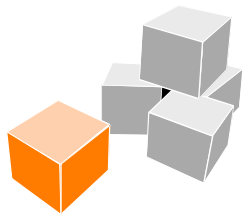




TECHNISCHE
UNIVERSITÄT
DARMSTADT



*Software
Technology
Group*

TU Darmstadt | FB Informatik

Dr.-Ing. Michael Haupt

in Zusammenarbeit mit
Dipl.-Inform. Christoph Bockisch
Stephan Ritzkowski
Prof. Dr.-Ing. Mira Mezini

Omniscient Debugging und Slicing für Java



Überblick

- das Fachgebiet Softwaretechnik
- Probleme beim Debugging
- "Debugging rückwärts" mit **Omniscient Debugging**
- statisches und dynamisches Slicing
- Fehlersuche mit **Omniscient Slicing**
- Zusammenfassung



Das Fachgebiet Softwaretechnik

"teile und herrsche" als Konstruktionsprinzip

modulare und entkoppelte Softwarearchitekturen
aspektorientierte und komponentenorientierte Softwareentwicklung
service-orientierte Softwarearchitekturen und Produktlinien
dynamisch anpassbare Software

**zwei komplementäre Wege zur
Beherrschung der Softwarekomplexität**



intelligente Konstruktionswerkzeuge

intelligentes Wissensmanagement und -Retrieval für Entwicklungsumgebungen
integrierte erweiterbare statische Analyseplattformen
Softwareexploration und -visualisierung für leichtere Wartung und Re-Engineering



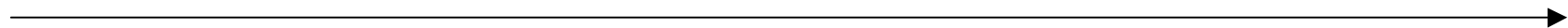
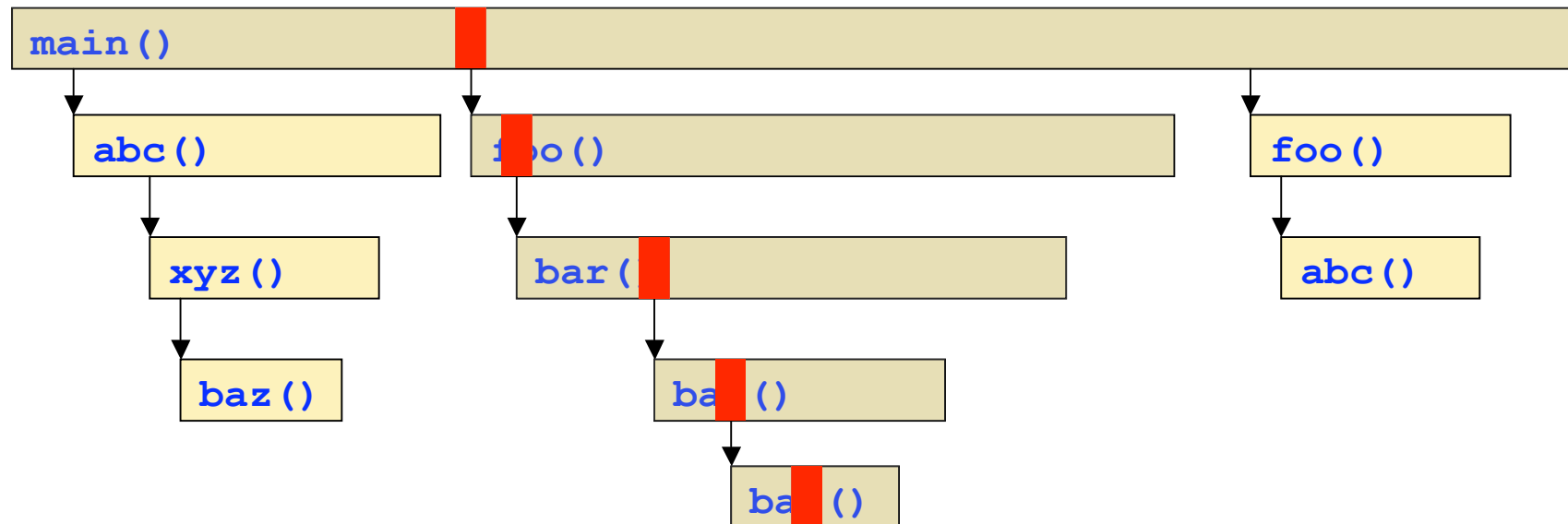
Aktuelle Aktivitäten

