

Performance Anti-Patterns

Aus Fehlern lernen

■ VON MIRKO NOVAKOVIC UND NICK PANIENSKI

Performance ist eine kritische Anforderung in Java EE-Projekten. In vielen Tuning-Projekten konnten Anti-Patterns identifiziert werden, die einen hohen Wiedererkennungswert haben und so in den eigenen Projekten vermieden werden können.



Java Enterprise-Projekte haben in vielen Unternehmen eine zunehmend integrative Bedeutung. Moderne Architekturansätze wie SOA, die viele externe und interne Services integrieren, stellen neue Anforderungen an Performance und Stabilität und erschweren gleichzeitig die Analyse und Behebung von Problemen durch eine immer größere Verteilung der Anwendung. In so genannten „Troubleshooting“-Einsätzen wurden praktische Erfahrungen gesammelt, wie Performance- und Stabilitätsprobleme vermieden werden können, um so die Qualität dieser komplexen Architekturen zu verbessern. Der Artikel beschreibt Performance Anti-Patterns, die helfen, Probleme zu erkennen und zu vermeiden.

Patterns (Entwurfsmuster) stammen aus dem Bauwesen und wurden erstmals von Christopher Alexander [1] aufgeschrieben und später auf das Software Engineering übertragen. Aber erst seit der Veröffentlichung des Buchs „Design Patterns“ von Erich Gamma [2] und seinen Kollegen haben Patterns die breite Masse erreicht und eine große Bedeutung in der Software-Entwicklung bekommen.

Design Patterns beschreiben dabei eine bewährte Lösung für ein objektorientiertes Entwurfsproblem in einer einheitlich definierten Form. In modernen Frameworks wie Spring findet man die beschriebenen Patterns (z.B. Factory oder Proxy) überall im Code, was das Verständnis stark erleichtert. In der Java Enterprise-Welt existieren Java EE-Patterns und Blueprints von Sun [3] und auch in vielen Spezialdisziplinen, wie EJB [4] oder EAI [5]. Diese Vorlagen helfen bei der täglichen Arbeit und vermeiden, das Rad immer wieder neu zu erfinden.

Anti-Patterns und Performance

Anti-Patterns sind hingegen weniger verbreitet, „Bitter Java“ von Bruce Tate [6] liefert eine Liste von Java Anti-Patterns, die teilweise auch das Thema Java Performance betreffen. Anti-Patterns sind Beispiele für schlecht durchgeführte Lösungsmuster und geben auf diese Weise Hinweise zur Verbesserung. Auch bei Anti-Pattern hat sich eine einheitliche Dokumentation bewährt, vor allem ein eindeutiger Name ist wichtig, damit in Teams jeder versteht, worum es geht.

(Wenn man heute von „Singleton“ redet, weiß jeder, was gemeint ist. Ob das vor „Design Patterns“ auch der Fall war, das darf in Frage gestellt werden – hier liegt ein entscheidender Wert von Patterns.)

Performance Anti-Patterns spezialisieren allgemeine Anti-Patterns auf performance- und stabilitätsrelevante Problemstellungen. Dies bedeutet, dass ein „erfolgreich“ umgesetztes Performance Anti-Pattern normalerweise zu schlechten Antwortzeiten, geringem Durchsatz und/oder schlecht verfügbaren Anwendungen führt – und wahrscheinlich auch unzufriedene Benutzer und verärgerte Manager mit sich bringt. Problematisch ist, dass die Umsetzung eines Design Patterns nicht selten zu Performance-Problemen führt, sodass aus einem Design Pattern auch mal ein Performance Anti-Pattern werden kann.

Ein gutes Beispiel hierfür ist die Java Petstore Blueprint-Applikation, die von Microsoft als Performance Benchmark Application im Vergleich zu .NET „missbraucht“ wurde [7] und ein eher schlechtes Performance-Verhalten aufwies. Als Reaktion wurde JPetstore [8]

gebaut, eine Performance-optimierte Variante der Applikation, in der viele Java EE-Patterns eliminiert wurden. Unter anderem wurde auch das schlanke O/R-Mapping Framework iBatis [9] hervorgebracht – hier als Lösung für das EJB 2.0 Entity Bean Performance Anti-Pattern. (Auch das Spring Framework ist als Komplexitäts-Anti-Pattern-Lösung für EJB entstanden [10] – EJB hatte also auch viele positive Effekte auf die Java-Welt.)

