Sicherstellung von Performanzeigenschaften durch kontinuierliche Performanztests mit dem KoPeMe Framework

David Georg Reichelt, Lars Braubach
davidgeorg_reichelt@dagere.de, braubach@informatik.uni-hamburg.de

Abstract: Um garantieren zu können, dass Performanzanforderungen von einer Anwendung erfüllt werden, werden Modelle zur Performanzvorhersage am Anfang der Entwicklung und Lasttestwerkzeuge am Ende der Entwicklung eingesetzt. Verschiedene Schwächen dieser Ansätze machen eine neue Methode zur Sicherstellung von Performanzanforderungen notwendig. Die Idee dieses Beitrags ist es, Performanzeigenschaften durch kontinuierliche Tests sicherzustellen. In diesem Papier wird ein Werkzeug vorgestellt, das kontinuierliche Performanztests für Java-Programme ermöglicht. Es stellt eine Schnittstelle bereit, mit der Performanztests implementiert werden können und enthält Möglichkeiten zur Einbindung in Ant und Maven. Weiterhin existiert die Möglichkeit, die Performanzverläufe im Integrationsserver Jenkins zu visualisieren. Die Wirksamkeit des Ansatzes wird mit Hilfe eines Tests von Performanzeigenschaften der Revisionen des Tomcat-Servers durch das Performanztestwerkzeug überprüft.

1 Einleitung

Performanz ist eine zentrale Eigenschaft von Software. Ist sie unzureichend, kann dies zu wirtschaftlichen Einbußen führen. Dennoch ist Performanz im Vergleich zur funktionalen Korrektheit von Programmen ein wenig beachtetes Problem [Mol09, Seite 10].

Ein Ansatz, um Performanz sicherzustellen, ist der "Beheb-Es-Später"(engl. "Fix-It-Later") Ansatz: Zuerst wird funktionale Korrektheit sichergestellt und am Schluss der Entwicklung wird die Performanz betrachtet. Dies ist ineffizient, da zur Performanzverbesserung oft Architekturveränderungen notwendig sind, die am Ende der Entwicklung aufwändiger sind als am Anfang [BMIS03]. Daneben ist zu beobachten, dass Software immer öfter langlebig ist, d.h. Software wird lange weiterentwickelt und während dieser Entwicklung eingesetzt [Par94]. Aktuelle Vorgehensmodelle legen es ebenfalls nahe, schnell in der Praxis einsetzbare Programmteile zu schaffen und diese einzusetzen [BBvB+01]. Wenn Software parallel zur Entwicklung eingesetzt wird, ist es notwendig, kontinuierlich die Erfüllung von Anforderungen, insbesondere den Performanzanforderungen, zu prüfen.

Grundidee dieser Arbeit ist es deshalb, ein Werkzeug zu schaffen, mit dem eine kontinuierliche Betrachtung von Performanz möglich wird. In Abschnitt 2 werden Anforderungen an ein solches Werkzeug genauer herausgearbeitet. Anschließend wird in Abschnitt 3 auf verwandte Arbeiten eingegangen. Danach wird in Abschnitt 4 dargestellt, wie das hier neue Performanzwerkzeug umgesetzt wurde. Ein Evaluationsansatz wird in Abschnitt 5 dargestellt. Abschließend wird eine Zusammenfassung in Abschnitt 6 gegeben.

2 Anforderungen

Die Basisanforderung an ein Performanztestwerkzeug ist es, die Ausführungszeit eines Anwendungsfalls zu messen. Analog zum funktionalen Testen sollte es möglich sein, zu überprüfen, ob Grenzwerte überschritten werden, und ggf. den aktuellen Build als fehlgeschlagen zu markieren. Diese Anforderung wird *Grenzwertüberprüfung* genannt.

Lange Antwortzeiten können u.a. durch Engpässe beim Arbeitsspeicher oder bei den Festplattenzugriffen entstehen. Daneben können andere Messwerte wie die CPU-Auslastung Auskunft über die Gründe von Performanzengpässen geben. Andere Performanzkriterien sollten, um derartige Gründe für schlechte Performanz schneller finden zu können, deshalb ebenfalls erfasst werden. Diese Anforderung wird *Messdatendiversität* genannt.

