S&N INVENT

Value Driven Performance.

LEISTUNG BLEIBT WICHTIG

Immer schneller - aber wie?

TEIL 2

Dr. Jürgen Lampe
S&N Invent GmbH

Abstract

Dies ist der zweite Teil eines Beitrags zur Beleuchtung der Konsequenzen, die sich für die Softwareentwicklung daraus ergeben, dass die Steigerung der skalaren Prozessorgeschwindigkeit seit Jahren nahezu stagniert. Im ersten Teil wurden allgemeine Fragen gestellt und diskutiert. Dieser zweite Teil befasst sich mit Konsequenzen für die Java-Entwicklung.

Ohne Zweifel werden die Leistungsanforderungen an die Software weiter steigen. Im vorangegangen Teil wurde gezeigt, dass die Quellen quasi automatischer Performance-Steigerungen nahezu ausgeschöpft sind und zukünftig ein zielgerichtetes *Design for Performance* als Teil der Softwareentwicklung zunehmend an Bedeutung gewinnen wird. Im Folgenden werden einige Grundsätze für die leistungsorientierte Softwareentwicklung diskutiert.

Um keinen falschen Eindruck aufkommen zu lassen, sei nochmals betont: Leistung ist kein Selbstzweck. Sie hat ihren Preis. Der besteht vorrangig in höherem Entwicklungsaufwand und schwierigerer Wartbarkeit. Diese Nachteile müssen gegen die erreichbare Verbesserung abgewogen werden. Aller Wahrscheinlichkeit nach wird man in Zukunft immer öfter vor der Notwendigkeit stehen, diesen Preis zu akzeptieren, um die geforderten Leistungswerte überhaupt noch realisieren zu können.

Algorithmus

Leistungsfähige Software beginnt mit der Auswahl der richtigen Verfahren. Das ist nicht einfach und erfordert als typische Ingenieursarbeit viele Abwägungen und daraus resultierende Kompromisse. Nicht umsonst und völlig zu Recht spricht man bisweilen von Ingenieurs*kunst*. Die erste und wichtigste Aufgabe in dieser Phase besteht darin, die vorgelegten Anforderungen zu prüfen und zu verstehen. Um erkennen zu können, an welchen Punkten die höchsten Belastungen zu erwarten sind und welche Auswirkungen unterschiedliche Realisierungsvarianten auf die zu erwartenden Leistungsparameter haben werden, ist eine gute Durchdringung der Aufgabe zwingend erforderlich. Deshalb müssen bereits zu diesem Zeitpunkt möglichst zuverlässige Aussagen über das geforderte Nutzungsprofil vorliegen. Dazu gehören u. U. auch Charakteristika der zu behandelnden Daten, wenn diese Einfluss auf die Performance haben können (das ist bei hinreichender Menge eigentlich immer der Fall). Beispielsweise haben Konto- oder Artikelnummern eine ganz andere statistische Verteilung als etwa Namen oder beschreibende Bezeichnungen. Häufigkeiten sind auch deshalb wichtig, weil viele besonders effiziente Algorithmen einen hohen Initialisierungsaufwand haben, sodass ihre Vorteile erst bei einem größeren Aufgabenumfang sichtbar werden. So kann eine Liste mit linearer Suche bei sehr kleinem Suchraum durchaus schneller sein als eine Hashtabelle, insbesondere wenn die Hashwertberechnung aufwendig ist. Als Beispiel für den Einfluss von

Verteilungseigenschaften sei das *Quicksort*-Sortierverfahren [1] genannt. Dessen Effizienz hängt auch vom Anteil identischer Schlüsselwerte ab.

Wenn es um die zentralen Teile des Verfahrens und wirklich große Mengen geht, sollte man nicht vergessen, dass keine automatische Optimierung in der Lage ist, die prinzipielle Komplexität eines Algorithmus etwa von $O(n^2)$ auf $O(\log n)$ zu ändern.