Performance-Probleme in Java-Projekten

Aus den Erfahrungen in vielen Troubleshooting-Einsätzen lassen sich drei Kernprobleme in Java-Anwendungen identifizieren, die zu Performance- und Stabilitätsproblemen führen:

- Memory-Probleme (Memory Leaks, Garbage Collection und Session-Größen)
- DB-Probleme (Statement-Menge, O/R-Mapping, DB-Design und Tuning)
- Falscher Einsatz von externen Frameworks.

Neben diesen Code- und Design-Problemen werden immer wieder organisatorische und architektonische Gründe für Performance-Probleme aufgedeckt, die in der Regel sogar schwerwiegender sind und sich nur mit großem Aufwand, z.B. durch Refactoring, beheben lassen. Nachfolgend werden die Performance Anti-Patterns daher in drei Kategorien eingeteilt:

- *Organisatorische Anti-Patterns* betreffen die Organisation, das Vorgehen

und die Kommunikation in einem Projekt.

- *Architektur-Anti-Patterns* betreffen die strategischen und strukturellen Entscheidungen für eine Applikation.
- *Umsetzungs-Anti-Patterns* betreffen die Implementierung und die Konfiguration der Anwendungs- und Infrastrukturkomponenten.

In den nachfolgenden Abschnitten werden Beispiele für Performance Anti-Patterns in den unterschiedlichen Kategorien gegeben.

Organisatorische Performance Anti-Patterns

Bei den organisatorischen Performance Anti-Patterns gibt es zwei Kandidaten, die man in den meisten Troubleshooting-

Performance-Werkzeuge

Ein Arzt benötigt für die Diagnose der Beschwerden eines Patienten Werkzeuge, damit die Therapie richtig gewählt werden kann und der Patient hoffentlich schnell wieder gesund wird. In der Medizin erfolgt die Diagnosestellung durch eine klinische Untersuchung und durch den Einsatz von Werkzeugen, wie z.B. Röntgengeräten, Ultraschall oder Magnetresonanz. Die Ergänzung der klinischen Untersuchung durch diese Werkzeuge hat es erst möglich gemacht, bestimmte Krankheiten genau zu diagnostizieren und zu heilen. In der Performance-Analyse wird dasselbe Diagnoseprinzip angewandt, und es werden auch Werkzeuge benötigt, um Performance- oder Stabilitäts-„Beschwerden“ zu diagnostizieren. Leider ist das „Hand auflegen“ in der Software-Entwicklung immer noch der Normalfall. Die hier beschriebenen Software-Komponenten sind ein Werkzeugkasten für den Performance Tuner.

Profiler und Memory Debugger

Profiler und Memory Debugger sind Entwickler-Tools, die Laufzeitanalysen der Anwendung bis auf Code-Ebene ermöglichen und es erlauben, eine Momentaufnahme des Heaps zu erzeugen, um die Objektgrafen auf mögliche Speicherlöcher zu untersuchen. Fast alle Profiler nutzen dafür das Java Virtual Machine Tooling Interface (JVMTI), das u.a. eine Bytecode-Instrumentierung der Anwendung zur Laufzeit ermöglicht. Profiler und Memory Debugger eignen sich gut für die Detailanalyse eines Performance-Problems, haben aber in der Regel einen viel zu hohen Overhead, um unter Last oder in Produktion eingesetzt zu werden. Moderne JVMs bieten mittlerweile auch viele Performance Analyse-Tools, wie z.B. *jhat* zur Analyse des Heaps.

Die BEA JRockit JVM hat sogar einen Profiler und ein Memoryleak Analyse-Tool integriert. Optimal ist die Möglichkeit, das Tool mithilfe von Ant zu steuern, um automatisierte Profiling mit dem Build erzeugen zu können. JProbe, JProfiler und yourKit sind kommerzielle Profiling-Tools, die alle benötigten Funktionen integriert haben.

