Lebenszyklusorientierte Performance-Instrumentierung verteilter Anwendungen

Master-Arbeit an der Fachhochschule Wiesbaden
in Kooperation mit dem Cork Institute of Technology

Jan Schäfer - jan.schaefer@informatik.fh-wiesbaden.de

Abstract: Leistungstests während der Entwicklungs- und Testphase, Service Level Management im Betrieb und Leistungsvalidierung nach Modifikationen sind notwendige Vorgänge im Lebenszyklus verteilter Anwendungen. Performance-Instrumentierung ist damit notwendige Voraussetzung für verschiedene Aktivitäten im Lebenszyklus von Softwaresystemen. Diese Arbeit verbindet Performance-Instrumentierung mit der Methodik der Model Driven Architecture zur Integration des Performance-Aspekts in den Softwareentwicklungsprozess. Dadurch können Entwickler die Leistungsüberwachung schon während der Entwurfsphase im UML-Modell der Anwendung planen, ohne Details der später verwendeten Instrumentierungstechnologien zu kennen. Für die im Modell platzierten "Sensoren" wird im Anschluss Instrumentierungscode generiert, der zur Laufzeit Performance-Messdaten liefert, die für Überwachung und Management der Anwendung verwendet werden können. Diese Arbeit ermöglicht eine lebenszyklusorientierte Performance-Instrumentierung bis hin zur Rückführung von Messdaten in das Anwendungsmodell.

1 Motivation und Zielsetzung

Die nachfolgend präsentierte Master-Arbeit entsteht im Rahmen des Forschungsprojekts *Effiziente Tool-gestützte Instrumentierung verteilter Anwendungen* (eMIVA) [Fac06] und wird kooperativ betreut von Prof. Dr. Reinhold Kröger von der Fachhochschule Wiesbaden und Dr. Jeanne Stynes vom Cork Institute of Technology (Irland).

Nicht nur für Diensterbringer und Dienstnutzer gewinnt die Leistungsüberwachung im Rahmen des *Service Level Managements* (SLM) [SMJ00] an Bedeutung. Zu professioneller Softwareentwicklung gehört neben dem Test auf funktionale Korrektheit auch die Planung und kontinuierliche Überwachung der Leistungskenndaten einer Anwendung. Die Überwachung benötigt eine Sensorik (durch *Instrumentierung*), die die gewünschten Messdaten aus der Anwendung heraus zur Laufzeit liefern kann. Für Software gibt es zwei wesentliche Arten der Instrumentierung: Die hier verwendete Quellcode-Instrumentierung bezeichnet den Prozess des Einfügens von zusätzlichen Anweisungen in den Quellcode einer Anwendung. Daneben existiert auch eine Binärcode-Instrumentierung beziehungsweise eine Instrumentierung von Bytecode (z.B. der JVM [Shi06]).

Oft wird die Leistunganalyse einer Anwendung erst bei der Installation oder im Betrieb nach Auftreten von Leistungsengpässen vorgenommen. Dazu kommt, dass viele der Leis-

tungstests bezüglich des Anwendungsverhaltens wenig aussagefähig sind, da entweder zu wenige oder nicht korrelierte Instrumentierungspunkte vorgesehen werden. Gründe hierfür können sowohl fehlende Kenntnis geeigneter Instrumentierungstechnologien seitens der Entwickler als auch technische Einschränkungen der Anwendung sein. Häufig empfinden Entwickler den zusätzlichen Aufwand für die Erstellung und Pflege von nicht-funktionalen Aspekten als Last. Auch werden durch das manuelle Hinzufügen von Instrumentierungscode zum Quellcode verschiedene Aspekte der Anwendung vermischt, so dass eine isolierte Betrachtung nachfolgend (z.B. zur Fehlersuche) erschwert wird. Insgesamt wird es als notwendig angesehen, den Performance-Aspekt frühzeitig im Entwicklungsprozess zu berücksichtigen [JDZ00] [Ang96] [B. 01]. Während Methoden des *Software Performance Engineering* (SPE) [Con02] dazu analytische oder simulative Modelle zur Beurteilung erzeugen, steht hier Leistungsmessung in der entwickelten Software im Fokus. Ziel der Arbeit ist es, durch entsprechende Methoden und Werkzeuge Entwickler während des Softwarelebenszyklus bei der Instrumentierung zu unterstützen.