In der Praxis stößt man häufig auf die Situation, dass für den überwiegenden Teil der Verarbeitungsfälle ein relativ schnelles und einfaches Verfahren existiert. Der Aufwand, d. h. letztlich die Leistungseinbuße, wird durch den nicht in das Schema passenden Rest verursacht. Diese Erfahrung wird manchmal und nicht ganz korrekt *80/20*-Regel genannt. Wenn Leistung wirklich wichtig ist, sollte man in so einem Fall nicht davor zurückschrecken, von vornherein verschiedene Verarbeitungszweige vorzusehen. Auf den ersten Blick mag das wie eine unnötige Verkomplizierung erscheinen. In Wirklichkeit vereinfacht diese Trennung die Entwicklung. Die beiden Ziele Geschwindigkeit und vollständige Funktionalität können getrennt angegangen werden. Im Hauptzweig entsteht vorrangig performanter Code, im Nebenzweig vor allem gut verständlicher Code. Das funktioniert natürlich nur, wenn eine entsprechende Auftrennung leicht möglich ist. Der Weg, (zeit-)kritische Funktionen durch eigens darauf spezialisierte Funktionseinheiten ausführen zu lassen, ist weit verbreitet, wird bei Software aber noch viel zu selten bewusst eingesetzt (siehe Kasten „Leistung auf Extrawegen“).

Leistung auf Extrawegen

Beispiel für das Abtrennen bestimmter Funktionen, um diese mit höchster Effizienz ausführen zu können, sind nicht nur Autobahnen und „rote Telefone“. Die Evolution hat dieses Prinzip schon lange vorher „entdeckt“. So führen beim Menschen (im Unterschied zu anderen Primaten) direkte Nervenbahnen u. a. zu den Händen oder zum Kehlkopf, was uns spezifisch menschliche Leistungen wie Klavierspielen oder ausdifferenzierte Lautsprache ermöglicht. Dass die ursprünglichen Wege noch rudimentär vorhanden sind, zeigt sich, wenn in lebensbedrohenden Situationen plötzlich gelähmte Körperteile wieder bewegt werden können.

Ein wichtiger Nachteil dieser Technik sei nicht verschwiegen: Sie erschwert die Modularisierung. Viele Versuche Betriebssysteme aus konfigurierbaren Bausteinen aufzubauen sind daran gescheitert, dass man aus Performancegründen nicht auf solche (die modulare Struktur zerstörenden) *Shortcuts* verzichten kann.

Schließlich gehört zur Bewertung eines Algorithmus nicht zuletzt die Betrachtung seiner Implementierungskosten. Nur selten werden die Anforderungen so extrem sein, dass der Preis eine untergeordnete Rolle spielt. Günstiger zu implementierende Verfahren haben oft den Vorteil auf die Dauer auch besser wartbar zu sein. Deshalb sollte man ihnen den Vorzug geben, selbst wenn die zu erwartende Leistung etwas schlechter ist. Die so gewonnenen Mittel können dann für Verbesserungen in anderen Teilen des Systems verwendet werden – man sollte sie aber nicht ganz einsparen!

Implementierungsstrategie

Wenn die grundlegenden Design-Entscheidungen getroffen sind, muss eine passende Implementierungsstrategie entwickelt werden. Da stellt sich zuerst die Frage nach vorhandenen Bibliotheken und Frameworks. Prinzipiell ist der nachnutzende Einsatz („Reuse“) von Software positiv zu sehen. Die Vorteile sind höhere Stabilität und geringerer Aufwand. Ein interessanter Gewinn besteht möglicherweise darin, faktisch ohne eigenen Aufwand an zukünftigen Verbesserungen teilhaben zu können. Allerdings bleibt gerade diese Chance oft theoretisch. Insbesondere Open-Source-Produkte überraschen gern mit inkompatiblen Veränderungen von Schnittstellen. Überdies sollte kritisch geprüft werden, ob in der bisherigen Entwicklung Performance wirklich ein wesentliches Ziel war. Gar nicht selten stehen Funktionserweiterungen so im Vordergrund, dass Leistungsverbesserungen eher unbedeutend bleiben.

