`
: Der Schlüssel, mit dem das gewählte Datum des Records verglichen wird. Er wird als Datentyp `type` interpretiert.
Default: In Abhängigkeit von `type` — `0.0`, `true` oder ein leerer String.

RecordResponseTimeProcessFilter

(`org.oasp.module.monitoring.analysis.RecordResponseTimeProcessFilter`)

Dieser Filter aggregiert die eingehenden `kieker.tools.opad.record.`**`NamedDoubleRecord`**-Objekte über ein konfigurierbares, fließendes Zeitfenster. Aus ihren Responsetimes ermittelt er in wiederum konfigurierbaren Zeitabständen Anzahl, Minimum, Maximum, Median und Durchschnitt und sendet diese als `kieker.tools.opad.record.`**`NamedDoubleTimeSeriesPoint`**-Objekte über fünf verschieden Ports weiter.

Input-Ports

`inputRecords`
: Empfängt Objekte vom Typ `kieker.tools.opad.record.`**`NamedDoubleRecord`**.

Output-Ports

`outputCount`
: Leitet alle <`deliverPeriodInMS`> Millisekunden die Anzahl der betrachteten Records als `kieker.tools.opad.record.`**`NamedDoubleTimeSeriesPoint`** weiter.

`outputMin`
: Leitet alle <`deliverPeriodInMS`> Millisekunden die kürzeste Responsetime der betrachteten Records als `kieker.tools.opad.record.`**`NamedDoubleTimeSeriesPoint`** weiter.

`outputMax`
: Leitet alle <`deliverPeriodInMS`> Millisekunden die längste Responsetime der betrachteten Records als

kieker.tools.opad.record.**NamedDoubleTimeSeriesPoint** weiter.

`outputMedian`
: Leitet alle <`deliverPeriodInMS`> Millisekunden den Median der Responsetimes der betrachteten Records als kieker.tools.opad.record.**NamedDoubleTimeSeriesPoint** weiter.

`outputAvg`
: Leitet alle <`deliverPeriodInMS`> Millisekunden die durchschnittliche Responsetime der betrachteten Records als kieker.tools.opad.record.**NamedDoubleTimeSeriesPoint** weiter.

Properties

`timeWindowInS`
: Bestimmt das fließende Zeitfenster (in Sekunden), in dem eingegangene Records aggregiert werden. Das Zeitfenster reicht vom Timestamp des neuesten Records <`timeWindowInS`> Sekunden in die Vergangenheit. Ältere Records werden verworfen.
Default: 3600

`maxRecords`
: Anzahl der Records die maximal betrachtet werden. Liegen mehr als <`maxRecords`> Records im Zeitfenster, so werden die ältesten verworfen.
Default: 100000

`deliverPeriodInMS`
: Frequenz (in Millisekunden) in der die Berechnungen stattfinden und die berechneten Daten versandt werden.
Default: 5000

TimeSeriesDisplayFilter
(`org.oasp.module.monitoring.analysis.TimeSeriesDisplayFilter`)

Dieses Plugin dient der visuellen Darstellung von kieker.tools.opad.record.**NamedDoubleTimeSeriesPoint**-Objekten als Liniendiagramm. Hierzu können bis zu fünf verschiedene Serien/Linien in einem Diagramm dargestellt werden.

Input-Ports

`inputSeriesA`
: Der Port für `NamedDoubleTimeSeriesPoints` die in der ersten Serie dargestellt werden sollen.

`inputSeriesB`
: Der Port für `NamedDoubleTimeSeriesPoints` die in der zweiten Serie dargestellt werden sollen.

`inputSeriesC`
: Der Port für `NamedDoubleTimeSeriesPoints` die in der dritten Serie dargestellt werden sollen.

`inputSeriesD`
: Der Port für `NamedDoubleTimeSeriesPoints` die in der vierten Serie dargestellt werden sollen.

`inputSeriesE`
: Der Port für `NamedDoubleTimeSeriesPoints` die in der fünften Serie dargestellt werden sollen.

Output-Ports

`relayedPoints`
: Alle eingehenden `NamedDoubleTimeSeriesPoints` werden über diesen Port weitergeleitet.

Properties

`seriesA`
: Der im Diagramm dargestellte Name für die erste Serie.

