JIVE
(Visualizing Java in Action)

Der Autor

Steven P. Reiss
Department of Computer Science
Brown University, Providence

Ziele

- dynamische Visualisierung
- minimale Kosten
- maximale Information
- „real time“
- möglichst alle Informationen gleichzeitig anzeigen
- Darstellung sollen möglichst wenig Raum einnehmen
- soll keine spezielle JVM benötigen (wie z.B. Jinsight von IBM)
- soll keine reine Performance Visualisierung sein (wie z.B. IBM's PV)
- „the result system had to be entertaining“

Daten Auswahl

- wie viel Zeit eines Intervalls wurde wo benötigt ?
- Zeitaufwand für Synchronisation?
- wann, wo, wie viel Speicher dynamisch reserviert (und freigegeben)?
- wann werden Threads erzeugt oder beendet?
- welchen Zustand haben die Threads: ausführend, blockierend, I/O, wartend, schlafend, synchronisierend
- wie oft blockiert ein Thread einen anderen?

...und Librarys?

Bibliotheken sind in der Regel:

- zahlreich
- zuverlässig und ausgetestet
- nicht vom Entwickler zu modifizieren

=> genaue Aufschlüsselung des Verlaufs innerhalb meist überflüssig und zu platzaufwendig (compact display)
=> wünschenswert ist eine mögliche Gruppierung des visualisierten Codes nach Package oder Package-Gruppen.

Ausführung vs. Darstellung

Javaprogramm:

kann viele verschiedene Funktionen in kurzer Zeit durchlaufen

Humaninterface („Auge“):

zu langsam um Änderungen in dieser Geschwindigkeit zu erkennen

Darstellung der Daten als Statistik nötig

Daten aufzeichnen, Möglichkeiten

1) mit JVMPI bzw JVMDI

- bereits in Java integriert
- keine Möglichkeit den Code zu Gruppieren
- sehr langsam

2) die Aufrufe von Methoden mit Stubs kapseln.

- Es müssen für alle Methoden Stubs erzeugt werden

Daten aufzeichnen, Möglichkeiten

3) Methoden so modifizieren, das relevante Stellen von speziellen Methodenaufrufen „umrahmt“ werden. Beispiel:

```
void myMethod(){
    visualiser.notifyMethodEntry();
    ...
    visualiser.notifyMethodExit();
}
```

- jede aufgerufene Methode muss modifiziert werden
- sehr flexibel

Daten aufzeichnen, Probleme

Nativ-Code für eingefügte Methoden verwende

+ Nativ-Code schneller

- Aufruf von Nativ-Code aus Java SEHR langsam

Java für eingefügte Methoden verwende

+ Java-Code Betriebsystem unabhängig

- jive läuft trotzdem nicht unter Windows....