Ganz generell ist zu beachten, dass allgemeine Lösungen von der Tendenz her weniger effizient sind als maßgeschneiderte, weil sie naturgemäß auch Funktionen unterstützen oder enthalten, die im konkreten Anwendungsfall nicht benötigt werden.

Eine eigene Problematik stellen Frameworks dar. Wirkliche *Rahmenstrukturen* – wie die Übersetzung lautet – sind nur wenige. Ein großer Teil dessen, was unter diesem Namen firmiert, sind streng genommen (unvollständige) Baukästen, teilweise vom Anwender zu komplettieren, aus deren Fertigteilen die Lösung konstruiert werden muss. Die Crux mit Bausätzen ist, dass sie, um Flexibilität zu gewährleisten, notwendigerweise viele kleine Teile und damit entsprechend viele innere Schnittstellen („Glue“) enthalten. Das ist vergleichbar mit einem Roboter, den man mit Teilen aus einem Technikbaukasten montiert. Damit lassen sich durchaus anspruchsvolle Aufgaben lösen. (Es soll sogar Kleinserienlinien geben, wo so etwas eingesetzt wird.) Aber abgesehen von der Dauerbelastbarkeit (die bei Software keine Rolle spielt), sind diese Konstruktionen nie so leistungsfähig wie ein für den speziellen Zweck optimiertes Gerät.

Generische Software benutzt häufig Reflektion und auf Strings basierende assoziative Strukturen („Maps“). Beide Techniken haben eine relativ schlechte Performance. Wenn immer möglich, sollten codegenerierende Werkzeuge bevorzugt werden, weil generierter Code praktisch ebenso effizient sein kann (und dabei auch noch weniger Fehler enthält), wie von Hand geschriebener und überdies Wartung und Debugging erleichtert wird. Falls irgendwo ein Feld `XName` benötigt wird, ist der Zugriff mit `getXName()` nicht nur wesentlich schneller als mit `getField("XName")`, sondern er verhindert auch ein irrtümliches und erst zur Laufzeit als Fehler bemerkbares `getField("YName")`.

Ein häufig unterschätzter Aspekt für die Gesamtperformance ist die Verteilung der Aufgaben auf die einzelnen Teilsysteme. Alle Bemühungen um eine leistungsfähige Anwendung sind sinnlos, wenn diese im Betrieb ständig auf andere Systeme wartet. Deshalb müssen die Leistungscharkteristika

aller beteiligten Komponenten in die Planung einbezogen werden. Ein typisches Beispiel für solche Überlegungen ist die Entscheidung, welche Funktionen in der Datenbank und welche als Java-Programm implementiert werden sollen. Die Verlagerung von Logik in die Datenbank erschwert zwar unter Umständen die Lesbarkeit des Codes und damit die Wartung, kann jedoch deutliche Leistungsverbesserungen bringen. Andererseits sind auch Konstellationen vorstellbar, in denen gerade der umgekehrte Weg, nämlich die Datenbank zu entlasten, der bessere ist.
Wie immer gilt, dass es keine generell gültige Einheitslösung gibt. Man sollte sich bemühen, auf dem für die Leistung kritischen Pfad *effizient laufende* Verfahren einzusetzen. Abseits davon haben die *effizient zu implementierenden* Verfahren ihren Platz. Warum soll eine Anwendung mit hohem Datendurchsatz nicht direkt optimierten JDBC-Code verwenden, während die nur zu Beginn und Ende einer Session transferierten Nutzereinstellungen mit Hilfe eines ORM-Tools verwaltet werden? Trotz einiger Nachteile ist eine so gesplittete Anwendung insgesamt vorteilhafter, als wenn nachträglich versucht werden muss durch endloses Tuning und fehlerträchtige Eingriffe in die Standardbausteine den eingesetzten Baukasten auf die erforderliche Leistung zu trimmen.
Schließlich sei nicht vergessen, daran zu erinnern, dass alles Bemühen um effiziente Teilsysteme vergeblich ist, wenn diese Teile nicht gut aufeinander abgestimmt sind. Ein gutes System ist bekanntlich mehr als die Summe seiner Teile – ein schlechtes kann sehr viel weniger sein.

