INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

FOPRA

Prototyp zur Automation von Performanz-Messungen in Javabeans-basierten Anwendungen

Erika Kalix

Aufgabensteller:	Prof. Dr. H.-G. Hegering
Betreuer:	Rainer Hauck
Abgabetermin:	16. März 2001

Datum : 19. November 2001

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Der Trend in der Entwicklung von Anwendungsprogrammen geht in Richtung der Erstellung aus vorgefertigten Bausteinen mit Hilfe grafischer Umgebungen. Der Vorteil ist schnellere Entwicklung, Kostensenkung und bessere Qualität, da die Bausteine ja bereits umfassend getestet wurden. Bei derartigen umfangreichen Anwendungen ist es wichtig, Performanz-Messungen durchzuführen, um akzeptable Antwortzeiten für den Benutzer garantieren zu können. Die zur Zeit verfügbaren Systeme zur Performanz-Messung sind jedoch nur mit grossem Aufwand einsetzbar.

Daher wird hier ein neues Modell untersucht, das auf einer Idee von Rainer Hauck beruht [Hauc01]. Die Erstellung von Messpunkten wird dabei automatisiert. Grundlage sind Javabeans, die mit einem grafischen Tool - der Beanbox - zu einer Baustein-basierten Anwendung zusammengefügt werden.

Es wurde ein Prototyp erstellt, der die Realisierbarkeit von Rainer Haucks Konzept der Automatisierung von Javabeans-basierten Anwendungen zeigt. Damit werden automatisch Messpunkte zur Ermittlung von Antwortzeiten in einer Benutzertransaktion und Laufzeiten der einzelnen Bausteine (Subtransaktionen) erzeugt. Ziel der vorliegenden Arbeit ist die Instrumentierung von Javabeans und Beanbox derart, dass Performanz-Messungen gemacht werden können, ohne dass der Anwendungsentwickler größere Eingriffe in die Javabeans machen muss.

2 Eine bisherige Lösung: ARM

1. **Verfahren**

Bei ARM wird die Performanz-Messung durch ARM-Agenten, die eine ARM-API zu den Anwendungsbausteinen haben, und eine ARM-Management-Anwendung unterstützt. Agenten sind in jedem System vorhanden, die Management-Anwendung braucht im Gesamtsystem nur einmal zu existieren.
An den ARM-Agenten werden von jedem Baustein, der beim Ablauf der Anwendung durchlaufen wird, Bausteinname und Transaktions-ID zu Beginn und am Ende übergeben. Der ARM-Agent teilt dies zusammen mit der Meldezeit dem Managementsystem auf Anfrage mit. Das Modell dieser Performanz-Messung ist in Abbildung 1 dargestellt.

Diese Aufrufe sollen die Transaktion aber nicht wesentlich verzögern, das könnte bei hoher Belastung des Systems kritisch sein, nämlich insofern als die Antwortzeiten nicht mehr im akzeptablen Bereich liegen. Die Zusatzbelastung muss daher minimiert werden [C807] .

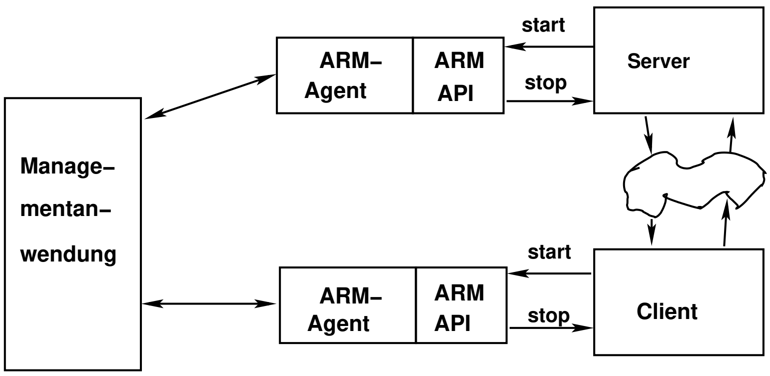

Abbildung 1: Modell der ARM-Agenten und Anwendungen

2. **Nachteile**

Bei ARM muss jeder Entwickler von Bausteinen für komplexe Anwendungen Schnittstellen zu den ARM-Agenten einbauen. Vorgefertigte Bausteine können nicht ohne größere Eingriffe in den Code eingebunden werden.
Für die Zuordnung von Subtransaktionen zu ihrer BTA ist es erforderlich, dass jeweils ein Identifikator übergeben wird. Das macht eine nachträgliche Instrumentierung von Legacy-Bausteinen praktisch unmöglich.
Ein weiterer Nachteil ist, dass keinerlei Automatisierung verfügbar ist.

3 Javabeans-basierte Anwendungen

3.1 Grundlagen der Javabeans

1. **Übersicht**

Javabeans sind wiederverwendbare Softwarekomponenten, die visuell in einer Toolbox verändert werden können. Meist sind sie relativ kleine Javaprogramme mit beschränkter Funktionalität. Sie enthalten Schnittstellen, welche eine Einbindung in ein größeres Programms mit Hilfe von bestimmten Events gestatten.
Sie ermöglichen so die Erstellung von Anwendungsprogrammen aus vorgefertigten, ausgetesteten Bausteinen. Die grafische Oberfläche der Anwendung kann den Wünschen des Benutzers entsprechend entwickelt bzw. angepasst werden.

Mit Hilfe von Javabeans sollen Anwendungsentwickler komplexe Anwendungen aus konfigurierbaren Basisbausteinen zusammensetzen können. Sie brauchen i.a. keine speziellen Javakenntnisse zu haben, sondern nur nur Kenntnisse des zu entwickelnden Systems. Bausteinentwickler können zusätzliche Javabeans erstellen; auch Ankauf ist möglich.

Es gibt 'sichtbare' Javabeans mit einer Schnittstelle zur grafischen Oberfläche und 'unsichtbare' Javabeans, die bestimmte Aufgaben im Hintergrund erledigen, z.B einen Zugang zu einer Datenbank ermöglichen.

Allen Javabeans gemeinsam ist Unterstützung von:

- Introspection: Die Entwicklungsumgebung und der Entwickler können erkennen, welche veränderbare Daten und welche Methoden eine Javabean bereitstellt-
- Customizing (Anpassung): Daten, Label und Eigenschaften können angepasst werden -
- Aneinanderkettung durch Events -
- Bereitstellung von Methoden, die von anderen Objekten aufgerufen werden können

Die Anpassung (Customizing) von Properties (Daten oder Eigenschaften wie Vorder- und Hintergrundfarbe einer grafischen Oberfläche) einer Javabean geschieht über ein Property-Fenster, in dem die eingestellten Werte angezeigt und gegebenfalls verändert werden.

Die für die Anwendung verwendbaren Methoden der Javabeans sind normale Java-Methoden, die von anderen Objekten aufgerufen werden können.

Standardevents wie `ActionEvent` erben von Klassen aus dem `package java/awt/event`, das die Grafikbehandlung in Java unterstützt; vom Bausteinentwickler erstellte `UserEvents` erben vom `java/util/EventObject`. Der Empfänger muss `java/awt/event/ActionListener` bzw. `java/util/EventListener` oder einen anderen eventspezifischen Listener implementieren.

2. **Struktur der Entwicklungsumgebung Beanbox:**

   Die Beanbox dient als Werkbank zur Erstellung bzw. Anpassung und zum Test von zusammengesetzten Anwendungen [Quin00]. Sie ist mit einer Beispiel-Javabean (Juggler) und zwei Buttonbeans in Abbildung 2 [Klui00] dargestellt.

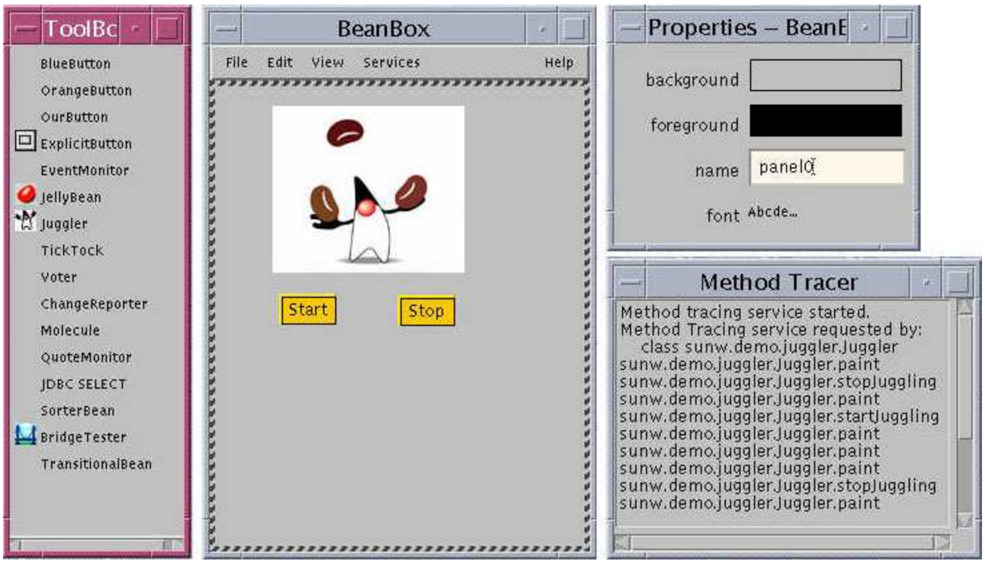

Abbildung 2: Entwicklungsumgebung Beanbox

In der Beanbox werden in einem Toolbox-Fenster Javabeans, z.B. Bausteine für Datenbankzugriffe u. ä. zur Verfügung gestellt. In einem weiteren Fenster, dem Werkbank-Fenster können die über `drag&drop` aus der Toolbox herübergezogenen Bausteine verknüpft werden.

Nach dem Laden der Beanbox werden drei Fenster aufgemacht (siehe Abbildung 2 (das Fenster `Method Tracer` gehört hier zum Juggler)) :

- das linke Fenster enthält die Toolbox, d.h. eine Liste derjenigen Beans, welche man als Bausteine verwenden kann (als jar-Dateien);
- das mittlere Fenster ist die Werkbank, auf der die Bausteine zusammengeführt, angepasst und getestet werden.
- Das rechte Fenster zeigt die Daten und Eigenschaften (Properties) des angeklickten Bausteins an, und ermöglicht eine Anpassung (Customizing). Im Runtime-Modus wird das Property-Fenster nicht angezeigt.

3. **Anwendung der Beanbox**

Es gibt hier den Design-Modus, in dem Javabeans angepasst und zu einer Anwendung verbunden werden, sowie den Runtime-Modus, in dem die Anwendung getestet wird. Im Design-Modus sind Sicherheitsrestriktionen abgeschaltet, ein Test kann hier nur die gröbsten Fehler anzeigen. In der vorliegenden Arbeit wird auf Security-Auflagen nicht eingegangen, da dies den Rahmen der Fopra sprengen würde [Jb101].

Das Menü `Edit` erlaubt die gerichtete Verbindung einer SourceBean zu einer TargetBean (s. Abbildung 3); z.B. über einen `ActionEvent` zu einer TargetBean-Methode, die den Event empfängt, wie z.B `actionPerformed` oder eine andere public-Methode, die die gleiche Schnittstelle wie `actionPerformed(ActionEvent e)` oder keinen Parameter hat.

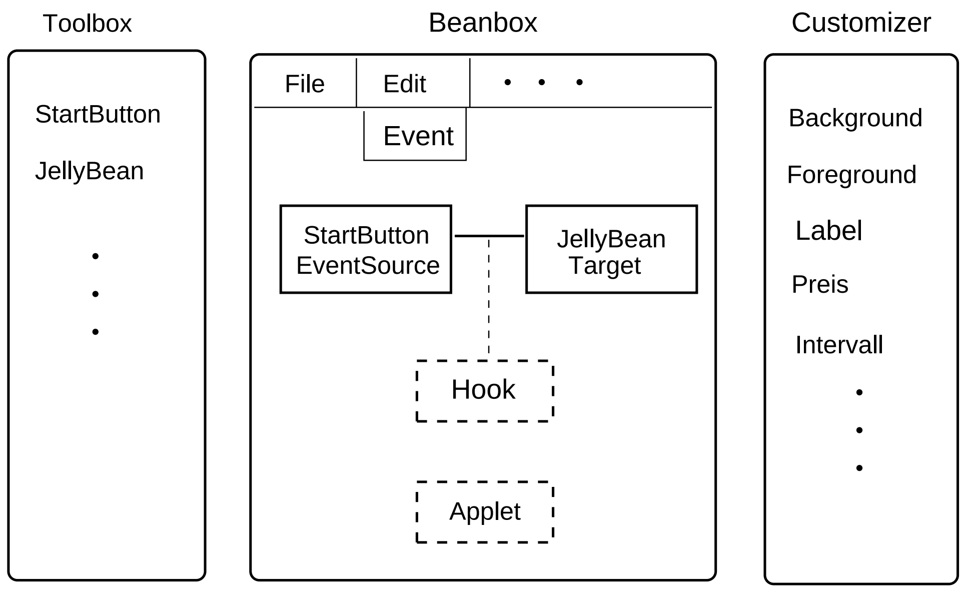

Abbildung 3: Struktur der Beanbox mit Supportsoftware

In der Beanbox kann mit dem `edit`-Menü erkannt werden,

- welche Events ein Javabean feuern - und
- welche Methode den Event empfangen soll (Schnittstelle im Hook) -sowie
- welche Methoden das Targetbean zur Verfügung stellt.