Daten aufzeichnen, Probleme

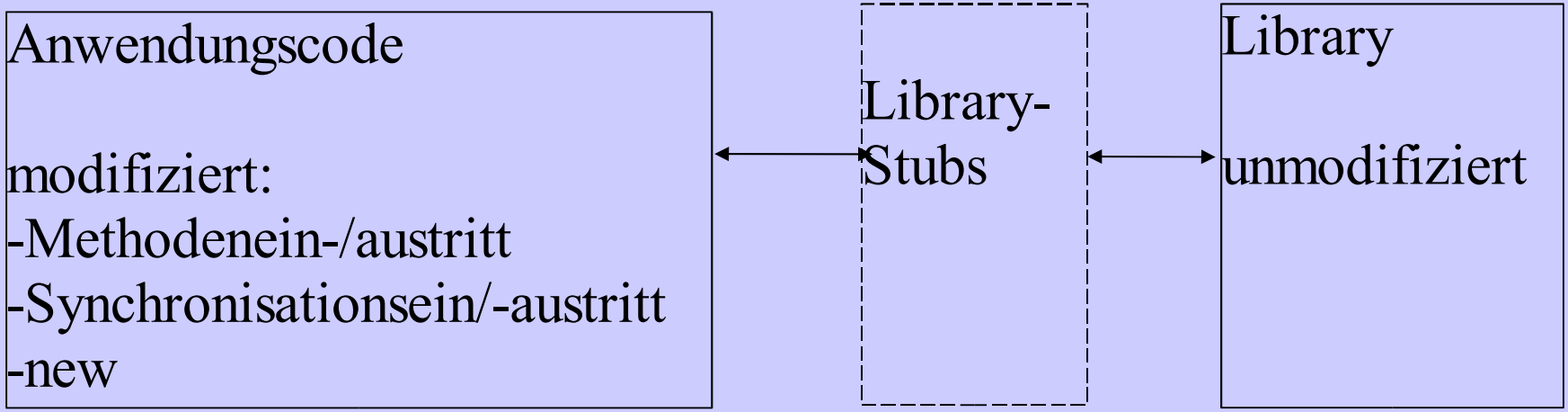
Thread.currentThread zu langsam

-> kann nicht an allen relevanten Stellen aufgerufen werden, nur aufrufen wenn sich der Thread-Zustand ändern kann, also alle Methoden in denen I/O, wait, ect. vorkommt.

-> Position der Threads im Code nicht genau bestimmbar

-> es können nur statistische Werte aufgezeichnet werden

Daten aufzeichnen, Umsetzung



Daten aufzeichnen, Umsetzung

Ausführungsintervall

monitoring thread

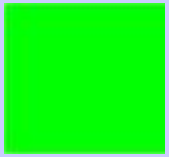
- das Programm ausführen
- durch eingefügte Methoden Daten sammeln

- gesammelte Daten zu einer Statistik zusammenfassen und an Visualisierung senden (via Socket oder Message-Server)
- Datenvariabeln reinitialisieren

Daten aufzeichnen, Ergebnis


```
<STATS TIME=’1035559445758’>
 <ENTRY NAME=’ja va’ COUNT=’1101551’ />
 <ENTRY NAME=’spr.onsets.OnsetExprSet’ COUNT=’159197’ ABY=’95171’ />
 <ENTRY NAME=’spr.onsets.OnsetCubeSet’ COUNT=’225’ AOF=’225’ />
 <ENTRY NAME=’spr.onsets.OnsetTypeSet’ COUNT=’94496’ AOF=’94496’ />
 <ENTRY NAME=’spr.onsets.OnsetExprSet$SetExpr’ COUNT=’225’ AOF=’225’ />
 <ENTRY NAME=’spr.onsets.OnsetCardSet’ COUNT=’1509’ AOF=’1734’ />
 <ENTRY NAME=’spr.onsets.OnsetCubeBase’ COUNT=’1729410’ />
 <ENTRY NAME=’spr.onsets.OnsetBitSet’ COUNT=’4577700’ />
 <ENTRY NAME=’spr.onsets.OnsetCardDeck’ COUNT=’1059’ ABY=’1059’ />
 <ENTRY NAME=’spr.onsets.OnsetCubeDeck’ COUNT=’2929392’ />
 <ENTRY NAME=’spr.onsets.OnsetExprSet$Expr’ COUNT=’225’ ABY=’450’ />
 <TOTALS COUNT=’10594989’ AOF=’96680’ ABY=’96680’ />
 <THREAD INDEX=’1’ NAME=’main’ SYNC=’1016’ />
 <THREAD INDEX=’2’ NAME=’Reference Handler’ WAIT=’1016’ />
</STATS>
```

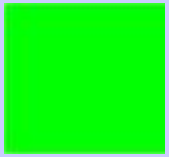
Box Display Visualization



Ein Rechteck hat:

- Höhe
- Breite
- Farbschattierung
- Farbsättigung
- Farbhelligkeit
- Textur

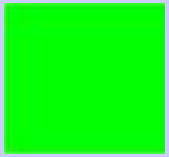
Box Display Visualization



Ein Rechteck hat:
- Höhe
- Breite
- Farbschattierung
- Farbsättigung
- Farbhelligkeit
- Textur

1) Darstellung einer Klasse
=Anzahl der Methodenaufrufe
=Anzahl der Speicheranforderungen
=Anzahl der Objekte dieser Klasse
=im Interval benutzt ja/nein
=Anzahl der Synchonisationen

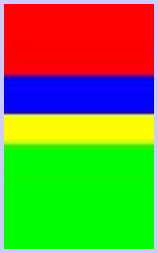
Box Display Visualization

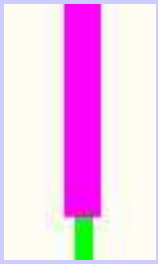


Ein Rechteck hat:

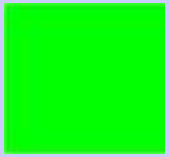
- Höhe
- Breite
- Farbschattierung
- Farbsättigung
- Farbhelligkeit
- Textur