- neue Programmiersprachen
 - inklusive geeigneter Laufzeit- und Entwicklungsumgebungen
 - Schwerpunkt: [aspektororientierte Programmierung](#)
 - CaesarJ, Steamloom
- Anwendung der neuen Programmiersprachen
 - Software-Produktlinien
 - modulare Komponenten-Middleware
 - aspektororientierte Web-Service-Komposition
 - modellbasierte Entwicklung mobiler Anwendungen
- Entwicklungsumgebungen
 - als Plattformen für Software Knowledge/Information Engineering
 - Exploration, Visualisierung, Analyse



"Traditionelles" Debugging

- **Breakpoints:** Markierung potenziell interessanter Stellen
 - Ausführung der Anwendung bis zum Breakpoint
 - Untersuchung des Zustands, (ggf. schrittweise) Weiterausführung
- nicht (oder nur schwer) beantwortbare Fragen
 - "Woher kommt diese **null**-Referenz?"
 - "Welche Methode wurde **vor dieser** aufgerufen?"
 - "Wie wirkt sich diese Zuweisung **im weiteren Verlauf** aus?"





Omniscient Debugging

- "debugging backwards in time"
 - vorgestellt von Bil Lewis, 2002
 - erlaubt Navigation durch komplette Ausführungs**vergangenheit**
 - Punkte in der Ausführung (samt Zuständen) werden aufgezeichnet

Bildrechte bei Bil Lewis

The screenshot displays the Omniscient Debugger interface for a Java application. The main window title is "Omniscient Debugger 27.July.02 - com.lambda.Debugger.Demo". The interface is divided into several panels:

- Threads:** Lists active threads including <main>, <Thread-4>, <Thread-5>, <Thread-3>, <Thread-7>, <Thread-6>, <Thread-8>, and <Thread-1>.
- Stack:** Shows the current stack frame: Demo.main(Object[0]_0), <Demo_0>.quick(0, 19), <Demo_0>.quick(11, 19), and <Demo_0>.average(11, 19).
- Locals:** Displays local variables: arg0 (11), arg1 (19), sum (0), and i (11).
- Code:** Shows the source code for the average method:

```
public int average(int start, int end) {
    int sum = 0;
    for (int i = start; i < end; i++) {
        sum += array[i];
    }
    return (sum/(end-start));
}
```
- TTY Output:** Shows the output of the program:

```
Starting QuickSort: 20
-- Done sorting --
-- 0 1 --
-- 1 0 --
-- 2 237 --
```
- Traces:** Provides a detailed execution trace, including method calls like <Demo_0>.quick(0, 19) -> void, <Demo_0>.average(0, 19) -> 994, <Thread-3>.start() -> void, <Demo_0>.quick(11, 19) -> void, <Demo_0>.average(11, 19) -> 1596, <Thread-4>.start() -> void, <Demo_0>.quick(16, 19) -> void, <Demo_0>.average(16, 19) -> 1885, <Demo_0>.quick(16, 17) -> void, and <Demo_0>.quick(18, 19) -> void.
- Objects:** Lists objects in memory, including an int array [20]_0 with values [19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0] and a <Demo_0> object with fields b ('=' (61)), c ('X' (88)), quick (<Demo_1>), and array (int[20]_0).