Durch Anklicken der Source-Bean (mit der Maus) wird diese aktiv; dies wird durch eine Umrandung angezeigt. Aus dem `Edit`-Menü kann ein Event (z.B. `ActionEvent`, `PropertyEvent` oder `MousePressedEvent`) gewählt werden, mit dem die Target-Bean verständigt werden soll.

Danach wird im Hintergrund durch Anklicken der gewünschten Target-Bean die Verbindung hergestellt. Dies wird durch einen roten Strich dargestellt . Ein Fenster geht auf, in dem man die zu aktivierende Methode des Targets auswählt.

Daraufhin wird vom `HookupManager`, einem Supportprogramm der Beanbox, durch Schreiben in ein File ein eindeutiger Adapter-Baustein `Hooknnn` erstellt. (Näheres siehe Abschnitt 3.3)

Der Entwickler kann selbst Beans erstellen. Diese werden in ein `jar`-File gebracht und im Verzeichnis `BDK1.1/jars` abgelegt. Damit stehen sie für die Beanbox bereit und können mit `File/loadjar` in die Toolbox geladen werden. Von da werden sie mit `drag&drop` auf die Beanbox-Werkbank gebracht und zusammen mit anderen zu einer Gesamtanwendung vereinigt.

3.2 Java-Eventkonzept

Events dienen im Javabean-Konzept dazu, Javabeans miteinander zu verbinden.

Es wird eine Java-JDK Version $>$ 1.2 und ein Javabeansystem BDK1.1 vorausgesetzt. In diesen wird das Event1.1-Modell von Java verwendet. Hierbei registrieren sich die empfangsbereiten Objekte bei den Event-versendenden Objekten. Diese senden nur an die registrierten Objekte, die den Event mit einer zu dessen Interface (s. EventListener) passenden Methode empfangen.

Der Java-Event-Mechanismus (V1.1) beruht auf Invocation, d.h. die dem Event zugewiesene Methode des Targetbeans wird wie eine Subroutine aufgerufen: nach Verlassen der Subroutine kehrt der Kontrollfluss zur Event-feuernden Javabean zurück (siehe Abbildung 4).

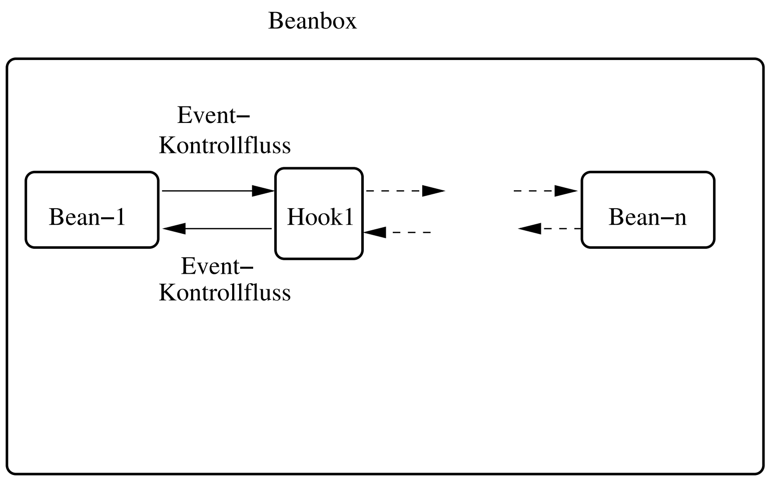

Abbildung 4: Beans und Hooks

Es handelt sich somit um ein synchrones Modell: der Kontrollfluss (`mainthread`) kommt nach Ausführung der durchlaufenen Methoden zum Startpunkt (`StartButton`) zurück. Javabeans können zur parallelen Bearbeitung von Aufgaben auch zusätzliche `Subthreads` erzeugen.

Java stellt Standardevents zur Verfügung, z.B. den `MousePressedEvent` und den `ActionEvent`. Der `MousePressedEvent` dient zum Start einer Anwendung durch Klick auf einen Button, evtl. auch zum Stop. Für die Verbindung der übrigen Javabeans in einer Anwendung eignet sich der `ActionEvent` am besten. In besonderen Fällen kann der `PropertyEvent` verwendet werden.

Der Baustein-Entwickler kann auch weitere Events implementieren, die vom `java/util/EventObject` erben. Parameter sollen aber laut Javabean-Spezifikation nur in besonderen Fällen übergeben werden ([Jb101], 6.4.1).

Potentielle Targetbeans registrieren sich als EventListener in der

- `addEventListener`-Methode der Event-Source bzw.

melden sich durch Aufruf der

-`removeEventListener`-Methode

wieder ab, wenn sie keine weiteren Events empfangen wollen.

3.3 Adapterkonzept

In der Beanbox dienen Adapter zu `Verdrahtung` von Javabeans.
Da Javabeans nur Sende- und Empfangsmethoden zur Verfügung stellen, aber keine Registrierung vornehmen, wird dies von der Supportsoftware der Beanbox durchgeführt.

Dynamisch zur Designzeit erfolgt (s. Abbildung 5)

- die Registrierung des empfangsbereiten Targetbean über Hook und Supportsoftware, die bei Abspeicherung in ein Applet überführt wird.
- die Zuweisung der gewünschten Methode im Targetbean.

Der HookupManager erstellt und übersetzt kleine Javaprogramme mit der folgenden Struktur:

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import java.awt.event.ActionListener;
import MediBean;
import java.awt.event.ActionEvent;
```

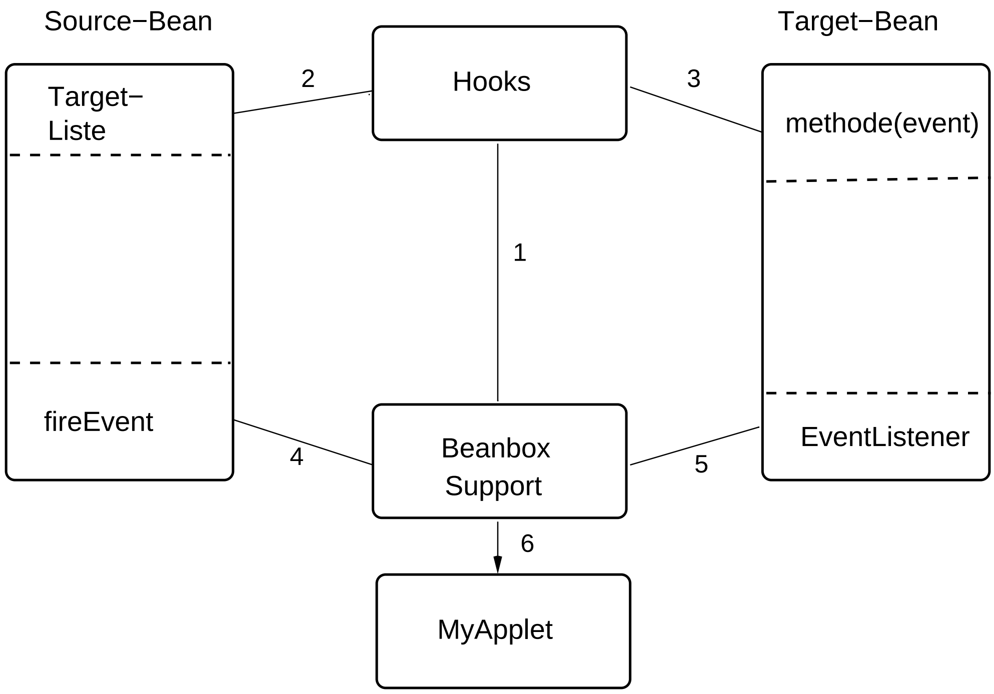

Abbildung 5: Beans, Hooks und Events in der Beanbox

```
public class ___Hookup_16c0e0beb7 implements java.awt.event.ActionListener,
                                 java.io.Serializable {

    public void setTarget(MediBean t) {
        target = t;
    }

    public void actionPerformed(java.awt.event.ActionEvent arg0) {
        target.mediAction(arg0);
    }