Diagnose-Tools

Diagnose-Tools bieten die Möglichkeit, Performance- und Memory-Analysen unter realen Bedingungen, d.h. in Produktion oder in der Lasttest-Umgebung durchzuführen. Dabei werden i. d. R. neben den Laufzeiten auch Informationen über den Application Server, das Betriebssystem und beteiligte Randsysteme (DB, MOM) gesammelt und aggregiert. Die meisten Tools arbeiten dabei mit statistischen Daten, wobei es mittlerweile moderne Werkzeuge gibt, die den realen Zugriffspfad von jedem Aufruf messen und so gezielte Fehleranalysen zulassen. Der Overhead dieser Tools sollte sich zwischen 3–10 Prozent bewegen. Mithilfe dieser Diagnose-Werkzeuge lassen sich Performance- und Stabilitäts-Probleme analysieren, die nur unter Last auftreten. *dynaTrace diagnostics*, *Quest PerformSure* und *CA/Wily Introscope* sind drei Vertreter dieser Kategorie.

Monitoring-Tools

Monitoring-Tools sind Werkzeuge für die Überwachung der Anwendung im Betrieb. Sie bieten meistens schwellwertbasierte Alarm und Eskalationsfunktionen und haben so genannte Dashboards für die Überwachung und Analyse von Problemen. Viele Monitoring-Tools haben zudem Diagnosefunktionen, die bei Bedarf oder auf

Basis einer Eskalation eingeschaltet werden oder immer mitlaufen (meistens aber auf Basis von Filtern eingeschränkt, um den Overhead gering zu halten). Monitoring ist wichtig, um proaktiv reagieren zu können, wenn sich Engpässe anbahnen. Bei bestehendem Service Level Agreement (SLA) sind diese Tools zudem in der Lage, diese zu kontrollieren und entsprechende Reports zu generieren. Mithilfe der Java Management Extension (JMX) lassen sich eigene Applikationsdaten und Frameworks in das Monitoring integrieren. *dynaTrace diagnostics*, *Quest Foglight*, *IBM IT-CAM* und *CA/Wily Introscope* sind Beispiele für Tools dieser Kategorie.

Lastgeneratoren

Lastgeneratoren zeichnen Benutzerinteraktionen auf und können diese mit einer bestimmten Anzahl virtuelle Nutzer wieder abspielen, um so gezielt Last auf einem System zu erzeugen. Die Tools unterscheiden sich vor allem in den unterstützten Protokollen (HTTP, HTTPS, RMI, CORBA, ...), den eingesetzten Skriptsprachen und dem Funktionsumfang und der Anpassbarkeit der APIs. Je komplexer die simulierten Anwendungsfälle, desto wichtiger ist die komfortable Entwicklung und Erweiterbarkeit der Skripte um eigene Daten und Funktionen. Auch AJAX stellt neue Anforderungen an Lasttest-Tools, die bei einer Toolauswahl berücksichtigt werden sollten. Über das Thema Lasttest-Tool folgt in einer späteren Ausgabe des *Java Magazins* ein detaillierter Überblick und Erfahrungsbericht. *Borland Silk-Performer*, *HP/Mercury Loadrunner* und auch das Open-Source-Tool *JMeter* sind bekannte Vertreter dieser Kategorie.