Durch die Veränderung von funktionalen und Performanzanforderungen kann es dazu kommen, dass Performanzwerte nicht mehr ausreichend sind. In diesem Fall ist es hilfreich, zu wissen, welche Entwicklungsschritte zu welcher Performanzveränderung geführt haben, um sie ggf. rückgängig zu machen. Zur Bereitstellung dieser Information ist die Speicherung und Anzeige des *Performanzverlaufs über Revisionen* notwendig.

Neben diesen zentralen Anforderungen gibt es weitere Anforderungen, die für kontinuierliches Performanztesten sinnvoll sind. Zu nennen sind hier eine Unterstützung von *paralleler Ausführung* von Teilprozessen eines Performanztests sowie eine Lösung, die bei Ausführung von Performanztests auf verschieden leistungsfähiger Hardware Vergleichbarkeit der Testergebnisse gewährleistet. Weiterhin sollte es möglich sein, durch *selbstdefinierte Datenmessung* Messwerte im Quelltext und nicht durch externe Messung zu bestimmen. Diese Probleme werden im Rahmen dieses Papiers nicht genauer behandelt.

3 Verwandte Arbeiten

Etablierte Methoden, um Performanz in der Softwareentwicklung zu verbessern, sind Lasttests, Performanzmodelle sowie Performanztests. Diese werden im Folgenden dargestellt.

Die Methode, Lasttests am Schluss der Entwicklung auszuführen, wird durch diverse Werkzeuge[1] unterstützt und durch Anleitungen wie [Mol09] beschrieben. Zur Überprüfung von Performanzanforderungen sind Lasttests vor dem Einsatz eines neuen Releases eines Programmes sinnvoll. Refactoring am Ende der Entwicklung ist jedoch oft aufwändiger als früheres Refactoring. Deshalb ist es sinnvoll, Performanz eher zu betrachten.

Hierfür existieren analytische Performanzmodelle. Bei diesen wird, ausgehend von Performanzannotationen an Architekturmodellen bspw. mit UML STP[Gro02] eine Performanzschätzung entwickelt. Als Modelle werden hier u.a. Wartschlangenmodelle, Petrinetze und Prozessalgebren verwendet [Bar09, Seiten 22-29]. Ähnlich arbeiten Simulationsmodelle [BGM03]: Hier wird statt des analytischen Modells eine Simulation eingesetzt, die unter bestimmen Szenarien Performanzwerte der Softwarearchitektur schätzt.

[1] Lasttestwerkzeuge sind u.a. JGrinder (http://jgrinder.sourceforge.net/) und JMeter(http://jmeter.apache.org/)

Ein Werkzeug zur Performanzvorhersage auf Basis der Architektur ist Palladio.[2]

Performanzmodelle können Engpässe frühzeitig aufdecken und Wege zur Vermeidung aufzeigen. Problematisch ist, dass weder die Architektur noch die Performanz einzelner Methoden vor dem Ende der Entwicklung exakt bestimmt werden kann. Deshalb können Performanzmodelle nur einen Teil der Performanzprobleme aufdecken. Es ist also eine Methode notwendig, die Performanz vor dem Ende der Entwicklung exakt misst, um Refactorings zu vermeiden. [Pod08] argumentiert dafür, neben den herkömmlichen Methoden Einzelnutzer-Performanztests als Unit-Tests durchzuführen, denn eine gute Performanz für einen einzelnen Nutzer ist Voraussetzung für eine gute Performanz bei großer Last.

Für Performanztests existieren bereits einige Ansätze. Das verbreitete Testframework JUnit ermöglicht es, über die *@Timeout*-Notation *Grenzwertüberprüfung* durchzuführen. Andere der genannten Anforderungen unterstützt JUnit nicht. JUnitBench,[3] eine Erweiterung von JUnit, ermöglicht zusätzlich das Speichern der Aufrufzeiten über verschiedene Revisionen und das anschließende Visualisieren des *Messwertverlauf über Revisionen*. Eine andere JUnit-Erweiterung, JUnitPerf[4], ermöglicht es, die Ausführungszeit nebenläufiger Tests zu prüfen. Beide JUnit-Erweiterungen erfüllen außer den genannten keine zusätzlichen Anforderungen. Ein weiteres Werkzeug, das die Performanzbetrachtung ermöglicht, ist das Performance Plugin für Jenkins.[5] Es ermöglicht das Speichern von Performanztestergebnissen von JUnit und JMeter. Damit ermöglicht es über JUnit die *Grenzwertüberprüfung* sowie die Erstellung des *Performanzverlaufs über Revisionen*. Eine Messung anderer Performanzkriterien ist nicht möglich.