   private MessBean target;
}
```

Die erste Hookmethode setzt den Namen des Targets, der Zielbean, ein; die zweite Hookmethode ruft bei Eintreffen des `ActionEvents` die Zielmethode des targets auf.

Die Targetbean kann in der Zielmethode wieder einen `ActionEvent` feuern, sodass sich als Verbindung von mehreren Javabeans eine Eventkaskade ergibt.

Generell sendet die Source an alle in die Listener-Liste eingetragenen TargetBeans. In dieser Fopra wird nur der Fall behandelt, bei dem jeweils nur ein Target pro Source und Event eingetragen ist.

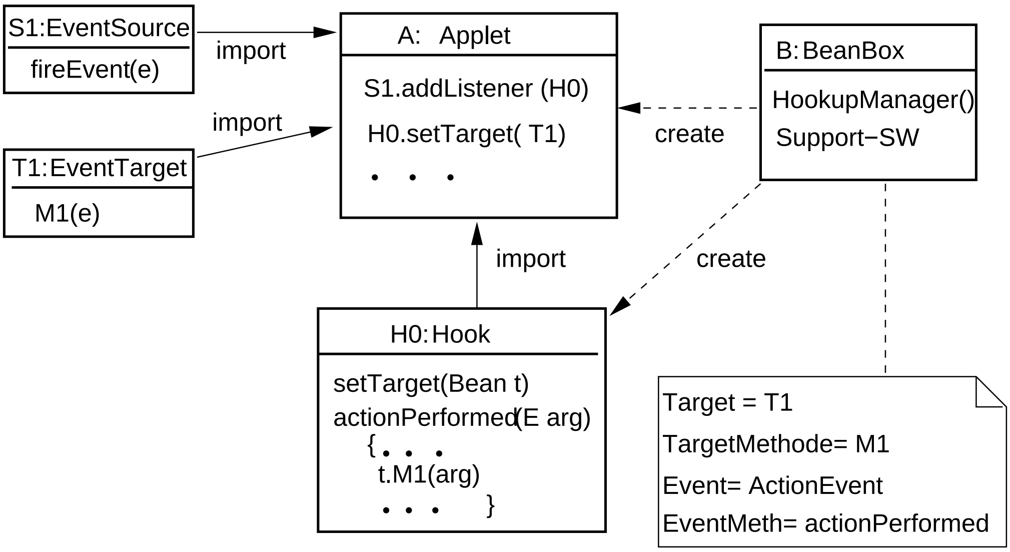

Abbildung 6: Beans, Hooks und Applet

Der `HookupManager` der Beanbox generiert einen anwendungs- und targetspezifischen `Hook`, ein kleines Java-Programm mit zwei Methoden, in dem dem Event die gewünschte Target-Methode zugewiesen wird.

Die Supportsoftware der Beanbox generiert zunächst ein Supportobjekt, das die ausgewählten Javabeans und die erzeugten Hooks importiert und instantiiert. Hier erfolgt auch der Eintrag der Hooks in die Listenerlisten. Das Supportobjekt kann in der Beanbox getestet und schließlich als Applet abgespeichert werden (siehe Abbildung 6).

4 Architekturkonzept

4.1 Übersicht

Ziel ist es, die Messdaten weitestgehend automatisch zu erzeugen. Dazu werden die Hooks bei ihrer automatischen Erstellung durch die Beanbox-SW instrumentiert. Die Instrumentierung von Javabeans soll nur in Ausnahmefällen erforderlich sein. Ein Messobjekt sammelt die erzeugten Messpunktdaten und gibt sie in ein Logfile aus, das als Ausgangsdatei für weitere Verarbeitung dient.

Eine Übersicht über die wichtigsten Elemente des Prototyps ist in der Abbildung 7 dargestellt. Die Reihenfolge der Aufrufe ergibt sich aus der Nummerierung. Alle aufgeführten Objekte liegen im gleichen Kontrollfluss:

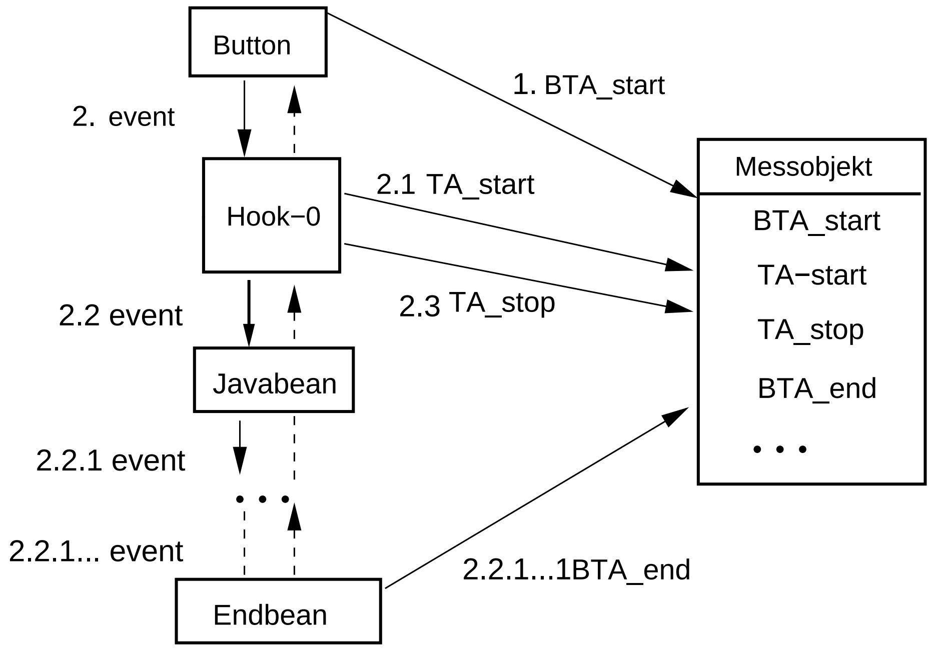

Abbildung 7: Wichtigste Schnittstellen und Hauptkontrollfluss

Zur Erstellung einer instrumentierten Anwendung ist erforderlich:

- **Instrumentierung der Beanbox: d.h. der Hooks:**

  Javabeans, die keine eigenen Subthreads und keine Oberflächenelemente enthalten, benötigen keine Instrumentierung. Ihre Laufzeit wird durch Messpunkte in dem Hook ermittelt, von dem sie aufgerufen werden. Feuern sie selbst noch einen Event, so werden im folgenden Hook weitere Messpunkte gesetzt. Die Zeiten werden im Messobjekt ermittelt, die Zeitdifferenzen im Agenten, der der Ergebnisse des Messobjekts auswertet.

- **Instrumentierung des StartButtons**:

  Damit die erzeugten Messpunkte einer bestimmten Instanz einer Benutzertransaktion (BTA) zugeordnet werden können, muss diese einen eindeutigen Namen haben. Dieser `BTA_name` wird der BTA durch Customizing des StartButtons (allgemein des Oberflächenelements mit dem die BTA gestartet wird) zugewiesen. Im Messobjekt wird beim Ablauf des Prozesses der `BTA_name` mit einer Zufallszahl zu einem eindeutigen BTA-Instanznamen (`BTA_ID`) ergänzt.

- **Instrumentierung der Javabibliotheken für Multithreads:**

  Bei Javabeans, die eigene Subthreads erzeugen, müssen der Start und das Ende des Threaddurchlaufs an das Messobjekt gemeldet werden. Dies geschieht in dieser Fopra durch Instrumentierung der `start`- und der `stop`- Methode der Klasse `BThread`, die von der Bibliotheksklasse `Thread` erbt.

  In einer Nachfolge-Fopra soll `BThread` durch die geänderte Klasse `Thread` der Javabibliothek ersetzt werden, da `BThread` nicht alle Beendigungsarten erfasst, und eine Instrumentierung der Multithread-Javabeans erfordert.

- **Instrumentierung der Multithread-Javabeans**

  In den Multithread-Javabeans müssen die Subthreads durch

  - `BThread thread = new BThread()`

  erzeugt werden. Dies entfällt, wenn in der Folge-Fopra die originale Klasse `Thread` der Javabibliothek instrumentiert wird.

- **Instrumentierung der aktiven Beans:**

  Aktive Javabeans erhalten Aufträge, die sie zeitunabhängig von der auftraggebenden BTA-Instanz durchgeführen. Damit die bei der Auftragsdurchführung erzeugten Messpunkte (für Start und Ende) der auftraggebenden BTA-Instanz zugeordnet werden können, muss bei ihrer Erzeugung die `BTA_ID` an das Messobjekt übergeben werden. Deshalb muss die aktive Javabeans bei Erhalt des Auftrags bei der Messbean die `BTA_ID` des Auftraggebers abfragen. Das ist möglich, weil die auftragempfangene Methode der aktiven Javabean im gleichen Kontrollfluss ausgeführt wird wie die auftraggebende Methode:

  - `BTA_ID = MessBean.get_BTA_ID`

  Für die Erzeugung von Messpunkten bei der Auftragsausführung ist erforderlich:

  - `ATA_start` bzw. `ATA_stop`

- **Erstellung des Messobjekts**

  Das Messobjekt stellt Methoden zur Zeiterfassung zur Verfügung, die von den `Hooks` bzw. `BThread` oder aktiven Javabeans aufgerufen werden. Die übergebenen Daten werden durch Zeitwerte ergänzt und periodisch in ein Logfile geschrieben.

- **Geeignete Events**

  In diesem Prototyp werden nur der `ActionEvent` und ein selbsterstellter `AlterEvent` zur Verkettung der Javabeans eingesetzt. Die verschiedenen `Mouse-Events` eignen sich zum Anstoß des `Startbuttons`. Der `PropertyEvent` ist weniger geeignet, da er Daten gleichen Typs in Source und Target voraussetzt, die auf gleiche Werte gesetzt werden sollen.

4.2 Messpunkte in Hooks

Die `Hook`-Objekte, die die Verbindungen zwischen den Javabeans erzeugen, werden nach Eingabe der relevanten Verbindungsdaten automatisch vom Beanbox-Programm `HookupManager` im Hintergrund erzeugt. Die Instrumentierung der `Hooks` wird daher über die Instrumentierung des `HookupManagers` durchgeführt.

Der `HookupManager` schreibt mittels `out.println` pro Verbindung ein `Hookupnnn.java`-File mit den Methoden:

- `setTarget` // Name der gewünschten Zielbean
- `actionPerformed` (z.B. bei ActionEvent) oder
- `alterPerformed` (bei alternativen Events)

In den `*Performed`-Methoden wird der von der SourceBean abgefeuerte Event empfangen und die benötigte Methode des TargetBeans aufgerufen. Nur der `StartButton` empfängt einen Event mit einer anderen, nämlich mit der `MousePressed`-Methode; diese kommt in keinem `Hook` vor.

In dieses `Hook-File` werden Aufrufe von Methoden eines Messobjekt so eingefügt, dass die Zeit vor und nach Ausführung der Zielmethode im TargetBean festgehalten wird.

Die verbundenen Javabeans und die `Hooks` werden von der Beanbox zu einer Gesamtanwendung (Beispiel `AbApplet` unten) zusammengefügt, die als Applet abgespeichert werden kann. Die folgenden Codezeilen zeigen einen Ausschnitt aus einem solchen Applet mit den für die Verbindung der Javabeans relevanten Anweisungen:

```
public class AbApplet extends Applet implements Serializable {

    public AbApplet() {
    ...
1.        // Create nested beans
```

```
        mediBean1 = (MediBean) Beans.instantiate(myLoader,"MediBean");

        ourButton1 = (sunw.demo.buttons.StartButton)
                   Beans.instantiate(myLoader,"sunw.demo.buttons.StartButton");
        addConnections();
    }

    private void addConnections() {
        try {
2.             hookup0 = new tmp.sunw.beanbox.___Hookup_16b2e51e6d();
               hookup0.setTarget(med6Bean1);
               ourButton1.addActionListener(hookup0);

        } catch (Exception ex) {
            System.err.println("Problems adding a target: "+ex);
            ex.printStackTrace();
        }
    }

 3.   // The fields used to hold the beans

    private MediBean mediBean1;
    private sunw.demo.buttons.StartButton ourButton1;

    // The hookups
    private tmp.sunw.beanbox.___Hookup_16b2e51e6d hookup0;
    private ClassLoader myLoader;
}
```

- Unter 1. werden die Javabeans der Anwendung `AbApplet` instantiiert;

- Unter 2. wird die Verbindung zwischen ihnen festgelegt:

  - in die `ActionEvent`-Listener-Liste der SourceBean `StartButton` wird Hookup0 eingetragen, das

  - ein Objekt der Klasse `MediBean` als Target setzt.

- Unter 3. sind die Objekte zu den Klassen `StartButton`, `Hookupnn`, `MediBean` aufgeführt. (`MediBean` ist eine Javabean, die in der Mitte einer Bausteinkette verwendet werden kann, da sie einen Event empfangen und einen Event senden kann.)

Feuert die SourceBean den `ActionEvent`, so gelangt er zunächst zum `Hook-Objekt` und veranlasst dort, dass die gewünschte Methode der TargetBean zwischen zwei eingefügten Messpunkten aufgerufen wird:

