Optimierungen

Notwendigkeit

häufige Situation: Programm funktioniert im Prinzip fehlerfrei, aber nicht mit gewünschter Performanz

Symptome des Mangels an Performanz:

- `OutOfMemoryError` wird geworfen
- kein flüssiger Programmablauf
- keine Termination bei größeren Eingaben
- ...

Bedarf an Optimierung besteht aber im Grunde immer (z.B. Testumgebung ungleich der Umgebung, in der Software eingesetzt wird)

Responsability

nicht nur wichtig, dass ein Programm i.A. schnell ist

bei Interaktion mit Benutzer auch psychologischer Effekt nicht zu unterschätzen

Programm, dass während langer Berechnung *irgendetwas* sichtbar macht, wird normalerweise als besser wahrgenommen

Bsp.: Fortschritt anzeigen, bewegende Bilder, ...

ebenfalls wichtig aus diesem Grund: Programm soll auf Eingaben *sofort* reagieren

Bsp.: längere Suche in einer Datenbank

Responsability kann z.B. erreicht werden durch Trennung von Threads für Interaktion und für rechenintensive Teile

Vorsicht

lauffähige Software zu optimieren ist prinzipiell sinnvoll, muss aber mit Vorsicht durchgeführt werden

Optimierungen bzgl. Laufzeitverhalten können auch nachteilige Effekte haben, gehen z.B. zu Lasten der Lesbarkeit des Codes / der Korrektheit / etc.

$\rightsquigarrow$ keine Optimierungen ohne Grund und erkennbaren Effekt durchziehen!

> *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.* – William Wulf

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.* – Donald Knuth

Optimieren und Verschlechtern

eine Optimierung kann mehrere Aspekte des Programms betreffen, z.B.

- Laufzeitverhalten
  - Geschwindigkeit
  - Platzverbrauch
- Sourcecode
  - Les- und Wartbarkeit
- Korrektheit
- Dokumentation
- Machbarkeit
- ...

Änderung ist oft Optimierung nur bzgl. eines Aspektes, aber Verschlechterung bzgl. anderer

Beispiele

- Zeit vs. Platz vs. Machbarkeit

  Repräsentation einer *Menge* als

  1. `boolean[]` mit Zugriff in $\mathcal{O}(1)$, Platz $\mathcal{O}(n)$
  2. Suchbaum mit Zugriff in $\mathcal{O}(m)$, Platz $\mathcal{O}(m)$

  (m = Anzahl eingefügter Elemente, n = Anzahl möglicher Elemente)

  beachte: $n \geq m$ und $n = \infty, m < \infty$ sogar möglich oder Grundmenge unbekannt

- Platz vs. Lesbarkeit

  Array der Größe n mit intuitiven Indizes $1, \ldots, n$ der Elemente

- Geschwindigkeit vs. Lesbarkeit

  vorzeitiger Abbruch einer Schleife; `while` statt `for`

Optimierungen durchführen

vor Änderung am Sourcecode muss abgewägt werden:

- ist Änderung wirklich eine Optimierung im Ganzen?
  - welche Aspekte werden optimiert?
  - welche anderen sind betroffen?
- Aufwand
  - wieviel Arbeit erfordert die Optimierung?
  - um wieviel wird das Programm dadurch verbessert?
- Design
  - steht geänderter Code z.B. im Widerspruch zu üblichem objekt-orientiertem Design?

danach/dabei muss z.B.

- Dokumentation angepasst werden
- Korrektheit neu überprüft werden

Optimierungspotential finden

bei nicht-offensichtlichem Mangel an Performanz:

- sinnvolle Wahl von Datenstrukturen

  welche Operationen werden häufig in dem Programm ausgeführt und welche Datenstrukturen sind bekanntermaßen am besten dafür?

- Einsatz von guten Algorithmen

  welches abstrakte Problem steckt hinter der Anwendung und welcher (bekannte) Algorithmus ist dafür der beste?

  Bsp.: Erreichbarkeit aller freien Felder auf Spielbrett kann als Nicht-Erreichbarkeit in ungerichteten Graphen aufgefasst werden $\rightsquigarrow$ Breitensuche

- Profiling

- Vermeidung von Java-Operationen, die als teuer bekannt sind

Profiling

Tabellieren des Ressourcenverbrauchs einzelner Programmteile bei einem Programmdurchlauf (vgl. Problematik beim Testen)

verschiedene Tools zum Profiling von Java-Programmen erhältlich; Eclipse und NetBeans z.B. haben auch integrierten Profiler; hier keine Vorstellung eines bestimmten

typische nützliche Information, die der Profiler zurückgibt:

- Häufigkeit und Dauer einzelner Methodenaufrufe (beachte Verschachtelungen!)
- Anzahl und Größe angelegter Objekte einer Klasse
- Anzahl von Objekten, die vom Garbage Collector eingesammelt wurden

anhand dessen Entscheidungen bzgl. Optimieren oder auch Nicht-Optimieren treffen

Teure Operationen in Java

- Anlegen von Objekten
- Exceptions werfen
- viele Klassen benutzen
- geboxte Datentypen (`Boolean`, `Integer`, etc.)
- manche Operationen auf Strings, insbesondere Konkatenation
- garbage collection
- type cast
- synchronisierte Methoden
- ...

Tips zur Programmbeschleunigung

- besonderes Augenmerk auf Schleifen

```
public static int[] countArr = new int[1];
private static Vector collection = new Vector(10000);
private static int points = 3;

// 1. Versuch
for(long i=0; i<collection.size()*50000; i++) {
  countArr[0]=countArr[0] + 5 * points;
}

// 2. Versuch
int count=countArr[0];
int addpoints=5*points;
int i=collection.size()*50000;
for(; --i>=0; ){
  count+=addpoints;
}
countArr[0]=count;
```

Performanzgewinn: 53894ms $\rightsquigarrow$ 846ms

Tips zur Programmbeschleunigung

- Datentyp `int` erlaubt schnelleren Zugriff als `long`, `byte`, `short` und als Objekte sowieso
- lokale Variablen verwenden
- Schleifen nicht durch Exceptions terminieren
- Iterationen statt Rekursionen
- Objektpools statt Neuerzeugung (siehe auch Factory Pattern)
- ...

Hinweise

- Geschwindigkeit moderner Rechner nicht unterschätzen!

  Platzoptimierung häufig wichtiger als Zeitoptimierung

- moderne Compiler nicht unterschätzen!

  viele Optimierungen (im Kontrollfluss) werden von Compilern automatisch durchgeführt

- Änderungen im Rahmen von Optimierungen können Korrektheitsannahmen wie z.B. Invarianten verletzen; Überprüfung / Umformulierung / neue Tests nötig

- `http://www.javaperformancetuning.com/`