2 Vorgehensweise und Ergebnisse

Da die Instrumentierung schon in der Entwurfsphase ansetzen soll, wird die *Unified Modelling Language* (UML) als Basis verwendet, da sie heute die meistgenutzte Modellierungssprache im Bereich Softwareentwicklung ist. UML bietet mittels UML-SPT [Obj05] die Möglichkeit, nur zeitliche Bedingungen aber keine korrelierten Instrumentierungsvorgänge zu modellieren. Deshalb wird für diese Arbeit eine Erweiterung in Form eines UML-Instrumentierungsprofils definiert, das zwei Stereotypen namens `Log` und `Measure` für diese Aufgabe zur Verfügung stellt. `Log` ermöglicht das Einfügen von Logpunkten in das Modell, z.B. zur Erzeugung von Statusmeldungen an kritischen Stellen, `Measure` dient der zeitlichen Vermessung von internen Abläufen.

Mit Hilfe dieser Stereotypen erfolgt die plattformunabhängige Definition von Instrumentierungspunkten in der Entwurfsphase der Anwendung, unabhängig von den eingesetzten Technologien. Dabei erfolgt ein Rückgriff auf sogenannte Instrumentierungsmuster, die Beziehungen zwischen verschiedenen Instrumentierungspunkten bezeichnen (z.B. RPC-Muster bestehend aus vier Punkten). So wird das UML-Modell einer verteilten Anwendung durch ein Instrumentierungsmodell ergänzt. Aus diesem Gesamtmodell kann dann unter Verwendung eines Code-Generators der instrumentierte Quellcode erzeugt werden. Die so instrumentierte Anwendung liefert zur Laufzeit Messdaten, die für Monitoring und Management der Anwendung oder als Grundlage für eine detailliertere Instrumentierung des Modells verwendet werden können. Basierend auf diesen Messdaten könnten auch Grenzwerte (*Constraints*) modelliert werden, die über Änderungen des Anwendungsverhaltens im Rahmen von Regressionstests Aufschluss geben. Der Schwerpunkt der Arbeit liegt auf der modellbasierten Instrumentierung und anschließenden Code-Generierung.

Der modellbasierte Instrumentierungsprozess wird in Abbildung 1 veranschaulicht. Das UML-Instrumentierungsprofil wird mit dem UML-Werkzeug MagicDraw 12.0 CE definiert und ins XMI-Format exportiert. Aus dem instrumentierten Modell wird mit Hilfe des *openArchitectureWare* (OAW) [ope06] [Obj03] Code-Generator-Frameworks instrumen-

tierter Java-Quellcode generiert. Hierzu werden *Templates* für die Erstellung des grundlegenden Java-Codes als auch des Instrumentierungscodes benötigt. Für den grundlegenden Java-Code wird die *JavaBasic-Cartridge* (CJB) der *Fornax Platform* (FNX) [For07], und für den Instrumentierungscode werden eigens entwickelte Templates verwendet. Der Ablauf der Code-Generierung wird in einem *Workflow* definiert. Dafür müssen UML-Profil und Anwendungsmodell im XMI-Format vorliegen. Für nicht instrumentierte UML-Elemente ruft der Code-Generator die JavaBasic-Templates auf, für die anderen die Instrumentierungs-Templates. Der dadurch entstehende Quellcode ist somit automatisch instrumentiert, so dass kein Eingriff von Entwicklern nötig ist. Für den generierten Code des `Log`-Stereotyps wird derzeit die *Jakarta Commons Logging* (JCL)-API [Apa06] verwendet, für den `Measure`-Stereotyp die *Application Response Measurement* (ARM)-API [Ope03].