Datenstrukturen

Eine Frage, die bezüglich ihres Einflusses auf die Performance generell viel zu wenig beachtet wird, ist die Implementierung des Datenmodells. Wie die Daten durch Java-Typen dargestellt werden, hat erhebliche Auswirkungen. Die wichtigsten Faktoren sind dabei:

- Allokation und Initialisierung
- Verwaltungsoverhead
- Speicherbereinigung (Garbage-Collection)
- Einfluss auf die Cache-Trefferrate (Hit-Rate)

Im Laufe der Jahre ist es gelungen, die Kosten der Objekterzeugung deutlich zu senken. Außerdem liegt es größtenteils in der Hand des Entwicklers und ist damit relativ klar erkennbar, welchen Aufwand die Initialisierung verursacht. Weniger sichtbar ist, dass jedes Objekt einen Satz von Metadaten mit sich führt, die unter anderem vom Garbage-Collector (GC) gebraucht werden. Die Anzahl der dafür benötigten Bytes variiert zwischen den verschiedenen JVM-Implementationen, ist aber unabhängig von der Größe der Objekte. Das bedeutet, viele kleine Objekte belegen bei insgesamt gleicher Größe des von den Objekten unmittelbar genutzten Speichers mehr davon für die Metadaten als wenige große. Dadurch wird indirekt die Aktivität des GC erhöht, weil der Speicher schneller voll wird. Gleichzeitig steigen die Kosten pro GC-Lauf, weil jedes Objekt besucht wird.

Neben der Größe der Objekte hat deren Lebensdauer Auswirkungen auf die Kosten der Speicherverwaltung. Unkritisch sind Objekte, die sehr lange oder sehr kurz leben. Kostspielig wird es, wenn Objekte einige GC-Läufe überleben und bei Reorganisierungen des Heap (Kompaktieren) umkopiert werden müssen.

Bevor mögliche Einflüsse auf die Trefferrate beim Caching betrachtet werden, sei nochmals betont, dass diese Betrachtungen immer unter dem Vorbehalt der Angemessenheit stehen. Nur wenn es sich um große Anzahlen und sehr häufige Zugriffe handelt, sind diese zusätzlichen Überlegungen gerechtfertigt. Mit der Betrachtung des Cache-Verhaltens gelangt man an die Grenze zum Tuning, weil der Einfluss der ganz spezifischen Hardware-Konstellation überwiegt. Eine allgemeine Regel kann jedoch angegeben werden: Datenlokalität wirkt sich fast immer positiv auf die Geschwindigkeit aus, weil dadurch die Wahrscheinlichkeit steigt, dass die als nächste benötigten Daten bereits mit den vorherigen in eine Cache-Line geladen wurden. Besonders bei oft durchlaufenen Schleifen kann das beobachtet werden. Wenn die Daten dann noch in konstanten Abständen vorliegen, wie das bei Arrays der Fall ist, können die in der Hardware implementierten Prefetch- und Preload-Techniken ihr Potential voll entfalten.

Beispiel 1 soll das Besprochene illustrieren. Die Aufgabe besteht darin, für eine große Menge von Punkten mit ganzzahligen Koordinaten, den maximalen x-Wert zu ermitteln. Für die Datenstruktur werden drei Varianten betrachtet:

1. Ein Feld von Punkt-Objekten
2. Zwei Felder, die jeweils die x- und y-Werte enthalten
3. Ein Feld doppelter Länge, das abwechselnd x- und y-Werte enthält.