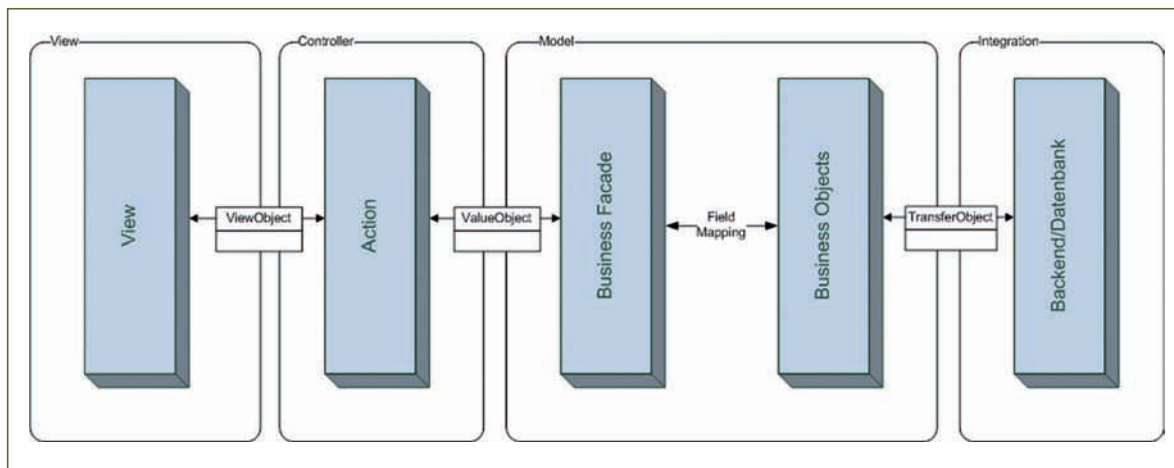


Abb. 1: Multi-Layering-Architektur

Projekten antrifft: Das „Paralleles Schrauben“ und das „Schuss ins Dunkle“ Performance Anti-Pattern (bitte E-Mail an die Autoren, wenn jemand eine zündende Idee für bessere Namen hat). Beim „Parallelen Schrauben“-Anti-Pattern werden die Entwickler von der Projektleitung mit geballter Kraft auf die Performance-Probleme losgelassen. Dabei entwickelt jedes Teammitglied seine eigenen Tuning-Theorien und setzt möglichst schnell erste Maßnahmen um. In regelmäßigen Abständen wird die Performance der verbesserten Anwendung gemessen – meistens mit wenig Erfolg.

Die Konsequenz aus diesem Vorgehen ist, dass sehr hoher personeller Aufwand erzeugt wird und nur wenig Zählbares dabei herkommt. Oftmals resultiert dies daraus, dass sich die umgesetzten Tuning-Maßnahmen gegenseitig beeinflussen und der Effekt jeder einzelnen Umsetzung sich nicht bewerten lässt.

Zum Beispiel kann ein Tuning eines Entwicklers 20 Prozent Verbesserung gebracht haben, eine parallel durchgeführte Maßnahme hatte aber eine 50 Prozent verschlechternde Auswirkung. Nach der Messung werden dann fälschlicherweise beide Maßnahmen verworfen.

Als negativer Nebeneffekt werden auch neue fachliche Fehler programmiert, die dann das Performance-Testen erschweren oder verhindern.

Um dieses Anti-Pattern zu lösen, sollte ein zentrales Performance-Team aufgesetzt werden, das alle Messungen und Maßnahmen erfasst, bewertet und die

erfolgreichste Maßnahme umsetzt. Entsprechende Verstärkung durch die einzelnen Entwickler wird punktuell dazugeholt. Das Tuning erfolgt iterativ und ist beendet, wenn die Performance-Ziele erreicht sind. (Keine Performance-Ziele zu haben, ist auch ein organisatorisches Performance Anti-Pattern, das häufig zu finden ist.) Zur Bewertung der Tuning-Maßnahmen werden entsprechende Performance-Tuning-Werkzeuge benötigt (siehe Kasten).

Das „Schuss ins Dunkle“-Anti-Pattern beschreibt Teams, die keine entsprechenden Tools haben und somit manuell versuchen, Performance-Probleme zu suchen und zu bewerten. Dies bedeutet in der Praxis, dass Code Inspections oder Walkthroughs [11] zum Finden von Fehlern genutzt werden – dies ist aufwändig und führt nur mit viel Glück zum Ziel. Die produktive Last, die Infrastruktur und externe Systeme können dabei nicht oder nur teilweise berücksichtigt werden. Auch so genannte Microbenchmarks betrachten nur Ausschnitte einer Anwendung und können zu falschen Rückschlüssen führen, wie Brian Goetz in seinen Artikeln beschreibt [12]. Das Stochern im Dunklen führt dann in vielen Projekten zu „Fingerpointing“ zwischen Betrieb und Entwicklung (Stichwort: In Eclipse ist alles schnell und in Produktion dann langsam), Stress im gesamten Entwicklungsteam, und im Endeffekt resultiert dies in einem Vertrauensverlust des Teams beim Management.