```
...
 actionPerformed(ActionEvent arg0){
...
 // setze Messpunkt (Targetname,..)
   target.zielmethode();
//  setze Messpunkt  (Targetname,..)
...
}
```

Mittels dieser Messpunkte wird im Agenten die Laufzeit der Zielmethode berechnet.

4.3 Messpunkte in Javabeans und Javabibliotheken

1. **StartButton**

   Der `StartButton` ist die erste Javabean in der Beankette der Anwendung. Der Start der Anwendung erfolgt durch Mausklick auf diesen Button. Um zu gewährleisten, dass alle Messpunkte der Anwendung einer eindeutigen Instanz zugeordnet werden, muss hier gleich das Messobjekt vom Start einer Anwendung mit deren `BTA_namen` verständigt werden. Dies ist dann gleich der erste Messpunkt. Eine Instrumentierung durch Codeergänzung ist erforderlich, weil die Beanbox es nicht gestattet, die Erzeugung eines Events als den Beginn einer `BTA` festzuhalten.

2. **Multithread-Javabeans**

   Unter `Multithread`-Javabean wird hier eine Javabean verstanden, in der mehr als ein `Thread` aktiv ist. In Abbildung 8 ist ein Ablauf dargestellt, bei dem der explizit von der Bean erzeugte Subthread vor Verlassen der MultiBean beendet wird. Allgemein werden die Subthreads aber nicht automatisch beendet, wenn der Hauptkontrollfluss die Bean verlässt.

   Damit das Messobjekt vom Start eines `Subthreads` verständigt wird, ist es notwendig, dessen Methode

   ```
   add_control_flow
   ```

   aufzurufen; entsprechend bei Stop oder Ende des Threads:

   ```
   rem_control_flow
   ```

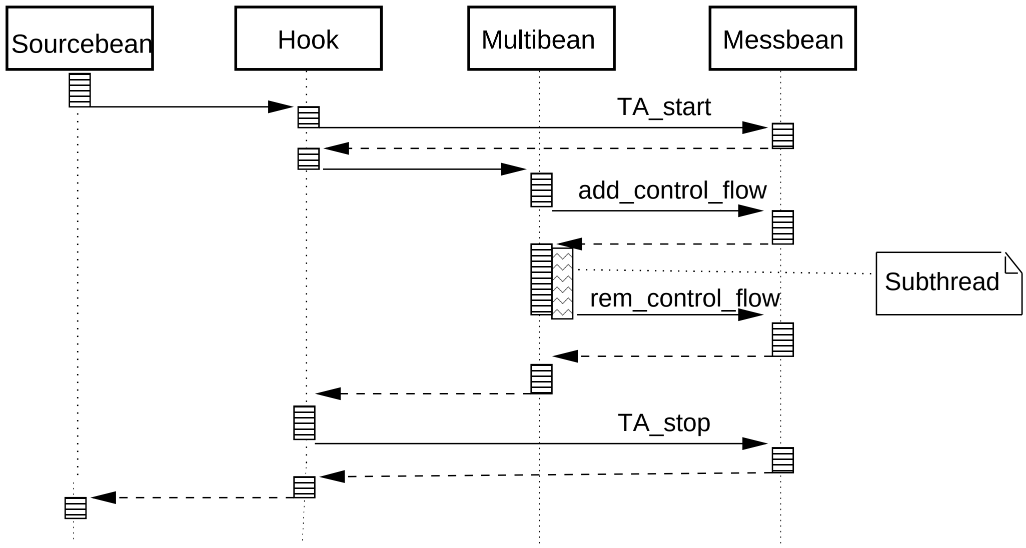

Abbildung 8: Multithread-Javabean

Das soll aber in einer erweiterten Javabibliothek geschehen. Dazu wird der Subthread als Instanz einer erweiterten Threadklasse `BThread` erzeugt.

3. **Javabibliothek**

   In der Extension `BThread` der Klasse `Thread` werden die erforderlichen Messpunkte in den Methoden `start()` bzw. `stop()` gesetzt. Die Methode `stop` ist in Java 1.2 deprecated, funktioniert aber noch.

   Für den in dieser Arbeit konstruierten Prototypen wird eine neue Klasse `BThread` erzeugt, die instrumentiert werden kann. Im Prototyp ist dieses Verfahren tolerierbar, in der Praxis ist eine Anpassung der Threadklasse anzustreben (in FolgeFopra vorgesehen).

4. **Aktive Javabeans**

   Aktive Javabeans sind neben den Oberflächenbausteinen (Eingabe- und `PresentationBeans`) die einzigen Javabeans, bei denen ein Eingriff in den Code nötig ist.

   Unter aktiven Javabeans versteht man solche, die Aufträge annehmen, in einer Warteschlange speichern und unabhängig vom Hauptkontrollfluss des Events später ausführen (siehe Abbildung 9). Damit hier gemessen werden kann, muss die BTA-Instanz der auftraggebenden Anwendung gespeichert und bei der Auftragsausführung ans Messobjekt weitergereicht werden.

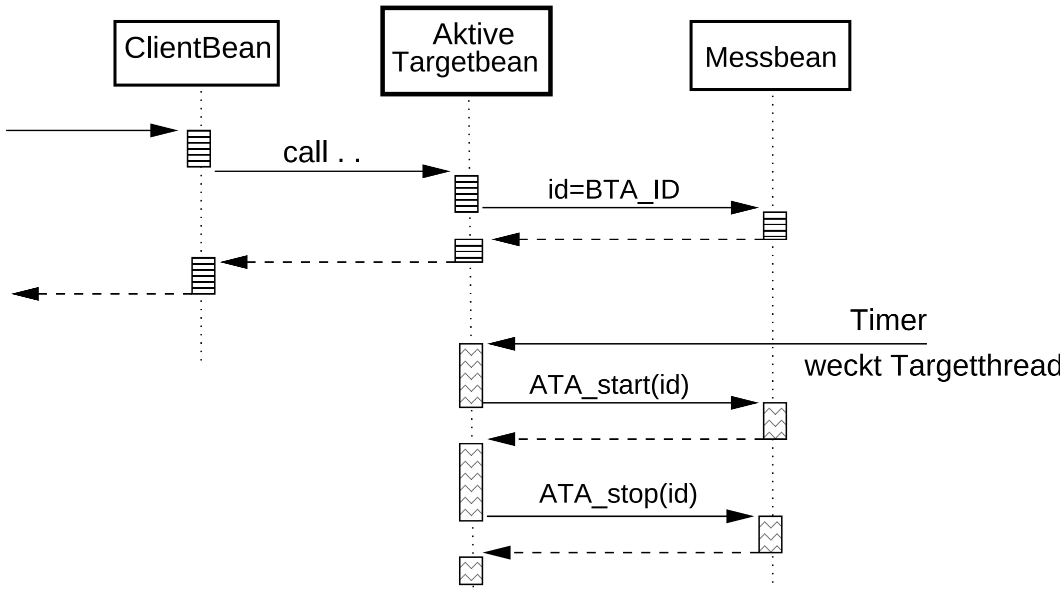

Abbildung 9: Aktive Javabean

Bei der Auftragsausführung ist der Beginn und das Ende der Bearbeitung sowie die zugehörige BTA-Instanz zu melden.

5. **Presentationbeans**

   Javabeans, die das Ergebnis der `BTA` präsentieren, rufen die Methode:

   - `BTA_end` der MessBean auf.

   Mit Hilfe dieses Messpunkts kann der Agent die Antwortzeit ermitteln.

4.4 Messobjekt

Das Messobjekt ist das einzige Teil, das hier vollständig neu entwickelt werden muss. Das Messobjekt wird als Javabean `MessBean` implementiert, da es als jarfile in dem jars-Verzeichnis so einfach von den `Hooks` gefunden wird, die es importieren müssen.

Die `public`-Methoden der `MessBean` werden innerhalb des BTA-Prozesses vom Kontrollfluss der BTA durchlaufen, sodass bei der Zeiterfassung hier wenig Zeitverlust entsteht.

Es soll nur eine Instanz der `MessBean` in jeder Beanbox bzw. jedem in einer Beanbox erstellten Applet geben, daher werden die `MessBean`-Tabellen als statisch implementiert. Auch die Methoden, die Tabellen ausfüllen, sind als statisch deklariert. Die `public`-Methoden der `MessBean` können so mit dem Namen der Klasse aufgerufen werden, Erzeugen einer Instanz ist nicht notwendig.

4.5 Schnittstellen der Javabeans-basierten Anwendung

1. **Externe Schnittstellen**

   Die Schnittstellen sind in der Abbildung 10 dargestellt:

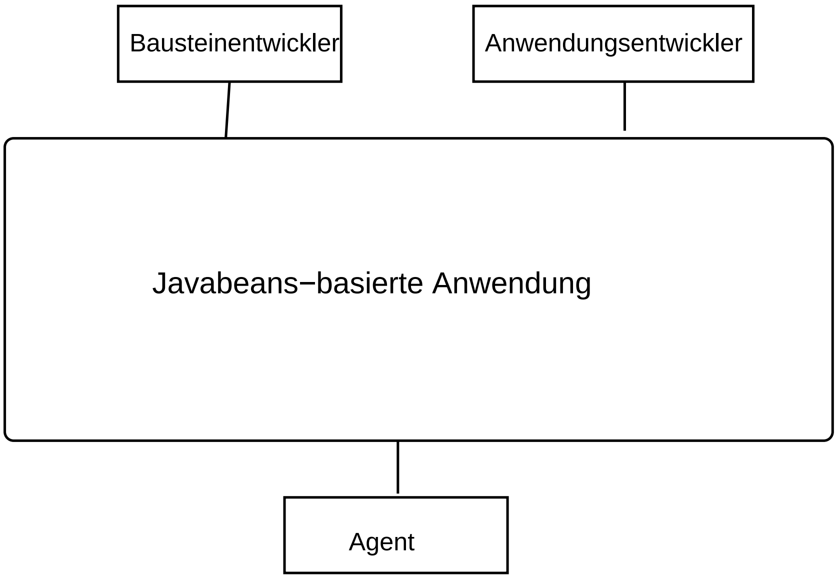

Abbildung 10: Externe Schnittstellen der Javabeans-basierten Anwendung

(a) **Schnittstelle des Baustein-Entwicklers zur Anwendung**

Der Bausteinentwickler muss nur bei aktiven Beans Codeänderungen einbringen. Unter einer aktiven Javabean bzw. Baustein versteht man einen Baustein, der Aufträge annimmt und diese dann zeitunabhängig von der Annahme in einem eigenen Thread ausführt.

Bei Entwicklung eines Präsentationsbausteins muss der Aufruf:

- `BTA_end(...)`

in die Javabean eingefügt werden, um einen Messpunkt zur Ermittlung der Antwortzeit bereitzustellen. (Die BTA ist erst mit der Rückkehr des Kontrollflusses zum Start abgeschlossen.)

Zur Entwicklung eigener Javabeans siehe unten.

(b) **Schnittstelle des Anwendungsentwicklers zur Anwendung**

- **Customizing:**
  Im Startbaustein muss der BTA ein Name (`BTA_name`) zugewiesen werden. Eventuell sind anwendungsspezifisch noch bestimmte Parameter in den Bausteinen zu setzen.
- **Zusammenstellen der Anwendung aus den Beans:**
  Die Bausteine (Javabeans) müssen in der richtigen Reihenfolge miteinander verbunden werden.

- **Test**

- **Abspeichern der Anwendung in einem Applet.**

(c) **Schnittstelle der Anwendung zum Managementagenten**

Das Ergebnis der Messung wird dem Managementagenten in einem Logfile übergeben, das folgende Daten enthält:

- Instanzidentifikation der Benutzertransaktion (BTA_ID)
- Zeitpunkt des Eintritts in die Bean (ms)
- Zeitpunkt des Austritts aus der Bean (ms)
- BeanName
- Kennung, ob Transaktion erfolgreich abgeschlossen wurde (success)

2. **Interne Schnittstellen**

In Abbildung 11 sind die Schnittstellen entsprechend der folgenden Spezifizierung durchnummeriert:

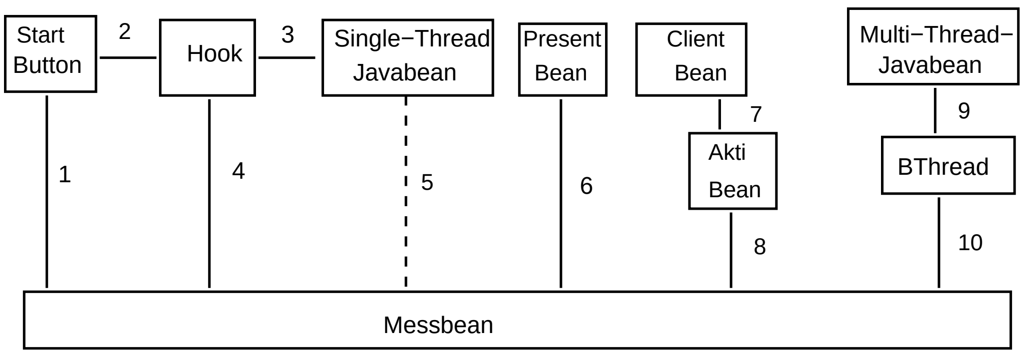

Abbildung 11: Interne Schnittstellen der Javabeans-basierten Anwendung

(a) **Schnittstelle des instrumentierten Buttons zur MessBean (1)**

Nachdem der Benutzer/Anwendungsentwickler die Transaktion durch Klicken auf den instrumentierten Button gestartet hat, wird durch:

- `BTA_start (mainThreadName, ButtonName);`

die Transaktion bei der `MessBean` angemeldet.

Wenn der Kontrollfluss zum `Startbutton` zurückkehrt, wird sein Verlassen der BTA-Instanz durch :

- `rem_control_flow ( ButtonName, success);`

abgemeldet.

(b) **Schnittstelle zwischen Javabeans und Buttons (2,3)**

Die Schnittstelle wird durch einen `ActionEvent`, in Ausnahmefällen durch den alternativen `AlterEvent` realisiert. Buttons sind immer nur Sourcen (Quellen), Javabeans meist Sourcen **und** Targets (Ziele).

Die Source sendet den Event (`ActionEvent` oder `AlterEvent`) an das mit ihr verbundene Target:

- `ActionEvent ae = new ActionEvent();`

Die Targetbean empfängt diesen mit der Methode:

- `target.meth(ae);` (meth beliebig, ActionEvent ae)

Zwischen Buttons und Javabeans sowie zwischen zwei Javabeans liegt immer ein `Hook` (siehe Abbildung 7).

(c) **Schnittstelle des instrumentierten Hooks zur MessBean (4)**

Die `Hooks` enthalten in der Methode

- `actionPerformed(arg0)`

vor dem Aufruf der Targetmethode die Aufrufe:

- `MessBean.TA_start(mainThreadName, TargetName);`

und nach dem Aufruf der Targetmethode den Aufruf:

- `MessBean.TA_stop(mainThreadName, TargetName, success);`

für den Zeitstempel in der `MessBean`.

Vor Rückkehr in den `Startbutton` wird der `mainThread` abgemeldet:

- `MessBean.rem_control_flow(success);`

(d) **Schnittstelle zwischen Javabeans und MessBean (5)**

Die Javabeans können optional Informationen an die `MessBean` senden mit dem Aufruf:

- `MessBean.logInfo( information, BeanName)`
wobei `information` ein String ist.

(e) **Schnittstelle der Presentationbean zur MessBean (6)**

Mit
- `BTA_end(..)`

wird der Messpunkt für die Ausgabe des Ergebnisses der BTA gesetzt.

(f) **Schnittstelle der auftraggebenden Bean zur aktiven Bean (7)**

Mit:
- `setTask (param)`

wird der aktiven Bean von der auftraggebenden `ClientBean` ein später unabhängig von der BTA-Instanz auszuführender Auftrag erteilt.

(g) **Schnittstelle der aktiven Bean zur MessBean (8)**

Es wird die BTA-Instanz abgefragt mit:
- `String BTA_ID = MessBean.getBTA_ID();`
Die aktive Bean meldet mit dem Aufruf:

- `ATA_start(BTA_ID, ownThread)`

den Start der Auftragsbearbeitung zur BTA mit der BTA-Instanz `BTA_ID` an.

Nach der Auftragsbearbeitung meldet sie ihren eigenen `Thread` wieder ab:

- `ATA_stop(BTA_ID, ownThread, success)`

(h) **Schnittstelle der Multithread-Beans zur BThread-Klasse (9)**

Bei Beans, die einen eigenen `Subthread` starten, werden Beginn und Ende der Transaktion durch den `SubThread` mittels instrumentierter `Thread-start-` und `-stop`-Methoden der `MessBean` angezeigt.

Dazu wird eine Klasse `BThread` bereitgestellt, die `Thread` um die Methoden:

- `start( String BeanName)` und
- `stop( boolean success)`

erweitert.

Die `Multithread`-Beans starten ihren `SubThread` mit:

- `BThread st = new BThread( subThreadName );`
- `st.start(BeanName);`

und beenden ihn mit:

- `st.stop(success);`

(i) **Schnittstelle der Klasse BThread zur MessBean (10)**

In der `BThread-start`-Methode wird der `SubThread` bei der `MessBean` angemeldet:

- `MessBean.add_control_flow(SubThreadName);`

In der `BThread-stop`-Methode wird der `SubThread` bei der `MessBean` abgemeldet:

- `MessBean.rem_control_flow(SubThreadName, success);`

5 Realisierung des Konzepts

5.1 Allgemeines

Als Toolbox dient im Rahmen dieser Arbeit die `Bean Development Kit BDK1.1`, eine von Java Sun für die Java-Version JDK1.2 und neuere zur Verfügung gestellte Testversion einer Beanbox. Sie baut auf dem `Java-Event1.1`-Modell auf, bei dem sich Objekte, die einen Event erwarten, in die Registrierungsliste der Source eintragen..

Diese Beanbox ist ein einfaches System, das zum Kennenlernen der Methodik des Zusammensetzens von Bausteinen dient. Es ist nicht zur Erstellung professioneller Anwendungen geeignet. Besondere Schwächen sind:

- keine Übergabe von Anwender-Parametern zwischen den Bausteinen mit den vorhandenen geeigneten Events, z.B. dem `ActionEvent`
- es kann nur ein `Applet` der Anwendung erstellt werden, keine `Applikation`

In den `Hook`-Bausteinen, die automatisch von der Beanbox erstellt werden, können Messpunkte gesetzt werden durch Aufrufe an ein zu implementierendes Messobjekt. In diesem werden Zeiten gemessen und zusammen mit `BTA-ID` und Bausteinname in ein Logfile geschrieben, das von einem Management-Agenten gelesen, weiterverarbeitet (z.B. Zeitdifferenzberechnung) und einem einem Managementsystem zur Verfügung gestellt werden.

Javabeans, die keine eigenen `Subthreads` enthalten und weder Anfang noch Ende einer `BTA` bilden, bleiben unverändert. Zur Messung ihrer Laufzeit dienen die Messpunkte in den anschließenden Hooks.

Ziel der Entwicklung ist es, die Machbarkeit von Performanz-Überwachung in Javabeans-basierten Anwendungen zu zeigen. Es werden daher möglichst einfache Bausteine erstellt, die nicht mehr als die notwendigen Funktionen haben.

Für die Verkettung der Javabeans wird der `ActionEvent` gewählt, in Ausnahmefällen der selbsterstellte `AlterEvent`. Der `PropertyEvent` ist hier nicht so geeignet, da seine Anwendung immer ein Vorhandensein und eine Veränderung von Variablen bestimmten Typs - in Source und Target der gleiche - erzwingt. Das kann aber bei vorgefertigten Javabeans i.a. nicht vorausgesetzt werden.

5.2 Instrumentierung von Javabeans und Javabibliotheken

1. **Instrumentierung des Startbuttons**

   Der `StartButton`, der den Beginn einer `BTA` bildet, wird instrumentiert durch:

   - Einbringen von standardisierten `Get-` und `Setmethoden` für den Wert `BTA_name`:

```
String getBTA_name()
void setBTA_name(String BTA_name)
```

- Übergabe des durch `Customizing` erhaltenen Anwendungsidentifikation `BTA_name` in der Methode `MousePressed` an das Messobjekt `MessBean` durch:

  - `MessBean.BTA_start( BTA_name, BeanName,thread);`

  Die Methode `MousePressed` wird durch Anklicken des Buttons mit der Maus aktiviert und startet die `BTA`.

Da der Kontrollfluss zu dem `Startbutton` zurückkehrt, wird dann in diesem Baustein mit `rem_control_flow` der Kontrollfluss beim Messobjekt abgemeldet.

2. **Instrumentierung des Präsentationsbausteins**

   Die Methode

   `MessBean.BTA_end`

   wird im Präsentationsbaustein aufgerufen, wenn dort das Ergebnis präsentiert wird. Wichtig ist diese Instrumentierung des Präsentationsbausteins für die Berechnung der Antwortzeit, die vom Agenten durchgeführt wird (siehe Abbildung 12).

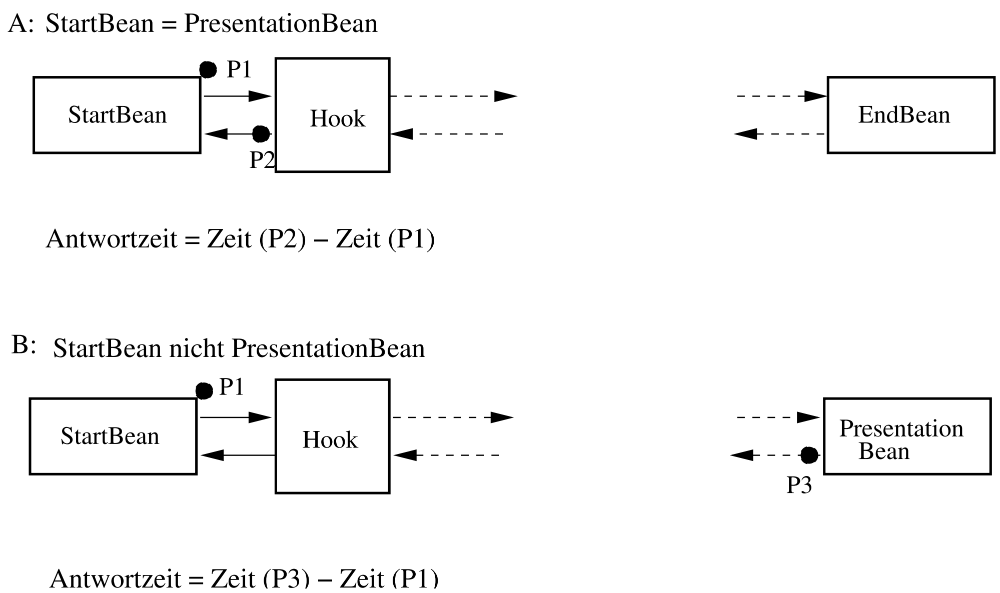

Abbildung 12: Ermittlung der Antwortzeit

3. **Instrumentierung der Beanbox**

In der Entwicklungsumgebung Beanbox werden die `Hook`-Programme instrumentiert, die die Verbindung zwischen den Bausteinen bilden. Da sie automatisch vom `HookupManager` erzeugt werden, erfolgt die Instrumentierung dort.
Der `HookupManager` schreibt die `Hooks` mit `out.println`.

Die für die Performanz-Messung in den `Hook` eingefügten Zeilen sind mit //* gekennzeichnet (Beispiel `ClientBean`):

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import ClientBean;
import java.awt.event.ActionListener;
import java.util.*;  //* fuer alternative Events
import java.io.*;    //* fuer Fehler- und Testausgaben
import MessBean;     //* Schnittstelle zum Messobjekt
import java.awt.event.ActionEvent;

public class ___Hookup_16d0c7a4d1 implements java.awt.event.ActionListener,
java.io.Serializable {

    public void setTarget(ClientBean t) {
        target = t;
    }

    public void actionPerformed(java.awt.event.ActionEvent arg0) {

     String fbName =  target.toString(); //* Targetnamen ermitteln
     int endpos = fbName.indexOf('[');   //* " "
     String tName = fbName.substring(0,endpos);    //* Targetname
     Thread ct = Thread.currentThread();  //* get Hauptthread
     String th = ct.getName();            //* get Hauptthreadname
     MessBean.TA_start(th, tName);  //* Messpunkt Beginn der Transaktion

     success = true;  //* Ergebnis initialisieren
     try {            //* Exception abfangen
        target.orderTask(arg0);        // auch im originalen Hook
     } catch ( Exception e) {    //*
       System.err.println("ERROR"+ e); //*
       success = false;          //* Ergebnis setzen
     }
       MessBean.TA_stop(th, tName, success); //* Messpunkt Ende Transaktion
    }
    private ClientBean target;
    private boolean success;  //* Ergebnisvariable
}
```

4. **Instrumentierung von aktiven Javabeans**

   Eine aktive JavaBean enthält öffentliche Methoden, die von auftraggebenden Javabeans (Clients) zur Speicherung von Aufträgen aufgerufen werden. In der implementierten Beispielbean `AktiBean` ist dies die Methode

   - `setTask (param).`

   Zunächst muss die aktive Javabean die BTA-Instanz des Auftraggebers ermitteln mit:

   - `String BTA_ID = MessBean.getBTA_ID();`

   `BTA_ID` und `param` werden in einer Tabelle abgelegt.
   Ferner enthält `AktiBean` einen eigenen `Thread`, der in einer `run`-Methode zyklisch Aufträge aus der Auftragstabelle liest und sie ausführt. Bei der Auftragsausführung ist folgende Instrumentierung erforderlich, um die entsprechenden Messpunkte mittels Messobjekt zu setzen:
   Beim Start:

   - `MessBean.ATA_start(BTA_ID, Name des eigenen Threads)`

   Dabei ist `BTA_ID` die BTA-Instanz der auftraggebenden `BTA`.

   Nach der Auftragsbeendigung:

   - `MessBean.ATA_stop(BTA_ID, Name des eigenen Threads, success)`

   Der Name des eigenen `Threads` wird eingefügt, weil auch in der aktiven Javabean ein Multithreading möglich ist und eine Zuordnung von Anfang und Ende eines `Threads` möglich sein soll.

5. **Instrumentierung von Client-Javabeans**

   Unter Client-Javabeans wird hier eine Javabean verstanden, die Aufträge an eine aktive Bean erteilt. Optional wird sie (es könnte auch jede andere Javabean sein) mit einem Informationsaufruf an die MessBean instrumentiert:

   - `MessBean.logInfo( info)` // (String info)

6. **Instrumentierung der Multithread-Javabeans**

   Multithread- Javabeans erzeugen zusätzliche `Threads`, Der Kontrollfluss wird parallel zur Ausführung der Thread-Aktivitäten im Hauptthread fortgesetzt.
   Bausteine, die eigene `Threads` erzeugen, um Aufgaben parallel zu bearbeiten, erzeugen die Threads mit:

   - `BThread tm = new BThread( threadname);`

   Mit `tm.start()` wird der neue `Thread` gestartet, mit `tm.stop(success)` beendet. (`BThread` s.u.; Instrumentierung entfällt mit neuer `Thread`-Klasse, die in Nachfolge-Fopra implementiert wird.)