Ein beispielhafter Vergleich der Ausführungszeiten (ohne Hotspot-Optimierungen) ergab wie erwartet, dass die zweite Variante (0,6 ms) die schnellste ist, deutlich abgeschlagen die erste (3,1 ms). Der relevante Unterschied zwischen zweiter und dritter Variante (0,8 ms) ist hauptsächlich dadurch zu erklären, dass sich durch den verdoppelten Abstand zwischen den Werten die Cache-Hit-Rate halbiert. Der letztere Effekt verschwindet sofort, wenn statt nach dem Maximum einer Koordinate nach dem Maximum der Summe der beiden Werte gesucht wird. Dieses Beispiel illustriert damit einerseits den Einfluss des Caches, zeigt andererseits die Grenzen, wenn es um die gezielte Beeinflussung geht.

```
private static final int ANZAHL= 1000000;
//  Variante 1: Array von Punkten (java.awt.Point)
  private static Point[] points= new Point[ANZAHL];
  static int maxX1(Point[] points) {
   int result= Integer.MIN_VALUE;
   for (Point point : points) {
     if (point.x > result) {
            result= point.x;
     }
   }
   return result;
  }
// Variante 2: Zwei Arrays von x- und y-Werten
  private static int[] pointsX= new int[ANZAHL];
  private static int[] pointsY= new int[ANZAHL];
  static int maxX2(int[] coordX) {
   int result= Integer.MIN_VALUE;
   for (int i= 0; i < coordX.length; i++) {
     if (coordX[i] > result) {
          result= coordX[i];
     }
   }
   return result;
  }
// Variante 3: Ein Array, abwechselnd x- und y-Werte
  private static int[] pointsXY= new int[2*ANZAHL];
  static int maxX3(int[] coord) {
   int result= Integer.MIN_VALUE;
   for (int i= 0; i < coord.length; i+= 2) {
     if (coord[i] > result) {
          result= coord[i];
     }
   }
   return result;
  }
```

Beispiel 1: Datenstrukturen für zweidimensionale Punkte

Die Dauer des Schleifendurchlaufs ist allerdings nur der relativ leicht nachweisbare Effekt der verschiedenen Implementierungen. Wenn diese Datenstruktur sehr häufig angelegt und verarbeitet wird, weil es beispielsweise um die fortlaufende Auswertung von Messreihen geht, haben die unterschiedlichen Objektzahlen erfahrungsgemäß einen noch deutlicheren Einfluss auf die Geschwindigkeit. Der Overhead für die Metainformationen variiert zwischen den Plattformen, aber 16 Byte pro Objekt ist ein realistischer Mittelwert. Im Beispiel, wo es um zwei int-Werte (je vier Byte) geht, benötigt die Variante 1 mit den Point-Objekten rund zweimal mehr Speicherplatz. Kleine Objekte in großer Zahl sind überproportional teuer. Wenn weniger Initialisierungs- und Verwaltungsoperationen auszuführen sind und weniger Speicherplatz belegt wird, stehen Ressourcen wie Cache und Prozessorzyklen anderen Codeabschnitten zur Verfügung, die auf diese Weise indirekt ebenfalls beschleunigt werden.

Ganz ohne Nachteil ist allerdings auch das vorgeschlagene Vorgehen nicht. Zu große Objekte können die Speicherverwaltung wieder verteuern, weil durch sie die Wahrscheinlichkeit steigt, dass der GC Kompaktierungen vornehmen muss, um ausreichend zusammenhängenden Speicher für die Allokation bereit stellen zu können.

Programmierregeln

Angesichts der vielen teils diffizilen Wechselwirkungen zwischen den die Ausführungsgeschwindigkeit beeinflussenden Faktoren mag es vermessen erscheinen, nach Regeln zu suchen. Die Erfahrung zeigt aber, dass es doch einige Grundsätze gibt, die in vielen Fällen helfen, leistungsfähige Software zu schaffen. Natürlich gibt es immer die berühmte *Ausnahme von der Regel* und niemand soll davon befreit werden, sich selbst Gedanken über den Sinn oder Unsinn der Anwendung in einer konkreten Situation zu machen. Insofern sind die aufgeführten Regeln eher als Denkanstoß denn als streng zu befolgende Vorschriften zu sehen.