Die Lösung dieses Anti-Patterns ist einfach, erfordert aber eine gewisse In-

vestition in Werkzeuge und Ausbildung der Mitarbeiter. „Measure, don't guess“ ist die einzige Möglichkeit, erfolgreich Performance- und Stabilitätsprobleme zu finden und zu beheben – hierfür werden Monitoring-, Diagnose- und Profiling-Tools benötigt, die alle erforderlichen Daten aus den beteiligten Systemen liefern. Zusätzlich sollte ein Lasttest-Tool vorhanden sein, um die reale Last im Alltagsbetrieb nachbilden zu können und Produktionsprobleme reproduzierbar zu machen. Eine entsprechende produktionsnahe Testumgebung muss daher auch vorhanden sein. Neben den Tools muss ein Performance-Management-Prozess von der Entwicklung bis zum Betrieb etabliert und die Mitarbeiter im Umgang mit den Werkzeugen und den Technologien entsprechend geschult werden.

Beide beschriebenen organisatorischen Performance Anti-Pattern verstärken sich, wenn sie in Kombination auftreten, d.h. „Paralleles Schrauben“ an der Blackbox. In den meisten Troubleshooting-Einsätzen hätte sich die Investition in vernünftige Prozesse, Tools und Schulungen deshalb schnell bezahlt gemacht. Neben diesen organisatorischen Anti-Patterns gibt es noch weitere, wie die „Testdaten-Falle“, bei der mit ungünstigen Daten und Anwendungsfällen gemessen wird und so falsche positive Performance-Aussagen getroffen werden. Oder „Falsches Timing“, womit der Zeitpunkt im Entwicklungsprozess gemeint ist, an dem Performance be-

trachtet werden sollte. Am Ende der Entwicklung Lasttests durchzuführen, ist meistens zu spät – genauso ist es ungünstig, Entscheidungen schon zu Beginn zu treffen, ohne Performance-Daten erheben zu können.

Architektur Performance Anti-Patterns

Performance Anti-Patterns gibt es viele in der Architektur – hier werden vor allem die beschrieben, die man sehr häufig in Projekten findet. Das „Multi Layering“-Anti-Pattern beschreibt eine Architektur, die versucht, eine hohe Abstraktion durch möglichst viele unabhängige, logische Anwendungsschichten zu erreichen. Als Entwickler erkennt man eine solche Architektur sehr schnell daran, dass ein großer Teil der Zeit beim Mapping und Konvertieren von Daten verloren geht und ein einfacher Durchgriff von der Oberfläche auf die Datenbank komplex ist.

Solche Architekturen entstehen meistens, weil die Anwendung möglichst flexibel gehalten werden soll, damit z.B. GUIs einfach und schnell ausgetauscht und die Abhängigkeiten zu anderen Systemen und Komponenten gering gehalten werden können. Die Entkopplung der Schichten führt zu Performance-Verlusten beim Mapping und Austausch der Daten – vor allem dann, wenn die Schichten auch physikalisch getrennt sind und der Datenaustausch über Remoting-Technologien wie SOAP oder RMI-IIOP erfolgt. Die vielen Mapping- und Konvertierungsoperationen können auch zu einer höheren Garbage Collection-Aktivität führen, was als „Cycling-Object-Problem“ bekannt ist. Als Lösung dieses Anti-Patterns sollten die Architekturtreiber genau durchleuchtet werden, um zu klären, welche Flexibilität und Entkopplung notwendig ist. Neue Framework-Ansätze, wie beispielsweise JBoss Seam [13], haben sich des Problems angenommen und versuchen, das Mapping von Daten möglichst vollständig zu vermeiden.

Ein weiteres Architektur-Anti-Pattern ist der so genannte „Session Cache“. Dabei wird die Web Session einer Anwendung als großer Datencache missbraucht und so die Skalierbarkeit der Anwendung

stark eingeschränkt. Es wurden bei Tuning-Einsätzen schon häufig Session-Größen weit über 1MB gemessen – in den meisten Fällen kennt kein Teammitglied den genauen Inhalt der Session. Große Sessions führen dazu, dass der Java Heap sehr stark ausgelastet wird und nur eine geringe Anzahl paralleler Benutzer möglich ist. Gerade beim Clustern von Anwendungen mit Session-Replikation ist, je nach eingesetzter Technologie, der Performance-Verlust durch Serialisierung und Datenübertragung sehr hoch. Einige Projekte helfen sich damit, neue Hardware und mehr Speicher anzuschaffen, aber auf Dauer ist dies eine sehr teure und riskante Lösung.