7. **Instrumentierung der Javabibliotheken**

   Die Java-Klasse `BThread` wird als Extension der Klasse `Thread` aus der Javabibliothek erzeugt. Dazu werden die Methoden start und stop erweitert. Die neue `start`-Methode meldet das Starten eines neuen `Threads` mit

   - `add_control_flow`

   an das Messobjekt und ruft danach die Standardmethode `start` der Oberklasse auf.

   Die neue `stop`-Methode meldet das Stoppen eines `Threads` mit

   - `rem_control_flow`

   an das Messobjekt und ruft danach die Standardmethode `stop` der Oberklasse auf.

   Da die Methode `stop` in Java1.2 deprecated ist, und auch nicht alle Threadbeendigungen damit erfasst werden, wird `BThread` in einem folgenden Fopra durch eine geänderte `Thread`-Klasse in der Javabibliothek ersetzt.

5.3 Anpassung (Customizing)

Die `BTA`-Klasse erhält ihren Namen durch `Customizing` des `Startbuttons`.
Bei Javabeans, die je nach gesetztem Parameter `status` eine Verzweigung in der Anwendung ermöglichen, kann dieser (z.B in der `IfelBean`) mittels `Customizing` gesetzt werden, sodass je nach seinem Wert ein verschiedener Ablauf (verschiedene Javabeans anschließen) möglich ist. (Setzbare Parameter benötigen standardisierte Zugriffsroutinen).

Die durch `Customizing` veränderten Werte werden persistent, wenn sie mit `file/save` der Beanbox in eine Datei geschrieben werden. Von dort können sie mit `file/load` wieder in die Beanbox geladen werden.

5.4 AlterEvent

Zum Test von möglichen Verzweigungen in Anwendungen, die aus Javabeans zusammengesetzt sind, ist ein von `ActionEvent` verschiedener Event erforderlich: dazu wurde der `AlterEvent` implementiert.

Die Verwendung alternativer Events, die z.B. auch Parameter übertragen können, ist nur in Linux-Umgebungen möglich.
Wie bei Properties sind dabei auch Namenskonventionen zu beachten.
Hat eine alternativer Event den Namen `AlterEvent`, so ist ein Interface

- `AlterListener.java`

zu erstellen und von den potentiellen Targetbeans zu implementieren. Die spezielle Empfangsroutine ist

- `alterPerformed(AlterEvent e)`

Zum Registrieren und Deregistrieren dienen die Methoden:

- `addAlterListener`
- `removeAlterListener.`

Javabeans laufen in einer Multithread-Umgebung, daher müssen Methoden, die sich gegenseitig beeinträchtigen oder bei gleichzeitigem Lauf inkonsistente Zustände bewirken könnten, als `synchronized` erklärt werden.
Dies muss berücksichtigt werden, wenn eine Javabeans mehrere Methoden enthält, die durch einen Event aktiviert werden können, wie `alterPerformed(AlterEvent e)` und `actionPerformed(ActionEvent e)`. (Beispiel: start- und stop-Methode beim Juggler [Bdk11])

Der `AlterEvent` wird von der Testjavabean `IfelBean` im Fall von `status` = 2 gesetzt versendet und von der `FineBean` empfangen.

5.5 Messobjekt

Das Messobjekt (Abbildung 13) wird als Javabean mit dem Namen `MessBean` implementiert.
Das bietet sich an, da so alle mit der Anwendung verbundenen Bausteine gleiche Struktur haben und somit der Erstellungsaufwand minimiert wird. Nur die Grafikteile sind bei dieser `unsichtbaren` Bean nicht erforderlich.
Ferner wird es so als `jarfile` im `jars`-Verzeichnis von dem erstellten `Hook`, der es importieren muss, einfach gefunden.

Die `MessBean` enthält zwei Tabellen:

- die `BTA-Instanztabelle`, die die Kontrollflüsse den BTA-Instanzen zuordnet

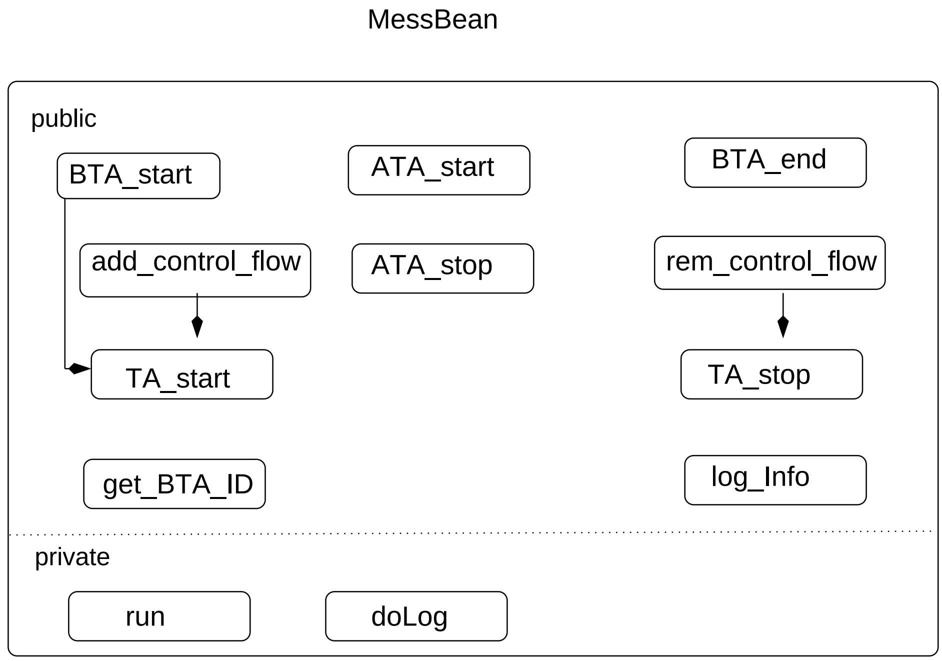

Abbildung 13: Methoden in MessBean

- die `TA-Tabelle`, bei der ein Eintrag (Messpunkt) folgende Werte enthält:
  - BTA-Instanz
  - Art des Messpunkts (`TA_start, TA_stop`,..)
  - ermittelter Zeitwert
  - Erfolg/ Misserfolg/ leer
  - Bausteinname

Die `MessBean` enthält die Methoden:

- `BTA_start(String BTA_name, String BeanName, String id)` :

  Aus `BTA_name` und einer mittels `getTime()` erzeugten Zufallszahl wird ein eindeutiger Instanzname gebildet; auch der Hauptthread erhält einen eindeutigen Namen. Beides wird in die `BTA-Instanz-Tabelle` eingetragen. `Ta_start` wird zum Setzen des ersten Zeitpunktes aufgerufen.

- `BTA_end ( String BeanName, String thread, boolean success)` :

  Ein Messpunkt zur Ermittlung der Antwortzeit wird gesetzt.

- `TA_start (String thread, String target)`:

Aus dem `thread`-Namen wird mit Hilfe der `BTA-Instanztabelle` die BTA-Instanz ermittelt. Die aktuelle Zeit wird abgefragt. Der Zeitpunkt des Aufrufs, die aktuelle Bean , die BTA-Instanz und der Zustand - (vor dem Baustein) werden in die `TA-Tabelle` eingetragen.

- `TA_stop (String thread, String target, boolean success)` :

  Aus dem `thread`-Namen wird mit Hilfe der `BTA-Instanz-Tabelle` die BTA-Instanz ermittelt. Die aktuelle Zeit wird abgefragt. Der Zeitpunkt des Aufrufs, die aktuelle Bean , die BTA-Instanz und der Zustand (erfolgreich / nicht erfolgreich, nach dem Baustein) werden in die `TA-Tabelle` eingetragen.

- `add_control_flow ( String subThread )` :

  Mit Hilfe des `mainthread`-Namens wird die zugehörige Instanz aus der `BTA-Instanztabelle` ermittelt. Der neue `Thread` wird der BTA-Instanz zugeordnet in die Instanztabelle eingetragen.
  Für `Subthreads` innerhalb einer Javabean-Anwendung werden vom JVM eindeutige Namen (thread1, thread2,...) vergeben, sofern vom Javabean nicht schon ein eindeutiger Name zugeordnet wurde. Diese `Thread`-Identifikation wird nicht geändert. Die Methode TA_start wird aufgerufen.

- `rem_control_flow ( String subThread, boolean success)` :

  Mit Hilfe des `Subthread`-Namens wird der zugehörige Eintrag in der `BTA-Instanztabelle` ermittelt und ausgetragen. Die Methode `TA_stop` wird aufgerufen.

- `String getBTA_ID()`:

  Auch die Methoden der `MessBean` laufen im `mainthread` ab. Sein Name kann hier ermittelt werden. Die zugehörige Instanz der Anwendung wird aus der `BTA-Instanz-Tabelle` gelesen und zurückgegeben. Diese Funktion wird in Javabeans benötigt, welche aktive Javabeans aufrufen.

- `ATA_start ( String BTA_ID, String subThread )`:

  Diese Methode wird von aktiven Javabeans aufgerufen, um den Beginn einer Auftragsausführung zu melden.

  Der neue `Subthread` wird der BTA-Instanz zugeordnet in die `BTA-Instanztabelle` eingetragen, dies ist notwendig für den Fall, dass die aktive Javabean vom eigenen

`Thread` aus weitere `Subthreads` erzeugt. Diese erhalten bei Start und Ende Messpunkte mit der gleichen Methode wie der Hauptthread.

Die aktuelle Zeit wird abgefragt und zusammen mit der BTA-Instanz und der Messpunktbezeichnung `TA_start` in die `TA-Tabelle` eingetragen. Als Beanname wird `dummy` geschrieben.

- `ATA_stop (String BTA_ID, String subThread, boolean success)` :

  Diese Methode wird von aktiven Javabeans aufgerufen, um das Ende einer Auftragsausführung zu melden. Der Zeitpunkt und die BTA-Instanz, sowie `dummy` für den Beannamen und `TA_stop` als Messpunktbezeichnhung werden in die `TA-Tabelle` geschrieben und der `Subthread` wird aus der `BTA-Instanztabelle` ausgetragen.

- `logInfo (String info, String beanName)`:

  Die Information wird übernommen und zusammen mit BTA-Instanz, BeanName und Zeitpunkt in die `TA-Tabelle` eingetragen.

- `run()`

  In der `run`-Methode läuft ein `Subthread`, der zyklisch die `doLog`-Methode aufruft und sich zwischendurch mit `sleep` schlafen legt.

- `doLog()`:

  Die Messdaten der `TA-Tabelle` werden in das Logfile geschrieben.

5.6 Erstellung eines Applets

Mit `file/create Applet` kann aus zusammengefügten Javabeans in der Beanbox ein `Applet` erstellt und abgespeichert werden.

5.7 Ausnahme- und Fehlerbehandlung

Für die Behandlung von Fehlern und Ausnahmezuständen gibt es folgende Möglichkeiten:

1. Fehler können durch einen Return-Parameter an die aufrufende Instanz weitergegeben werden. Der Nachteil dabei ist, dass alle Javabeans geändert werden müssten.
2. Fehler könnten eine Verzweigung in eine `ErrorBean` mittels eines `User-Events` auslösen: Der Aufwand dafür ist relativ gross, da `User-Event` und `ErrorBean` erstellt

werden müssten. Alle Javabeans müssten instrumentiert werden.

3. Bei Fehler wird eine Variable in der Bean gesetzt, die im `Hook` über eine `get`-Routine ausgelesen werden kann. Das Fehlen der `get`-Routine kann im `Hook` abgefangen werden. Auch hier müssten alle Javabeans instrumentiert werden.

4. Fehler durch `Exceptions` im `Hook` abfangen: Zuvor können Fehler schon in den Javabeans durch `try und catch` abgefangen und mittels `throw` weitergeleitet werden.

Das letzte Verfahren ist hier wohl die beste Lösung, da sie in vielen Javaprogrammen und somit auch in den Javabeans ohnehin eingesetzt wird.
Eine Weiterleitung vom `Hook` aus ist bei Verwendung des `ActionEvents` nur für Errors und Runtime Exceptions möglich, leider nicht für beliebige andere Exceptions, da die zugehörige Methode `actionPerformed` kein `throws Exception` enthält, und nicht durch eine erweiterte Methode mit `throws` überschrieben werden darf.
Dazu müsste ein entsprechend definierter `UserEvent` implementiert werden. Im `Hook` könnte aber auch ein entsprechender Parameter im aufrufenden Bean gesetzt werden, wenn eine Setmethode dafür bereitgestellt wird.