Insgesamt erfüllt kein Werkzeug die genannten Mindestanforderungen für Werkzeug für kontinuierliche Performanztests. Aus diesem Grund wurde ein eigenes Framework zur **Ko**ntinuierlichen **Pe**rformanz**me**ssung erstellt: KoPeMe.[6]

4 Konzeption und Umsetzung

Um die in Kapitel 2 dargestellten Anforderungen umzusetzen, ist es notwendig, an drei Stellen des üblichen Buildprozesses Erweiterungen zu schaffen: Es muss die Möglichkeit geschaffen werden, Performanztests zu definieren und auszuführen, es muss die Möglichkeit geben, diese im Buildprozess aufzurufen und es muss die Möglichkeit geben, die Ergebnisse der Performanztests zu visualisieren. Das KoPeMe-Framework ermöglicht die Definition der Performanztests in Java, das Aufrufen der Performanztests im Buildprozess mit Maven und Ant sowie das Visualisieren der Ergebnisse in Jenkins. Im Folgenden werden die einzelnen Komponenten erläutert.

Die Tests in Java könnten als Erweiterung eines der verbreiteten Testframeworks JUnit oder TestNG oder als eigenständiges Werkzeug entwickelt werden. Die Tests wur-

[2] Offizielle Webseite von Palladio: http://www.palladio-simulator.com
[3] https://code.google.com/p/junitbench/
[4] http://www.clarkware.com/software/JUnitPerf.html
[5] https://wiki.jenkins-ci.org/display/JENKINS/Performance+Plugin
[6] Offizielle Webseite: www.dagere.de/KoPeMe. Quelltext unter https://github.com/DaGeRe/KoPeMe.

den eigenständig und als JUnit-Tests implementiert. Durch die Umsetzung der JUnit-Performanz-Tests ist die Umwandlung bestehender Tests in Performanztests unproblematisch. Die JUnit-Tests umfassen sämtliche Funktionalitäten der eigenständigen Tests und müssen durch JUnit-spezifische Annotationen gekennzeichnet werden. Ein eigenständiger Test kann folgendermaßen aussehen:

Listing 1: KoPeMe-Test

```
@PerformanceTest(executionTimes=5, warmupExecutions=2 )
  public void testMoebelkauf(final TestResult tr) {
    tr.startCollection();
    //Hier werden die Berechnungen ausgeführt
    tr.stopCollection();
    tr.setChecker(new Checker() {
      @Override
      public void checkValues(TestResult tr) {
        MatcherAssert.assertThat(
          tr.getValue(CPUUsageCollector.class.getName()),
          Matchers.greaterThan(10L));
      }
    });
  }
```

Messungen müssen mehrmals ausgeführt werden, um Verfälschungen, bspw. durch das initiale Laden einer Klasse, zu vermeiden. In der Annotation wird deshalb angegeben, wie oft die Methode ohne Messung zum Aufwärmen (*warmupExecutions*) und mit Messung (*executions*) aufgerufen werden soll. Diese Parameter sind optional, die Standardwerte sind 2 und 5. Die Messung verschiedener Daten zur Umsetzung der *Messdatendiversität* erfolgt über beliebig erweiterbare Datenkollektor-Objekte. Derzeitig sind Standardkollektoren für Zeit, Arbeitsspeicherauslastung und CPU-Auslastung vorhanden. Es ist bspw. denkbar, einen Kollektor zu schreiben, der nur die Auslastung eines Prozessors in einer Mehrkernmaschine misst. Mit *setCollectors* kann festgelegt werden, welche Kollektoren genutzt werden sollen. Da es unerwünscht ist, die Performanz der Sicherstellungen in die Performanz des Anwendungsfalls hineinzurechnen, wurde die Möglichkeit geschaffen, mit *startCollection* und *stopCollection* zu markieren, an welchem Punkt die Datensammlung beginnen bzw. enden soll. Um selbst im Quelltext Messwerte festzulegen, kann nach *stopCollection* über *addValue(String key, long value)* ein Messwert hinzugefügt werden.