Eine Grundregel sollte immer sein, möglichst einfachen, übersichtlichen und damit leicht verständlichen Code zu schreiben. Zum einen übersetzt der Java-Compiler, mit geringfügigen Ausnahmen, auf die noch eingegangen wird, relativ direkt, sodass sich aus dem Quellcode der erforderliche Aufwand gut abschätzen lässt. Zum anderen kann verständlicher Code leichter und gefahrloser optimiert werden, wenn sich später die Notwendigkeit dazu ergibt.

Es ist nicht überraschend, dass es eine größere Schnittmenge zwischen den Regeln für schnellen und denen für sauberen („clean“) Code gibt (allerdings behandelt *Clean Code* das Thema Performance leider etwas zu stiefmütterlich) [2] Insbesondere das unter dem Kürzel „YAGNI – you ain’t gonna need it“ bekannte Prinzip sollte beherzigt werden. Das heißt, den Code auf das Notwendige beschränken. Was nicht vorhanden ist, verbraucht keine Ressourcen und muss auch nicht optimiert werden.

Flexibilität verursacht Aufwand. Das weiß jeder. Trotzdem ist die Versuchung groß, Entscheidungen aufzuschieben: *Das machen wir konfigurierbar.* Dem liegt meistens ein (noch) unvollständiges Verständnis der Anwendungsdomäne zu Grunde oder eine falsche Interpretation der Lean-Software-Entwicklungsregel, Entscheidungen möglichst spät zu treffen. [3] Tatsächlich werden solche *Pseudovarianten* oft nicht gebraucht und vor allem werden sie nur selten so gründlich getestet, dass eine Umkonfiguration im laufenden Betrieb gefahrlos möglich wäre. Was bleibt, ist dann zusätzlicher – letztlich überflüssiger – Code, der im schlimmsten Fall sonst mögliche Optimierungen verhindert.

Ebenso sinnvoll ist es, den Grundsatz „Don't be too cute!" zu befolgen. Hier heißt das vor allem, auf Tricks und Kniffe zur vermeintlichen Leistungssteigerung zu verzichten. Die JVM sind wirklich gut darin, viele häufig vorkommende Funktionen zu beschleunigen. Aber dabei stützen sie sich auf verbreitete Codemuster. Besonders „clevere" Konstruktionen gehören eher nicht dazu und können somit letztlich zu langsamerer Ausführung führen.

Eine wichtige Quelle für Leistungsengpässe sind Zeichenketten (Strings). Java hat die Verwendung für Programmierer sehr einfach gemacht. Das verleitet jedoch zu leichtfertigem Umgang, weil der dahinterstehende Aufwand nicht sichtbar ist. Das grundlegende und nicht zu beseitigende Dilemma mit Zeichenketten ist, dass sie gewöhnlich aus vielen Zeichen bestehen und damit mehr Bytes benötigen als der Prozessor in einem Zyklus verarbeiten oder über den Speicherbus transportieren kann. Insbesondere letzteres ist für die Performance kritisch, denn die Busoperationen sind deutlich langsamer. In der JVM sind einige Heuristiken implementiert, um diesbezüglichen Aufwand zu reduzieren. Wirklich lösen kann man dieses Problem aber nicht.

Es gibt weitere Strukturen, mit denen (bislang) kein Computer wirklich effizient umgehen kann. Dazu gehören assoziative Collections, deren bekanntester Vertreter in Java die *Map* ist. Solche Abbildungen ordnen einem Wert (Schlüssel, Argument) einen anderen Wert (Funktionswert) zu. Praktisch ist das nicht ohne eine Art von Berechnung realisierbar und im primitivsten Fall eine einfache Suche. In der Regel werden effizientere Strukturen wie Hashtabellen oder Bäume benutzt. Allen gemeinsam ist, dass kein unmittelbarer Zugriff möglich ist. Im Gegensatz dazu erlaubt ein Array oder eine Liste einen wesentlich schnelleren Zugriff, weil aus dem angegebenen Index sofort auf den Speicherplatz geschlossen werden kann.