6 Test

Der Test erfolgt in der Beanbox, (erst im Design-Modus, dann im Runtime-Modus) die Anwendung läuft also im gleichen Adressraum wie diese. Die Tests wurden in einer Linux-Umgebung (SuSE Linux 7.0) und in Windows95 durchgeführt.
Zum Test wird die Anwendung in der Beanbox aus angepassten, miteinander verbundenen Javabeans erstellt, bzw. als bereits erstelltes Applet in die Beanbox geladen.
Der Test des Applets außerhalb der Beanbox ist wegen des Fileschreibens der MessBean nicht möglich.

Bei den folgenden Testergebnissen für einen Messpunkt ist:

```
Zeile  Beispiel               Bedeutung
-------------------------------------------------------------------
1      X-Applit979985131430   BTA-Instanz
2      TA_start               aufgerufene Methode von MessBean
3      979985131430           der Zeitpunkt in ms
4      -                      Erfolg / kein Erfolg / noch nichts
5      StartButton            Name der JavaBean
```

6.1 Testfall: nur ein Kontrollfluss

Die Struktur der getesteten Anwendung ist in Abbildung 14 dargestellt:

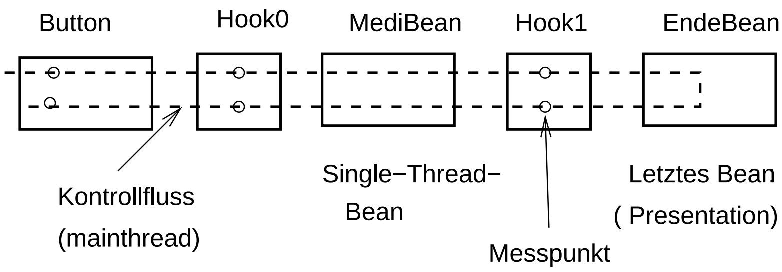

Abbildung 14: Test mit Single-Thread-Beans

Für diesen Test wurden die Javabeans `MediBean` und `ZetaBean` implementiert. `MediBean` empfängt und sendet einen `ActionEvent`, `ZetaBean` empfängt einen `ActionEvent`. Der Kontrollfluss (Thread bei Java) geht vom Button über `MediBean` und `ZetaBean` wieder zurück zum Button.
Ergebnis:

```
X-Applit979985131430
BTA_start
979985131430
-
StartButton
```

```
X-Applit979985131430
TA_start
979985131540
-
MediBean

X-Applit979985131430
TA_start
979985131650
-
PresentBean

X-Applit979985131430
BTA_end
979985131710
erfolgreich
PresentBean

X-Applit979985131430
TA_stop
979985131760
erfolgreich
PresentBean

X-Applit979985131430
TA_stop
979985131870
erfolgreich
MediBean

X-Applit979985131430
TA_stop
979985131980
erfolgreich
StartButton
```

Die Anzeige von `BTA_end` besagt, dass in dieser `PresentBean` das Ergebnis der BTA angezeigt wurde, Der Test läuft erfolgreich in Linux und Windows95.

6.2 Testfall: mehr als ein Kontrollfluss

Die Struktur der getesteten Anwendung ist in Abbildung 15 dargestellt:

Zu diesem Test wurde die Javabean `MuliBean` implementiert, die einen eigenen Thread erzeugt. Die `run`-Methode wird durch `stop` beendet. Damit die Zeitpunkte der Threadbeginns

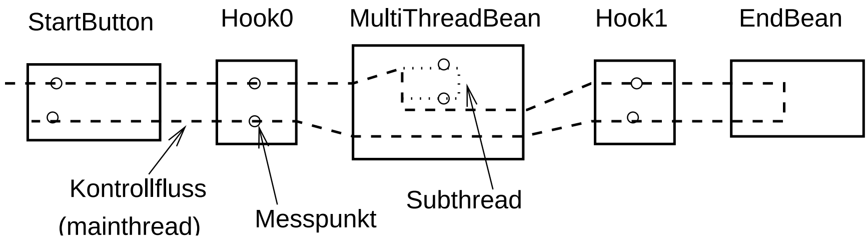

Abbildung 15: Test mit Multi-Thread-Bean

und -endes automatisch festgehalten werden, wird der Thread als Objekt der Klasse `BThread` erzeugt, die eine Extension der Klasse Thread ist.
Ergebnis:

```
...
...

X1-Applit979986050890
TA_start
979986051000
-
MuliBean

X1-Applit979986050890
TA_start
979986051160
-
dummy

X1-Applit979986050890
TA_stop
979986051220
erfolgreich
dummy

X1-Applit979986050890
TA_stop
979986051490
erfolgreich
MuliBean

...
```

`dummy` bezeichnet hier einen Subthread. In diesem speziellen Test waren die Wartezeiten so eingestellt, dass der Subthread vor Ausgang des Hauptthreads aus der `MuliBean` beendet war. Es ist mit anderen Parametern aber auch möglich, dass der Subthread seine Aufgaben

erst dann durchführt, wenn der Hauptkontrollfluss die `MuliBean` schon verlassen hat. Dies hängt auch von der Priorität der Subthread ab.
Der Test läuft erfolgreich in Linux und Windows95.

6.3 Testfall: Alternativer Event

Die Struktur der getesteten Anwendung ist in Abbildung 16 dargestellt:

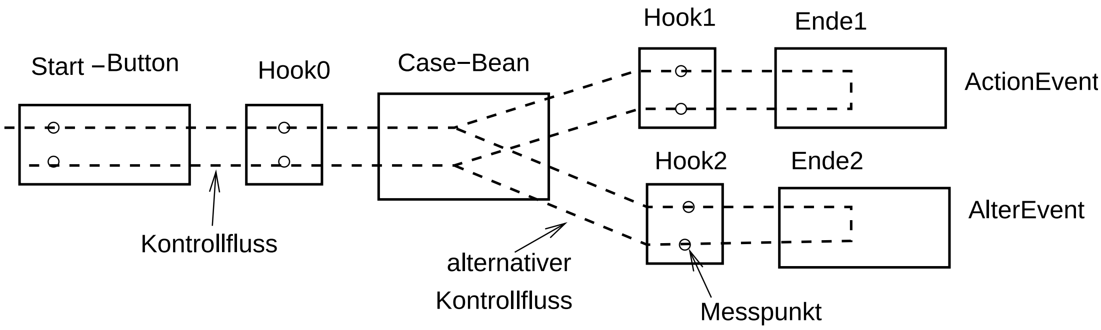

Abbildung 16: Test mit alternativem Event

Alternative Events sind erforderlich, wenn in einer Javabean eine Verzweigung eingeleitet werden soll, z.B. im Fehlerfall eine besondere Bean durch einen Event verständigt werden soll.

Zum Test wurde ein `AlterEvent`-Objekt und ein `AlterListener`-Interface erstellt (siehe bei Installation). Das Event-Feuern wurde in die Javabean `IfelBean` eingebaut; es erfolgt nach geeignetem Customizing. Der `AlterEvent` wird von `FineBean` empfangen. Die Verknüpfung läuft wie beim üblicherweise verwendeten `ActionBean`, auch am Ergebnis kann man nicht erkennen, dass ein anderer Event verwendet wurde. Zweckmäßig ist ein alternativer Event nur dann, wenn je nach Zustand der Bean zu verschiedenen Methoden verzweigt werden soll.
Ergebnis:

```
X5-Applit980001801313
BTA_start
980001801342
-
StartButton

X5-Applit980001801313
TA_start
980001801414
-
IfelBean

X5-Applit980001801313
```

```
TA_start
980001801443
-
FineBean

TA_stop
980001801479
erfolgreich
FineBean

X5-Applit980001801313
TA_stop
980001802477
erfolgreich
IfelBean

X5-Applit980001801313
TA_stop
980001802497
erfolgreich
StartButton
```

Um welchen Event es sich bei dem Ablauf handelt geht aus den für den Managementagenten gesammelten Daten nicht hervor. Der Test läuft in Linux-Umgebung erfolgreich; in Windos95 kann ein alternativer Event nicht implementiert werden (Fehler beim Übersetzen).

6.4 Testfall: Aktive Bean

Die Struktur der getesteten Anwendung ist in Abbildung 17 dargestellt:

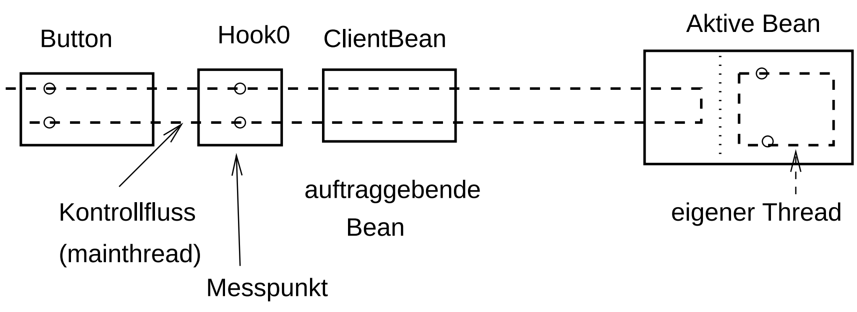

Abbildung 17: Test mit aktiver Bean

Zu diesem Test wurde eine auftraggebende Javabean `ClientBean` erstellt, die ihre BTA-Instanz bei der `Messbean` abfragt, die Auftragvergabe mit `logInfo` meldet und einen Auftrag bei der aktiven `AktiBean` setzt.

Ergebnis:

```
...
...
Y-Applit980071185406
Info: Auftrag an AktiBean gegeben
980071185867
-
ClientBean

Y-Applit980071185406
TA_stop
980071185997
erfolgreich
ClientBean

Y-Applit980071185406
TA_stop
980071186003
erfolgreich
StartButton

Y-Applit980071185406
ATA_start
980071188321
-
dummy

Y-Applit980071185406
ATA_stop
980071188365
-
dummy
```

Hier wird die Methode `logInfo` der `MessBean` aufgerufen und die Information `Auftrag an AktiBean gegeben` als Parameter mitgegeben. Der Kontrollfluss kehrt zum `StartButton` zurück. Erst danach wird die Auftragsbearbeitung in der aktiven Bean gestartet: `ATA_start`.

6.5 Testfall: Exception

Die Struktur der getesteten Anwendung ist in Abbildung 18 dargestellt:

Zu diesem Test wurde eine Exception auslösende Javabean `XcepBean` erstellt, die eine Nicht-Runtime-Exception wirft. Diese wird im `Hook` aufgefangen und als `success = false` mit `TA_stop` an die `MessBean` gemeldet. Der Hook kann mit der Methode `actionPerformed` hier keine Exception werfen, sodass nur am Ausgang der `XcepBean nicht erfolgreich`

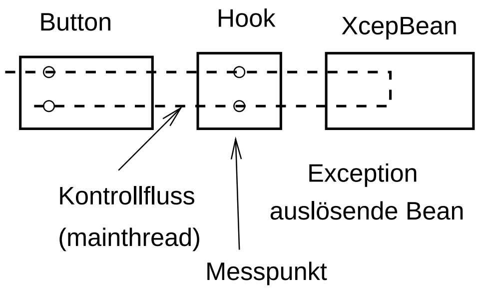

Abbildung 18: Test Exception bewirkende Bean

gemeldet wird, später aber nicht mehr. Dies muss der Agent berücksichtigen (im Gegensatz zu Errors und Runtime Exceptions).

```
...
...

Y3-Applit980090421740
TA_stop
980090422070
nicht erfolgreich
XcepBean

Y3-Applit980090421740
TA_stop
980090422120
erfolgreich
StartButton
```

Der Ausschnitt zeigt die Log-Information nach dem Austritt des Kontrollflusses aus der den Fehler erzeugenden `XcepBean`: der Fehler wird mit `nicht erfolgreich` angezeigt. Er kann durch die `actionPerformed`-Methode des `Hooks` nicht an den Button weitergereicht werden, da diese keine Exception wirft. Eine Verbesserung ist mit selbstdefinierten Events möglich.

7 Zusammenfassung und Ausblick

Ausgehend von Rainer Haucks Idee ([Hauc01]), Performanz-Messungen unter Verwendung der `Hooks` zu machen, die die Javabeans verbinden, wurde zunächst der Code der Beanbox nach der Stelle durchsucht, an der die `Hooks` generiert werden. Diese fand sich im `HookupManager`: Dort wird ein `Hookfile` generiert und sein Code mittels `println` geschrieben. Anschließend wird der `Hook` übersetzt.