Standardmäßig wird über die Ausführungen ein Durchschnitt gebildet und dieser gespeichert. Es ist möglich, über das Setzen eines *MeasureSummarizer* andere Wertzusammenfassungen wie Median, Maximum und Minimum zu wählen oder eigene Verfahren zu implementieren. Die Zusicherungen werden als Objekt, das die *Checker*-Schnittstelle implementiert, übergeben. Diese wird nach dem mehrmaligen Aufruf der Methode ausgeführt. Neben der Performanzüberprüfung durch ein *Checker*-Objekt können Grenzwerte über Annotationen darzustellen. Auf diesem Weg wird *Grenzwertüberprüfung* ermöglicht. Mit einer ähnlichen Synthax können parallele Tests beschrieben werden.

JUnit-KoPeMe-Tests werden über die jeweiligen JUnit-Werkzeuge in den Buildprozess eingebunden. So ist auch eine Testausführung in einer Entwicklungsumgebung möglich. Die Ausführung der eigenständigen Tests durch Ant und Maven wurde basierend auf einer gemeinsamen Konsolenaufrufsklasse, die die Tests in der Konsole ausführt, umgesetzt.

Dadurch lassen sich Erweiterungen in beiden mit einer Implementierung umsetzen.

Die Ausgabedaten der Performanzmessung sind YAML-Dateien.[7] In diesen Daten wird für jedes Performanzkriterium die Abbildung des Ausführungszeitpunktes auf den Messwert gespeichert. Diese können im KoPeMe-Jenkins-Plugin visualisiert werden. So wird der *Performanzverlauf über Revisionen* erzeugt.

Insgesamt ermöglicht KoPeMe das Spezifizieren von Tests, das Ausführen sowie das Visualisieren der Ergebnisse. Damit sind mit KoPeMe kontinuierliche Performanztests möglich.

5 Evaluation

Es wurde eine quantitative Evaluation an Tomcat 6 durchgeführt. Da Tomcat 6 ein umfangreiches, frei verfügbares Projekt mit vielen Beteiligten und performanzkritischen Bestandteilen ist, wurde es für die Evaluation ausgewählt. Die Tests wurden auf einem PC mit i5-Prozessor und Ubuntu 12.04 ausgeführt.

Bei den Anwendungsfällen Laden einer Servlet- bzw. JSF-Seite, die ihre Darstellungstexte jeweils durch Java-Methoden erhielten, zeigten sich gravierende Performanzunterschiede zwischen verschiedenen Revisionen. Die Abbildungen 1 und 2 zeigen auf der X-Achse die Zeit der Ausführung und auf der Y-Achse die Antwortzeit des jeweiligen Testfalls. Die Zeit der Ausführung lässt sich auf eine Revision abbilden.