From last: 2 local = value



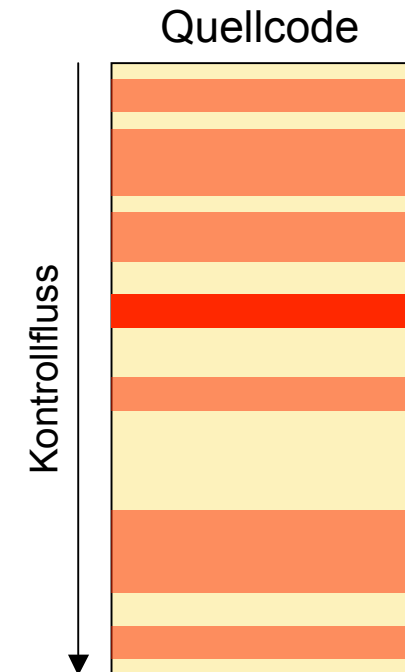
Omniscient Debugging

- "debugging backwards in time"
 - vorgestellt von Bil Lewis, 2002
 - erlaubt Navigation durch komplette Ausführungs**vergangenheit**
 - Punkte in der Ausführung (samt Zuständen) werden aufgezeichnet
- Navigation
 - ODB erlaubt **schrittweise** Suche nach Fehlerquellen
 - **erwünscht**: unmittelbare Übersicht aller möglichen Fehlerquellen
- Zukunft der Ausführung
 - **vorausschauendes** Debugging ist im ODB nicht möglich
 - **erwünscht**: Übersicht möglicher Einflüsse bestimmter Zustände



Slicing

- **Slice**: die Menge der Anweisungen,
 - die den Zustand an **dieser** Anweisung beeinflussen (backward slice)
 - deren Zustand **von dieser** Anweisung beeinflusst wird (forward slice)
- **statisches Slicing**
 - basiert auf statischer Programmanalyse
 - Slices enthalten **potenziell** beeinflussende / beeinflusste Anweisungen
 - Slices sind größer als nötig
- **dynamisches Slicing**
 - basiert auf einem konkreten (terminierten) Programmlauf
 - Slices enthalten **tatsächlich** beeinflussende / beeinflusste Anweisungen





Slicing

```
void foo(int x) {  
    if(x < 10)  
        System.out.println(„kleiner als 10“);  
    else  
        System.out.println(„größer als 10“);  
}  
  
void bar() {  
    int k = (int) (Math.random() * 100.0);  
    foo(k);  
}
```

statischer Forward Slice

```
void foo(int x) {  
    if(x < 10)  
        System.out.println(„kleiner als 10“);  
    else  
        System.out.println(„größer als 10“);  
}  
  
void bar() {  
    int k = (int) (Math.random() * 100.0);  
    foo(k);  
}
```

dynamischer Forward Slice (k=5)

```
void foo(int x) {  
    if(x < 10)  
        System.out.println(„kleiner als 10“);  
    else  
        System.out.println(„größer als 10“);  
}  
  
void bar() {  
    int k = (int) (Math.random() * 100.0);  
    foo(k);  
}
```



Omniscient Slicing – Backtrac

- Kombination von Omniscient Debugging mit Slicing
 - Anwendungslauf generiert vollständigen Trace
 - erreicht durch Instrumentation zur Ladezeit ([class loader/agent](#))
 - keine Interaktion zur Laufzeit: „[post mortem](#)“-Debugging
- [Backtrac](#)
 - realisiert als Eclipse-Plugin
 - OD-ähnliche Navigation über die Ausführungszeit
 - zusätzliche Möglichkeit, dynamische Slices zu berechnen

→ [DEMO](#)



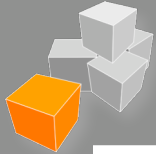
Hintergründe zu Backtrac

- Projekt **Steamloom** am FG Softwaretechnik
 - JVM mit Unterstützung für dynamische AOP
 - (De)Installation von Aspekten zur Laufzeit
 - **warum?** → Sprachmechanismen gehören in die VM!
 - minimale Aspekt-Infrastruktur, hohe Geschwindigkeit
- dynamische Aspekte
 - Verhalten der Anwendung ändert sich mit Aspektzuständen
 - dynamisches Slicing bietet gute Analysemöglichkeiten
- Unterstützung in Backtrac
 - Debugging für dynamische Aspektorientierung (→ Steamloom)
 - Backtrac bietet auch **Aspekt-Slicing**



Eigenschaften von Backtrac

- Backtrac ist ein Prototyp
 - entstanden im Rahmen einer Diplomarbeit
 - einige OD- und Slicing-Features (noch) nicht implementiert
- Kritikpunkte
 - Größe von Traces (derzeit: reines Textformat)
 - interner Speicherbedarf für Slicing-Graphen
- zukünftige Arbeiten
 - Erweiterung der OD-Fähigkeiten
 - Verbesserung der Slicing-Unterstützung
 - Anpassung für weitere AOP-Ansätze
 - Integration mit Lewis' Omniscient Debugger



Vielen Dank für Ihre Aufmerksamkeit!

<http://www.st.informatik.tu-darmstadt.de/>