2) Darstellung eines Threads:
Stapel mehrerer Rechtecke,
jedes repräsentiert genau einen
Threadstatus (ausführend, blockierend,
I/O, wartent, schlafend, synchronisierend)





Box Display Visualization

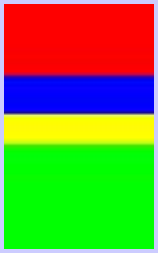


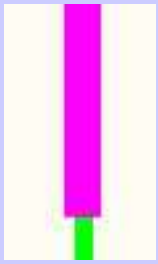
Ein Rechteck hat:

- Höhe
- Breite
- Farbschattierung
- Farbsättigung
- Farbhelligkeit
- Textur

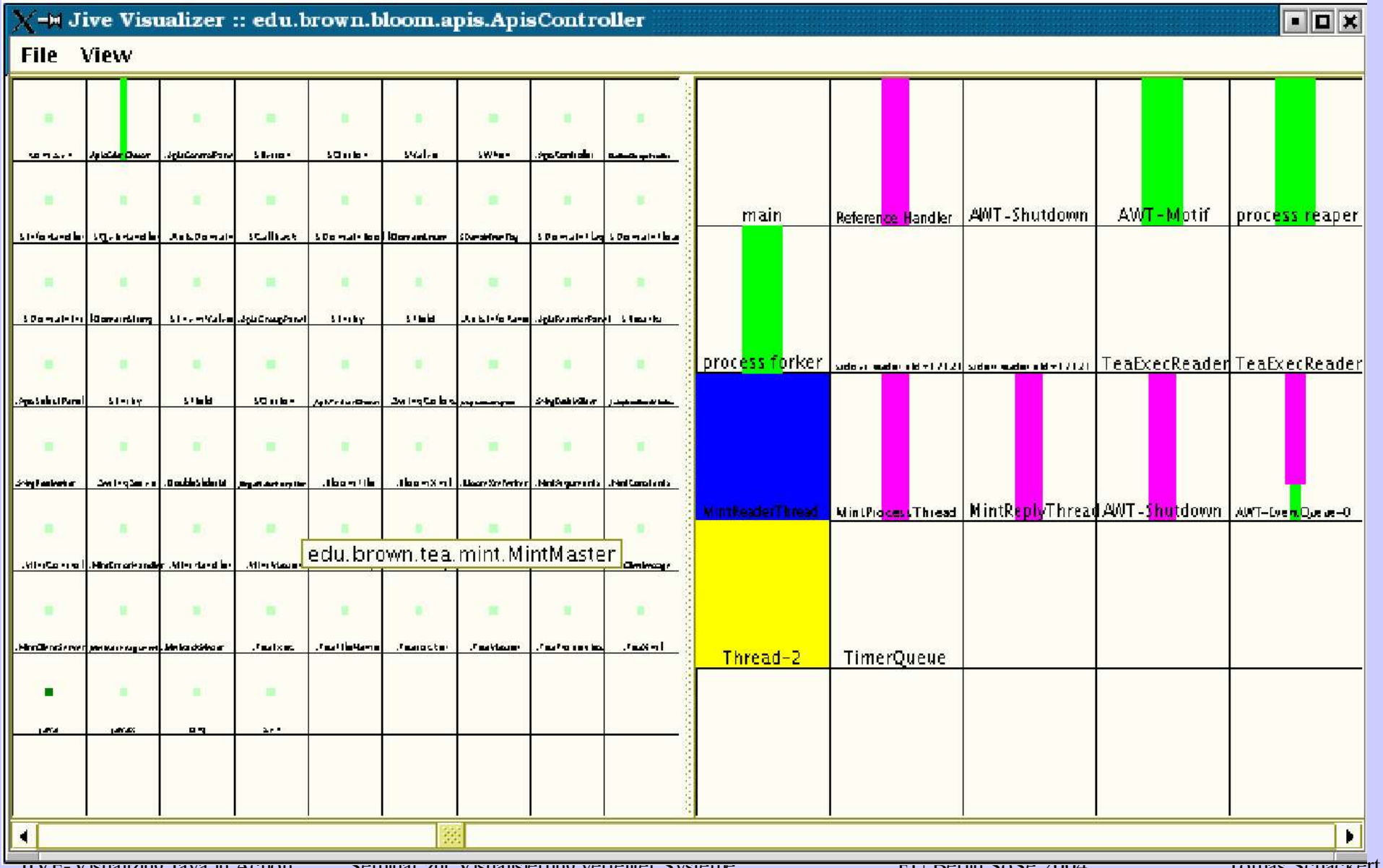
3) Darstellung eines Threadsstatus:
=Dauer des Threads in diesem Status
=Anteil an allen Threads in diese Status
=Art des Status