Default: series-A

`seriesB`
Der im Diagramm dargestellte Name für die zweite Serie.
Default: series-B

`seriesC`
Der im Diagramm dargestellte Name für die dritte Serie.
Default: series-C

`seriesD`
Der im Diagramm dargestellte Name für die vierte Serie.
Default: series-D

`seriesE`
Der im Diagramm dargestellte Name für die fünfte Serie.
Default: series-E

`maxEntries`
Die maximale Anzahl von Punkten die pro Serie dargestellt werden.
Default: 1000

`timestampDisplayUnit`
Zeiteinheit, die auf der X-Achse dargestellt werden soll.
Default: SECONDS

JMXDataProviderFilter (`org.oasp.module.monitoring.analysis.JMXDataProviderFilter`)
Dieses Plugin ist als MBean implementiert und dient der Bereitstellung einzelner Double-Werte für die JMX-Schnittstelle. Hierzu liest es den `value` eingehender `kieker.tools.opad.record.NamedDoubleTimeSeriesPoint`-Objekte aus.

Input-Ports

`inputData`
Empfängt Objekte vom Typ `kieker.tools.opad.record.NamedDoubleTimeSeriesPoint`.

Output-Ports

`relayedData`
Leitet die eingehenden `NamedDoubleTimeSeriesPoints` unverändert weiter .

Properties

`valueName`
Bezeichnung des ausgelesenen Wertes, der über JMX bereitgestellt wird.
Default: `<unknown-data>`

`domain`
Domain-Name der MBean.
Default: `org.oasp.module.monitoring.analysis`

3.5. Individuelle Lösungen

Wenn die in Abschnitt 3.4. aufgeführten Plugins Deinen Ansprüchen nicht genügen, so muss eine individuelle Lösung gefunden werden. Hierzu kann zunächst ein Blick auf die von Kieker bereitgestellten Plugin-Klassen geworfen werden. Diese sind in der Kieker-API einzusehen, doch ist die Dokumentation beim derzeitigen Stand (Kieker-1.9) noch nicht vollständig und u.U. kann eine selbst implementierte Klasse schneller zu einer guten Lösung führen.

Solltest Du Dich dafür entscheiden eigene Plugins zu schreiben, so kannst Du Dich an den Klassen im Package `org.oasp.module.monitoring.analysis` orientierten, die ausführlich im Javadoc dokumentiert sind. Außerdem ist bei der Implementierung von Plugins folgendes zu beachten:

- alle Plugin-Klassen müssen mit der Annotation `@Plugin` gekennzeichnet sein, in der Output-Ports und Properties definiert werden

- alle Reader-Plugins müssen von der Klasse `kieker.analysis.plugin.reader.AbstractReaderPlugin` abgeleitet sein
- alle Filter-Plugins müssen von der Klasse `kieker.analysis.plugin.filter.AbstractFilterPlugin` abgeleitet sein
- alle Filter-Plugins müssen mindestens eine Input-Methode haben, die mit der Annotation `@InputPort` gekennzeichnet ist
- alle Properties eines Plugins sollten immer über ein Objekt der Klasse `kieker.common.configuration.Configuration` gesetzt werden
- alle Properties eines Plugins sollten immer mit der Methode `getCurrentConfiguration()` in einem Objekt der Klasse `kieker.common.configuration.Configuration` zurückgegeben werden

Eine ausführliche Anleitung zum Schreiben von Reader- und Filter-Plugins findet sich im Kapitel „4.2 Developing Analysis Plugins and Repositories“ im Kieker-User-Guide.

3.6. Stand-Alone-Analyse mit dem „kax-run“-Skript

Die in der Kieker-WebGUI erstellten Plugin-Netzwerke werden in `.kax`-Dateien gespeichert (abgelegt im Verzeichnis `kieker-webgui-1.9_binaries\bin\data\<Dein_Projekt>`). Es ist möglich diese ohne die WebGUI zu starten. Dies macht natürlich nur Sinn, wenn die Ausgabe nicht auf Visualisierung innerhalb der WebGUI basiert. So z.B. beim `org.oasp.module.monitoring.analysis.JMXDataProviderFilter`. Hierzu wird allerdings ein Skript aus dem Kieker-Release benötigt. Einmal heruntergeladen und entpackt, findest du es unter `kieker-1.9\bin\kax-run.bat`. Nun kannst Du mit folgendem Befehl dein `.kax`-File starten:

```
kax-run.bat -i <Pfad zu Deinem .kax-File>
```

Wichtig Die Datei `performance-monitoring-1.0.jar` sowie alle `.jar`-Dateien, die etwaige verwendete Klassen enthalten, müssen dem Verzeichnis `kieker-1.9\lib` hinzugefügt werden.