Session Caches entstehen, weil in der Architektur der Anwendung nicht klar definiert wurde, welche Daten sitzungsabhängig und welche persistent, also zu jedem Zeitpunkt wiederherstellbar, sind. Während der Entwicklung werden dann schnell erstmal alle Daten in der Sitzung abgelegt, weil dies eine sehr komfortable Lösung ist – oft werden diese Daten auch nicht mehr aus der Sitzung entfernt. Für die Lösung dieses Problems sollte zunächst eine Memory-Analyse der Sitzung mithilfe eines Heapdumps aus der Produktion durchgeführt werden und die Sitzung um Daten bereinigt werden, die nicht sitzungsabhängig sind. Caching kann die Performance positiv beeinflussen, wenn der Prozess, die Daten zu holen, performancekritisch ist – beispielsweise bei Datenbankzugriffen. Optimal erfolgt das Caching dann transparent für den Entwickler innerhalb des Frameworks. Hibernate bietet z.B. einen First und einen Second Level Cache, um den Zugriff auf Daten zu optimieren, aber Vorsicht: Die Konfiguration und das Tuning solcher Frameworks sollte von Experten durchgeführt werden, sonst hat man schnell ein neues Performance Anti-Pattern.

Umsetzungs-Performance-Anti-Patterns

Es gibt viele Java Performance Anti-Pattern und Tuning-Tipps – das Problem bei diesen technologischen Anti-Patterns ist, dass sie stark von Java-Version und Hersteller abhängig sind und vor allem auch vom Anwendungsfall. Ein sehr häufiges

Anzeige

Anti-Pattern ist aber das „Unterschätzte Frontend“. Bei Webanwendungen ist das Frontend oft die Performance-Achillesverse. HTML- und JavaScript-Entwicklung sind „echten“ Anwendungsentwicklern häufig nur lästiges Beiwerk und werden daher oft nur unzureichend auf Performance hin optimiert. Auch bei immer stärkerer Verbreitung von DSL

Um Performance-Engpässe zu vermeiden, sollten die eingesetzten Frameworks richtig verstanden werden.

ist die Anbindung häufig immer noch ein Bottleneck – gerade, wenn es sich um eine mobile Anbindung über UMTS oder GPRS handelt. Webanwendungen werden – angetrieben durch den Web-2.0-Hype – immer komplexer und nähern sich von der Funktion immer stärker Desktopanwendungen an. Dieser Komfort führt durch viele Sever Roundtrips und große Seiten zu verlängerten Wartezeiten und höherer Server- und Netzwerklast.

Es gibt eine ganze Palette von Lösungen, um webbasierte Oberflächen zu optimieren. Das Komprimieren der HTML-Seiten mit GZip reduziert die übertragene Datenmenge erheblich und wird seit HTTP 1.1 von allen Browsern unterstützt. Webserver wie Apache verfügen über entsprechende Module (*mod_gzip*), um die Komprimierung ohne Änderung der Anwendung durchzuführen. Die Seitengrößen können aber auch im HTML schnell reduziert werden, indem konsequent CSS eingesetzt wird und CSS und JavaScript Sourcen in eigene Dateien ausgelagert werden – so können diese besser vom Browser gecached werden. Auch AJAX kann – richtig eingesetzt – die Performance deutlich verbessern, weil das komplette Neuladen von Webseiten eingespart werden kann, indem z.B. nur die Inhalte von Listen neu übertragen werden.

Aber schon in der Analyse kann die Performance der Oberflächen deutlich verbessert werden, indem die Inhalte der Seiten an die Anforderungen der Benutzer angepasst werden. Wenn beispielsweise nur die Felder auf einer Seite angezeigt

werden, die in 80 Prozent der Fälle benötigt werden, kann die durchschnittliche Übertragungsmenge deutlich reduziert werden – die entfallenen Felder werden auf eigene Seiten ausgelagert. In vielen Webanwendungen existieren Formulare mit mehr als 30 Eingabefeldern, von denen in 90 Prozent der Anwendungsfälle immer nur zwei Felder gefüllt wurden – angezeigt und übertragen wurden aber immer alle Felder, inklusive aller Listen für die Auswahlboxen.