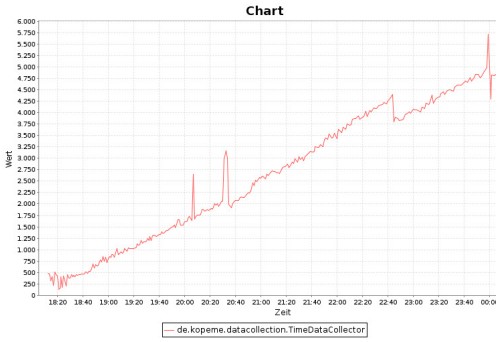

Abbildung 1: Verlauf der JSF-Downloadzeit

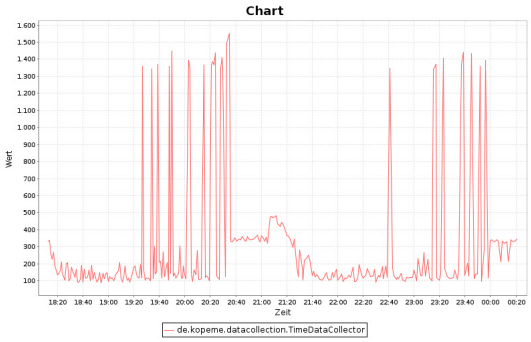

Abbildung 2: Verlauf der Servlet-Downloadzeit

Bei der JSF-Seite (Abbildung 1) steigt die Antwortzeit stetig an. Dies deutet darauf hin, dass in dem JSF-Renderingquelltext immer neue Veränderungen hinzukamen, die den Quelltext langsamer machen. Es ist davon auszugehen, dass bei kontinuierlicher Performanzbetrachtung performanzverbessernde Änderungen früher durchgeführt worden wären, wie bspw. in der Revision, die 22:40 getestet wurde. Beim Laden einer Servlet-Seite (Abbildung 2) zeigt sich noch deutlicher, dass ein KoPeMe-Einsatz die Entwicklung effizienter gemacht hätte: Die hohen, unregelmäßig auftretenden Ausschläge wären bei dem Einsatz von Performanztests schon während des Einreichens bzw. kurz danach entdeckt wurden. Ingesamt deutet die Evaluierung darauf hin, dass der Einsatz von kontinuierlichen Performanztests eine effizientere Softwareentwicklung unterstützen kann.

[7] Das YAML-Format (http://yaml.org/) zeichnet sich durch besonders gute Lesbarkeit und geringen Speicherverbrauch aus.

6 Zusammenfassung und Ausblick

In diesem Beitrag wurde ein Ansatz zur Softwareentwicklung vorgestellt, der eine Entwicklung mit kontinuierlicher Performanzmessung und -überprüfung ermöglicht. Hierzu wird vorgeschlagen, ein Test-basiertes Vorgehen auch für nicht-funktionale Anforderungen der Software zu etablieren und diese bei jedem Build zu messen und sowohl gegenüber den initialen Anforderungen als auch im historischen Revisionsvergleich zu betrachten und bewerten. Im Gegensatz zu bestehenden praktischen Ansätzen wird dabei Performanz als Spektrum an Kriterien wie Ausführungszeit, CPU-Auslastung und Arbeitsspeicherverbrauch betrachtet. Für die Erstellung von Performanztests wird eine an JUnit angelehnte Spezifikation verwendet, die wenige zusätzliche Annotationen einführt. Die Ausführung im Build-Prozess wird sowohl für Ant- als auch für Maven-basierte Projekte unterstützt und eine Visualisierung der Messungen wird als Zusatz für den Jenkins Integrationsserver bereitgestellt. Die Wirksamkeit des Ansatzes und des Werkzeuges wurden anhand einer quantitiven Evaluierung des Tomcat-Servers untersucht. Hierbei konnten Performanzsprünge zwischen Revisionen nachgewiesen werden, die bei Nutzung des Ansatzes u.U. hätten vermieden werden können. Als wichtiger Teil zukünftiger Arbeiten soll untersucht werden, ob und wie die Ergebnisse der Performanztests auch bei Ausführung auf unterschiedlicher Hardware weiter genutzt werden können.

Literatur

[Bar09] M. Barth. *Entwicklung und Bewertung zeitkritischer Softwaremodelle: Simulationsbasierter Ansatz und Methodik*. Dissertation, 2009.

[BBvB+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland und D. Thomas. Manifesto for Agile Software Development, 2001.

[BGM03] S. Balsamo, M. Grosso und M. Marzolla. Towards Simulation-Based Performance Modeling of UML Specifications. Bericht, 2003.

[BMIS03] S. Balsamo, A. Di Marco, P. Inverardi und M. Simeoni. Software Performance: state of the art and perspectives. 2003.

[Gro02] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. OMG Adopted Specification ptc/02-03-02, Juli 2002.

[Mol09] I. Molyneaux. *The Art of Application Performance Testing - Help for Programmers and Quality Assurance.* O'Reilly, 2009.

[Par94] D. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, Seiten 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Pod08] A. Podelko. Agile Performance Testing. In *Int. CMG Conference*, Seiten 267–278. Computer Measurement Group, 2008.