=Anzahl der durch ihn Blockierten Threads

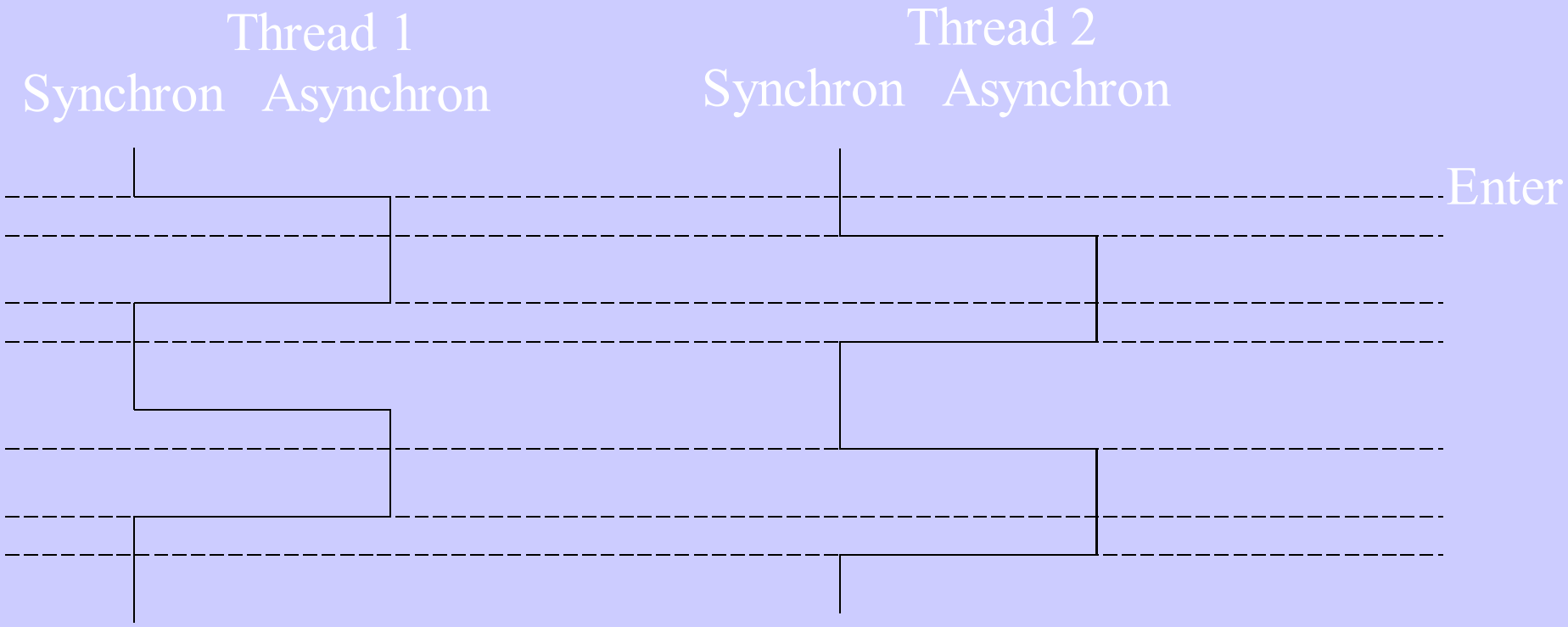




Box Display Visualization



Demo



Threadzustände:

- **ausführen (async)**
- **Synchron**
- **Wait**
- **Block**

Pro&Contra

- verlangsamt das Programm nur um das 2-3 fache
- benötigt vor jedem Lauf ca. 30 Sekunden um das Programm zu modifizieren
- Autoren bewerten die Total-Statistik als sehr gelungen
- Probleme mit Thread-Status-Wechsel wenn im nicht modifizierten Codeteil oder in Nativ-Code (JVM)
- nicht alle Schritte im Programm sind gleich stark von Modifikation belastet, daher ist die Darstellung etwas verzehrt
- keine exakten Werte aufgezeichnet: welcher Thread blockt welchen anderen, wie lang braucht etwas (nur Intervalle)
- Durchlauf kann (laut Autor) nicht gespeichert werden