Manchmal lassen sich Maps mit Hilfe einer Technik ersetzen, die an die Normalisierung in relationalen Datenbanken erinnert. Die Schlüssel werden dabei in einer Liste registriert und im Weiteren durch ihren Index ersetzt. Um das Prinzip zu veranschaulichen, wird ein Beispiel genommen, das sich an die in Java übliche Implementation von Mehrsprachigkeit anlehnt (Beispiel 2). Statt die Schlüssel-Strings direkt als Parameter zu verwenden, werden diese durch einfache int-Werte ersetzt. Die Zuordnung, d. h. die Ermittlung der Indexwerte, übernimmt eine Liste („Key-Registry"), die jeden Schlüssel genau einmal enthält und beim Laden der jeweiligen Klasse abgefragt

wird. Bei allen Verwendungen kann dann, wenn die angenommene *FastMessages*-Klasse existiert, ohne irgendein Suchverfahren direkt auf die Werte zugegriffen werden. Voraussetzung für dieses Verfahren ist lediglich, dass die Schlüssel statisch sind. Beim Einlesen der Werte, z. B. aus einer Properties-Datei, wird die gleiche Registry benutzt. Abhängig von den konkreten Umständen erfordert die Verwaltung etwas Sorgfalt im Umgang mit Indexwerten, für die keine Abbildung definiert ist. Um beispielsweise Werte für nicht vorhandene Schlüssel durch `'!' + key + '!'` ersetzen zu können, muss die Registry um eine Methode zur inversen Zuordnung ergänzt werden.

```
    // Key-Registry
    static List<String> keyList= new ArrayList<String>();

    public static int registerKey(String key) {
           int result= keyList.indexOf(key);
           if (result < 0) {
                  result= keyList.size();
                  keyList.add(key);
           }
           return result;
    }
// ----------------------------------------------------
// Anwendungsfragment
    private static final String KEY1= "example.key1";
    private static final int IDX_OF_KEY1= registerKey(KEY1);
 …
// Übliche Verwendung mit String-Parameter:
    txt= Messages.getString(KEY1);
// Schnelle Alternative mit int-Parameter:
    txt= FastMessages.get(IDX_OF_KEY1);
// oder evtl. sogar nur (ggf. auf Exception achten):
    txt= currentMessages[IDX_OF_KEY1];
}
```

Beispiel 2: Mapping mit registrierten Schlüsseln

Ganz nebenbei sieht man an diesem Beispiel, wie das Bemühen um Effizienz die Lesbarkeit des Codes durch die zusätzliche Indirektion verschlechtern kann. Wenn die Namen sorgfältig gewählt werden, ist das hier gerade noch vertretbar.

Don’t Repeat Yourself

Das ist eine weitere Regel aus dem Clean-Code-Portfolio. Dort zielt sie allerdings rein auf das Vermeiden von Code-Dopplungen. Wenn hohe Leistung erreicht werden soll, muss das auf die Ausführung erweitert werden. Nicht nur beim Programmieren gilt: Doppelte Arbeit vermeiden! Das erscheint zunächst trivial, aber in der Praxis wird dieser Grundsatz häufig verletzt. In gewisser Weise werden solche Verletzungen durch die Objektorientierung sogar gefördert, weil einzelne Aspekte des Zustands eines Objekts über (möglicherweise parametrisierte) Methodenaufrufe erfragt und häufig erst dabei berechnet werden. Bei jeder Zustandsänderung alle möglichen Werte neu zu berechnen und dann vorzuhalten (zu *cachen*), ist andererseits auch keine gute Lösung. Im Hinblick auf die Leistung, muss man mit diesem Widerspruch angemessen umgehen. Im Folgenden soll das an Hand einer öfter anzutreffenden Situation erläutert werden.