4. Nützliche Links und Literaturverweise

- Kieker User Guide — Kieker Project (Apr. 2014). Kieker 1.9 User Guide. Software Engineering Group, Kiel University, Kiel, Germany.
- Kieker-Online-API
- Kieker Blog — gibt einen Überblick über die Kieker-WebGUI.
- AspectJ-Programmers-Guide — erklärt die Pointcut-Expression-Language, derer Spring-AOP sich bedient.
- Stefan Eberlein. „*Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance-Monitoring*“. Dimplomarbeit. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2011 — Bietet einen umfangreichen Überblick über das Thema.
- Johannes Mensing „*Methodische Vorgaben für Performance-Monitoring im Kontext betrieblicher Java-Anwendungen*“. Bachelorarbeit. Hochschule RheinMain Wiesbaden, 2014 — Im Rahmen der Bachelorarbeit entstand diese Anleitung.

Literatur

[1] Ralf Däschlein, Jochen Englert, Thilo Hermann, Michael Friedrich, Christian Harms, Andreas Hess, Bernhard Humm, Marc Jäckle, Alexander Ramisch, Joachim Wenzel, Arno Weiß, Thomas Wolf und Marcus Zander. *Architekturleitfaden Anwendungskonstruktion*. 1.0.0. Capgemini sd&m AG. Carl-Wery-Str. 42 81739 München, 2010 (siehe S. 2, 16, 29).

[2] Thomas Maier. *Einfaches Performance-Monitoring von Java-Anwendungen*. Capgemini-interner Vortrag. 2013 (siehe S. 3).

[3] Stefan Eberlein. „Erhebung und Analyse von Kennzahlen aus dem fachlichen Performance-Monitoring“. Diplomarbeit. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2011 (siehe S. 3, 5, 7, 8, 10, 13, 29).

[4] Wikipedia. *Monitoring*. Webseite: `http://de.wikipedia.org/wiki/Monitoring` (letzter Zugriff August 2014) (siehe S. 5).

[5] Wikipedia. *Aspektorientierte Programmierung*. Webseite: `http://de.wikipedia.org/wiki/Aspektorientierte_Programmierung` (letzter Zugriff August 2014) (siehe S. 7).

[6] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey und Dennis Kieselhorst. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Forschungsbericht. Kiel University, Nov. 2009 (siehe S. 7, 10).

[7] Johannes Weigend, Johannes Siedersleben und Josef Adersberger. „Dynamische Analyse mit dem Software-EKG“. In: *Informatik-Spektrum* 34.5 (Okt. 2011), S. 484–495 (siehe S. 8).

[8] André van Hoorn, Jan Waller und Wilhelm Hasselbring. „Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis“. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Webseite: `http://kieker-monitoring.net/` (letzter Zugriff August 2014). ACM, Apr. 2012, S. 247–248 (siehe S. 10).

[9] Kieker Project. *Kieker User Guide*. Forschungsbericht. Apr. 2014 (siehe S. 11, 22, 28).

[10] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad und Arjen Poutsma. *The Spring Framework - Reference Documentation*. Webseite: `http://springcert.sourceforge.net/2.5/core/index.html` (letzter Zugriff August 2014) (siehe S. 15, 16).

[11] Kieker Project. *Die Kieker-API*. Webseite: `http://kieker-monitoring.net/api/1.9/` (letzter Zugriff August 2014) (siehe S. 17, 24).

[12] Apache. *Apache JMeter™*. Webseite: `http://jmeter.apache.org/` (letzter Zugriff August 2014) (siehe S. 19).

[13] Nils Ehmke. *Everything in Sight: Kieker's WebGUI in Action*. Webseite: `http://kieker-monitoring.net/blog/everything-in-sight-kiekers-webgui-in-action/` (letzter Zugriff August 2014) (siehe S. 23).