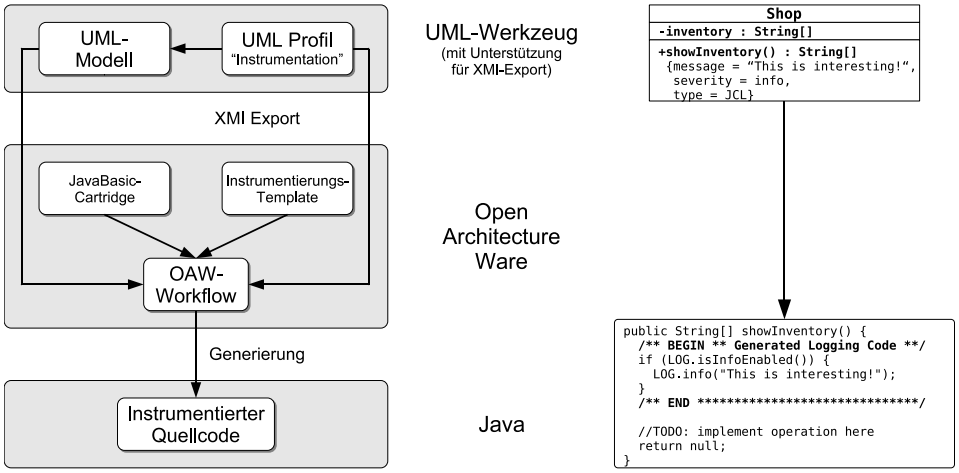

Abbildung 1: Der modellbasierte Instrumentierungsprozess

Die Performance-Instrumentierung von UML-Modellen mit Hilfe eines UML-Instrumentierungsprofils ist im aktuellen Zustand der Arbeit bereits möglich. Bisher können mit den beiden Stereotypen jedoch nur UML-Operationen in Klassendiagrammen erweitert werden, so dass die Granularität der Instrumentierung auf Beginn und Ende eines Methodenaufrufs beschränkt ist. An der Anwendbarkeit auf Sequenz- und Aktionsdiagrammen wird derzeit gearbeitet. Die Anwendung des Ansatzes in einem gemeinsamen Projekt mit Lufthansa Systems und tang-IT ist in Vorbereitung.

Literatur

[Ang96] C. Anglano. Performance Modeling of Heterogeneous Distributed Applications, 1996.

[Apa06] The Apache Software Foundation. *The Jakarta Project Commons Logging*, 2006. http://jakarta.apache.org/commons/logging/.

[B. 01] B. Theelen, J. Voeten, L. van Bokhoven, P. van der Putten, A. Niemegeers and G. Jong. Performance Modeling in the Large: A Case Study, 2001.

[Con02] Connie U. Smith and Lloyd G. Williams. Performance and Scalability of Distributed Software Architectures: An SPE Approach, 2002.

[Fac06] Fachhochschule Wiesbaden - Labor für verteilte Systeme. *Effiziente Tool-gestützte Instrumentierung verteilter Anwendungen (eMIVA)*, 2006. http://wwwvs.informatik.fh-wiesbaden.de/projekte/emiva.html.

[For07] Fornax Platform. *Fornax Platform - An Open Platform for Model Driven Software Development*, 2007. http://www.fornax-platform.org/.

[JDZ00] J. Freiheit J. Dehnert und A. Zimmermann. Workflow Modeling and Performance Evaluation with Colored Stochastic Petri Nets, 2000.

[Obj03] Object Management Group. *OMG's Model Driven Architecture Guide Version 1.0.1*, 2003. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.

[Obj05] Object Management Group. *UML Profile for Schedulability, Performance, and Time*, 2005. http://www.omg.org/cgi-bin/apps/doc?formal/05-01-02.pdf.

[Ope03] The Open Group. *ARM 4.0 Java Language Binding Technical Standard 4.0*, October 2003. http://www.opengroup.org/arm/uploads/40/3945/C037.pdf.

[ope06] openArchitectureWare.org. *openArchitectureWare - A Modular MDA/MDD Generator Framework*, 2006. http://www.openarchitectureware.org/.

[Shi06] ShiftOne. *JRat the Java Runtime Analysis Toolkit*, 2006. http://jrat.sourceforge.net/.

[SMJ00] Rick Sturm, Wayne Morris und Mary Jander. *Foundations of Service Level Management*. SAMS Publishing, April 2000.