Es ist eine übliche Praxis, Eingaben vor ihrer eigentlichen Verarbeitung zunächst auf Plausibilität zu prüfen und dann zu validieren. Im Sinne eines sauberen Systemaufbaus werden die drei Schritte Plausibilisierung, Validierung und Verarbeitung klar getrennt und nur die ursprünglichen Eingabedaten werden durchgereicht, bzw. im Fehlerfall ausgesteuert. Gewöhnlich liegen die Daten als Strings vor. Wenn ein Feld jetzt beispielsweise ein Datum (oder Datum und Uhrzeit) enthält, trifft man häufig auf Code, in dem dreimal die Umwandlung in `java.util.Date` zu finden ist. Beim ersten Mal, wenn es um die Kontrolle des Formats geht, interessiert nur ein eventuell auftretender Fehler. Bei der Validierung werden vielleicht weitere Prüfungen auf zulässige Werte vorgenommen und erst beim dritten Mal wird der Wert wirklich benutzt. Konvertierungen sind relativ aufwendige Operationen, insbesondere, aber nicht nur, wenn es um Datumswerte geht. Eine Lösung in diesem Fall könnte darin bestehen, die eingegebenen Daten in ein passendes Objekt zu packen, das außerdem Felder für die konvertierten Datumswerte enthält. Es ist grundsätzlich eine gute Idee, Datenformate und daraus resultierende Konvertierungen nicht nur als Implementierungsdetail zu behandeln. Eine klare Struktur mit möglichst wenigen Umwandlungen verbessert nicht nur die Leistung, sondern auch die Übersichtlichkeit des Codes.

Fazit

Design for Performance wird in Zukunft immer wichtiger werden. Weil für die Leistungssteigerung kaum noch leicht begehbare Wege offen stehen, hat man es zunehmend mit IT-Systemen zu tun, die auf ganz verschiedenen Ebenen zu optimieren versuchen. In ihrer Gesamtheit führt die wechselseitige Beeinflussung der eingesetzten Verfahren zu einem Verhalten, dass nahezu nichtdeterministisch erscheint. In einer solchen Umgebung ist es aussichtslos, einfache Regeln für schnellere Programme formulieren zu wollen. Deshalb sollten diese Bemerkungen eher als

Denkanstöße denn als einfache Handlungsanweisung verstanden werden. Im konkreten Fall bleibt nichts übrig als zu experimentieren, Erfahrungen zu gewinnen – und diese auch weiterzugeben! – und immer wieder zu messen.

Über S&N Invent GmbH

Die S&N Invent GmbH bündelt die Leistungen und das Angebot der beiden Gruppengesellschaften S&N AG (Paderborn) und A:gon Solutions GmbH (Eschborn). Gemeinsam sind sie unter dem Dach der S&N Invent GmbH noch leistungsfähiger und damit für Kunden und Partner attraktiver – dank einem gemeinsamen und damit wesentlich erweiterten Portfolio, verbesserter überregionaler Präsenz, einer deutlichen personellen Verstärkung und sich hervorragend ergänzender Kompetenzen.

Sowohl die A:gon Solutions GmbH als auch die S&N AG sind langjährig erfolgreiche und etablierte IT-Unternehmen. Insgesamt beschäftigt die S&N Invent ca. 350 feste und freie Mitarbeiterinnen und Mitarbeiter an sieben Standorten in Deutschland. So ist sie hervorragend aufgestellt, um die anstehenden Herausforderungen bei der IT-gestützten Optimierung von Geschäftsprozessen, in der Implementierung neuer und der Renovierung bestehender Softwarelösungen sowie in allen Bereichen der IT-Infrastruktur zusammen mit ihren Kunden und Partnern erfolgreich bewältigen zu können.

Autor

Jürgen Lampe ist IT-Berater bei der A:gon Solutions GmbH, einem Unternehmen im Verbund der S&N Invent GmbH in Frankfurt. Seit mehr als 15 Jahren befasst er sich mit Design und Implementierung von Java-Anwendungen im Bankenumfeld. Sein Interesse gilt besonders Fachsprachen (DSL) und Werkzeugen für deren Implementierung.

Links & Literatur

[1] https://de.wikipedia.org/wiki/Quicksort
[2] Wintersteiger, A.; Mathis, Ch.: „Clean Code“ Entwickler Magazin 5.2011
[3] https://en.wikipedia.org/wiki/Lean_software_development#Decide_as_late_as_possible