Ein weiteres, häufiges Anti-Pattern ist das „Phantom Logging“, das man in fast allen Projekten antrifft. Beim Phantom Logging werden Log-Nachrichten erzeugt, die im aktiven Log-Level eigentlich nicht erstellt werden müssen. Nachstehender Code ist ein Beispiel für das Problem:

```
logger.debug("Eine Log Nachricht" + param_1 + "Text" +
param_2);
```

Obwohl die Nachricht im INFO-Level nicht geloggt würde, wird der String zusammengebaut. Dies kann je nach Anzahl und Komplexität der Debug- und Trace-Meldungen zu enormen Performance-Einbußen führen – gerade wenn Objekte eine überschriebene und kostspielige *toString()*-Methode haben. Die Lösung ist einfach:

```
if (logger.isDebugEnabled())
logger.debug("Eine Log Nachricht" + param_1 +
"Text" + param_2);
```

In diesem Fall wird der Log-Level zunächst abgefragt und die Log-Nachricht nur dann erzeugt, wenn der DEBUG-Log-Level aktiv ist.

Um Performance-Engpässe bei der Entwicklung zu vermeiden, sollten vor allem die eingesetzten Frameworks richtig verstanden werden. Bei den meisten kommerziellen und Open-Source-Lösungen gibt es ausreichend Dokumentation zum Thema Performance – zudem sollten in regelmäßigen Abständen Experten bei der Implementierung der Lösung hinzugezogen werden. Selbst wenn bei einem Profiling das Bottleneck innerhalb eines Frameworks gefunden wird, bedeutet dies nicht, dass auch das Problem innerhalb des Frameworks liegt. In den meisten

Fällen besteht das Problem in der falschen Nutzung oder Konfiguration.

Fazit

Performance Anti-Patterns existieren nicht nur in der Entwicklung, sondern vor allem auch innerhalb der Projektorganisation und der Architektur. Die richtigen Prozesse und Tools [14], sowie das notwendige Expertenwissen sind die Basis, um die beschriebenen Anti-Patterns zu vermeiden und eine gute Performance und Stabilität im eigenen Projekt zu erreichen. In Zukunft planen die Autoren, die Anti-Patterns in den verschiedenen Kategorien auf ihrer Webseite zu veröffentlichen und eine Community rund um Performance- und Stabilitäts-Anti-Pattern zu etablieren, um so möglichst viele Probleme aus bestehenden Projekten in Zukunft vermeiden zu können.



Mirko Novakovic ist Geschäftsführer der codecentric GmbH und Spezialist im Bereich Performance Tuning, Java EE-Architekturen und Open-Source-Frameworks. Kontakt: novakovic@codecentric.de.



Nick Panienski arbeitet als Consultant in Open-Source-basierenden Java EE-Projekten und ist seit Gründung der codecentric GmbH für deren Kunden im Einsatz. Kontakt: panienski@codecentric.de.

Links & Literatur

- [1] Christopher Alexander: A Pattern Language. Oxford University Press, 1977.
- [2] Erich Gamma: Design Patterns. Addison-Wesley Longman, 1995.
- [3] Sun Java Blueprints: java.sun.com/reference/blueprints
- [4] Floyd Marinescu: EJB Design Patterns. Wiley & Sons, 2002.
- [5] Gregor Hohpe: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman, Oktober 2003.
- [6] Bruce Tate: Bitter Java. Manning, Mai 2002.
- [7] Pet Store J2EE vs. NET: www.onjava.com/pub/a/onjava/2001/11/28/catfight.html
- [8] JPetstore: sourceforge.net/projects/ibatisjpetstore
- [9] iBatis Data Mapper Framework: ibatis.apache.org
- [10] Rod Johnson: J2EE Development Without EJB. Wiley & Sons, 2004.
- [11] Glenford Myers: The Art of Software Testing. Wiley & Sons, 2004)
- [12] Java theory and practise series: www-128.ibm.com/developerworks/views/java/libraryview.jsp?search_by=practice
- [13] JBoss Seam: labs.jboss.com/jbossseam
- [14] Niklas Schlimm: Performance-Analyse und -Optimierung in der Softwareentwicklung. Informatik Spektrum, Volume 30, 04.2007