Um Messpunkte zu erzeugen, wurden entsprechende Aufrufe an ein Messobjekt vor und nach dem Zielmethodenaufruf eingebracht. Das Messobjekt mit diesen Methoden, die die Messpunktdaten ergänzen, wurde implementiert. Es setzt die automatisierten Aufrufe in Tabelleneinträge um und schreibt die um Zeitwerte ergänzten Messdaten in ein Logfile. Dieses dient als Ausgangspunkt für eine weiteres Fopra zur Übergabe an das Management.

Es wurden Testjavabeans und ein `UserEvent` erstellt und Tests für die verschiedenen Testfälle durchgeführt. Auch die Sonderfälle Multithreadbean und Aktive Bean wurden implementiert. Zum ersteren war eine erweiterte Threadklasse zu erstellen.

Die Untersuchung hat gezeigt, dass das Konzept von Rainer Hauck zur Überwachung von Antwortzeiten und Prozessorzeiten baustein-basierter Anwendungen sich mit Javabeans umsetzen lässt.

Die verwendete Beanbox ist ein einfaches Instrument und zur Demonstration der Machbarkeit von zusammengesetzten Javabean-Anwendungen gedacht. Daher traten noch einige Mängel auf: Zum Beispiel kann keine Applikation aus den Beans erstellt werden, nur `Applets`.

Ein weiterer Nachteil bei Verwendung von Standardevents wie `ActionEvent` ist, dass keine Daten zwischen den Beans übergeben werden. Das könnte durch selbstdefinierte Events geschehen, ist aber im Konzept der einfachen Beanbox nicht vorgesehen. In aufwendiger gestalteten Beanboxen sollten aber diese Mängel behebbar sein.

A Anhang A: Installation

A.1 Installation der Beanbox

Voraussetzung für die Installation ist ein Java mindestens Version 1.2, das die Eventversion 1.1 von Java unterstützt ([Bdk11]).

1. **Installation in Linux**

   Mit dem Download der bdk-Datei (Typ Solaris) wird die Datei `bdk1_1-solsparc.bin` übertragen. Sie muss das Ausführrecht erhalten, denn sie beginnt mit einem Skript. Danach wird sie mit ihrem Namen `bdk1_1-solsparc.bin` oder `./bdk1_1-solsparc.bin` gestartet.

   Java muss in einem Verzeichnis der Form `/usr/lib`, `/usr/bin`, `/soft/bin` oder ähnlich vorhanden sein. Ist das Java in einem anderen Verzeichnis enthalten, so muss dieses in die `PATH`-Variable aufgenommen werden.
   Nach der Bestätigung eines geeigneten Verzeichnisses läuft die Installation automatisch (m.H. von `InstallAnywhere`) ab. Die Beanbox wird im Verzeichnis BDK1.1 eingerichtet.

2. **Installation in Windos95**

   Mit dem Download der bdk-Datei (Type Windows) `bdk1_1-win.exe` wird gleich eine Installation in BDK1.1 durchgeführt.

A.2 Installation anwendungsspezifischer Javabeans

Nicht für alle Anwendungen werden die vorgefertigten Javabeans alle Funktionen bereitstellen können. In diesem Fall müssen weitere Javabeans entwickelt werden. Dies ist Aufgabe des Bausteinentwicklers.

Bei der Entwicklung zusätzlicher Javabeans ist zu beachten, dass diese serialisierbar sein müssen, damit sie mit einer Anwendung abgespeichert und wiedergeladen werden können:

- `implements Serializable bzw. Externizable`

Sollen Properties (Daten und Eigenschaften wie Farben) anpassbar sein, dann müssen standardisierte Zugriffroutinen erstellt werden, z.B. für die Variable `status`:

```
int status:
int getStatus()
void setStatus(int status)
```

Javabeans, die von Bean-Entwicklern modifiziert oder neu erstellt wurden, müssen wie Javaprogramme übersetzt und als `jar`-File komprimiert in das Verzeichnis `jars` der Beanbox eingebracht werden. Dazu wird ein `manifest`-File benötigt, z.B.:

```
Name: MuliBean.class
Javabean: true
Depends-On: BThread.class
```

Das Erstellen des `jar`-Files erfolgt dann mit dem Aufruf:

```
jar cfm mulibean.jar manifest.tmp MuliBean.class BThread.class
```

Javabeans, die zugekauft oder aus dem Web geladen wurden, lassen sich über Laden in die Toolbox einführen, wenn sie als `jar`-files vorliegen oder in diese Form gebracht wurden.

A.3 Installation der BThread-Klasse

Die `BThread`-Klasse ist eine Erweiterung der `Thread`-Klasse aus der Javabibliothek, Sie wird hier verwendet, um in Subthreads von Multithread-Javabeans (`MuliBean`) Messpunkte zu setzen:

```
// Instrumentierter Thread fuer Beans mit eigenem Thread (SubThread)
// BThread.java Kalix Fopra

import java.io.Serializable;
import MessBean;

public class BThread extends Thread{
String name;

public BThread(){
   super();
}

public BThread(String name){
   super();
   this.name = name;
}

public synchronized void start( String bName){
     Thread t=Thread.currentThread();
     String mt = t.getName();
     String st = this.getName();
     MessBean adap = new MessBean();
     adap.add_control_flow(mt,st);
     start();
}

public synchronized void stop(  boolean success){
```

```
        Thread t=Thread.currentThread();
        String mt = t.getName();
        String st = this.getName();
        MessBean adap = new MessBean();
        adap.rem_control_flow(mt,st, success);
        stop();
}
}
```

Die Methode `stop` ist deprecated, funktioniert aber. `BThread` soll in einem folgenden Fopra durch die geänderte `Thread`-Klasse ersetzt werden.

A.4 Installation eines alternativen Events

Dies ist nur in einer linux-Umgebung, nicht in Windows95 möglich.
Zuächst werden ein `AlterEvent`-Objekt und ein `AlterListener`-Interface erstellt:

```
public class AlterEvent extends java.util.EventObject{
  AlterEvent(Object source){
     super(source);
  }
}

public interface AlterListener extends java.util.EventListener{
   void alterPerformed(AlterEvent ae);
}
```

Beide werden übersetzt und in `jar`-Files überführt. Letztere werden in das `jars`-Unterverzeichnis gebracht.

Javabeans, die alternative Events feuern, benötigen:

- als Import bzw. Implement:

  ```
  import java.util.Vector;
  import java.util.*;

  implement AlterListener
  ```

- zum An- und Abmelden der Empfangsbeans:

  ```
  public synchronized void addAlterListener(AlterListener l) {
          getListeners.addElement(l);
      }
  ```

```
public synchronized void removeAlterListener(AlterListener l) {
        getListeners.removeElement(l);
    }
```

- zum Eventfeuern eine Routine wie:

```
public void doAction() {
  Vector destis;
  synchronized (this) {
    destis = (Vector) getListeners.clone();
  }
  for (int i=0; i < 20000; i++){
        x = x+i;}

  AlterEvent alterEvt = new AlterEvent(this);
  for (int i = 0; i < destis.size(); i++) {
    AlterListener desti = (AlterListener)destis.elementAt(i);
    desti.alterPerformed(alterEvt);
  }
}
```

- eine Dummy-Empfangsmethode:

```
public void alterPerformed(AlterEvent ae){
       System.out.println("IfelBean: alterPerformed\n");
}
```

- und Daten:

```
private Vector getListeners = new Vector();

private AlterEvent aevt;
```

Javabeans, die alternative Events empfangen, benötigen eine Methode mit dem Argument:

`AlterEvent aevt`

`AlterEvent.class` und `AlterEventListener` müssen in die `manifest`-Files derjenigen Beans aufgenommen werden, die den `AlterEvent` verwenden.

A.5 Installation des Messobjekts

Das Messobjekt ist als Javabean realisiert und wird wie diese installiert.

A.6 Installation einer Anwendung

Die benötigten Javabeans werden als `jar`-Files ins `jars`-Verzeichnis eingebracht. Die Beanbox wird geladen durch den Aufruf

- `run.sh` (Linux) im Verzeichnis `BDK1.1/beanbox` bzw.

- `run` (Windows) in `BDK1.1\beanbox`.

Buttons und gegebenenfalls Beans werden durch Customizing modifiziert, die Änderungen werden mit Abspeichern und Wiederladen persistent gemacht und die Javabeans werden in der (topologisch) richtigen Reihenfolge miteinander verbunden.

Mit `file/create Applet` kann ein Applet der Anwendung erstellt werden.

B Anhang B: MessBean.java

```
// MessBean.java 6.2.01 v45  15:17
// Funktion: Messpunkte von Anwendungen werden aufgenommen und
//           in periodischen Abstaenden in ein Logfile geschrieben
// Autor:    Erika Kalix
// Projekt:  Fopra: Implementierung der Performance-Ueberwachung
//                  Javabean-basierter Anwendungen
// Version 1.0 fuer BThread, StartButton v1.0
// Datum: 06.02.2001
// Uebersetzen: javac MessBean.java
// Als Javabean muss MessBean.java in eine jar-datei verpackt werden:
// jar cfm messbean.jar manifest.ta4 MessBean.class
// Datei manifest.ta4 ist nur fuer Windows95 noetig, wird in Linux mit
// Makefile automatisch erzeugt.
// cp messbean.jar /BDK1.1/jars
// -----------------------------------------------------------------------

import java.util.*;
import java.io.*;
import java.awt.*;
import java.beans.*;
import java.io.Serializable;
import java.lang.Thread;

public class MessBean extends Component implements Runnable {
    static String mT [] = new String [40];
    static int nt;   // Index der TA-Tabelle
    static int ni;   // Index der BTA-Tabelle
    static int nAnf; // aktueller Anfangsindex fuer Schreiben in Logfile
    static int nEnd; // aktueller Endeindex fuer Schreiben in Logfile

// TA-Tabelle fuer Messpunkte:
    static long tifld[] = new long [100];
    static String rifld[] = new String [100];
    static String isfld[] = new String [100];
    static String scfld[] = new String [100];
    static String tgfld[] = new String [100];
    static String okfld[] = new String [100];

// BTA-Tabelle fuer BTA-ID und zugehoerige Threads:
    static String bta_tab [][] = new String [100][2];

    static String BTA_instanz = new String();
    static String mainTh = new String();

    java.io.FileWriter flog;    // Schreiben in logfile
    java.io.PrintWriter log;    // Text an Logfile anhaengen

    private String beanName ;
    private int baseline;
    private String ourLabel = " MessBean  ";
    private transient Thread tm;  // eigener Thread
```

```
// ---------------------------------------------------------------------
// Konstruktor:
// Parameter : keine fuer Beanbox
// ---------------------------------------------------------------------

public MessBean() {
        String tstr = toString();
        int endpos = tstr.indexOf('[');
        beanName = tstr.substring(0,endpos);
}

// ---------------------------------------------------------------------
// run: ewige Schleife: Aufruf von doLog zum Schreiben ins Logfile
//      legt sich schlafen und wird periodisch geweckt, um neue Daten
//      auszugeben
// ---------------------------------------------------------------------

public void run(){
    while (true){
      try{
         Thread.sleep(350);                // schlafen, Wecker setzen
        } catch (InterruptedException ei){ // Weckevent abfangen
        }
         nEnd = nt-1;
         doLog(nAnf, nEnd);  // neue Eintraege ins Logfile schreiben
         nAnf = nEnd+1;      // Anfang neu setzen
     try{
         Thread.sleep(1000);
        } catch (InterruptedException ei){
        }
      } //While
     }


// ----------------------------------------------------------------------

// BTA_start: Eintrag der BTA-Instanz  BTA_ID, MainThread in BTA-Tabelle,
//            ersten Messpunkt speichern
// Parameter: BTA_name: Name der BTA
//            BName:    Name der meldenden Javabean
//            id:       Thread-Id
// ---------------------------------------------------------------------

   public static void BTA_start(String BTA_name, String BName, String id){
      Thread th = Thread.currentThread(); // get Mainthread
      Date myDate = new Date();           // aktuelle Zeit/Datum
      String chtim = Long.toString(myDate.getTime());
      String tid = new String("t"+chtim);
      th.setName(tid);                      // eindeutiger Threadname

      BTA_instanz = new String(BTA_name+tid);// eindeutige BTA-ID
      bta_tab[ni][0] = BTA_instanz;     // Eintrag BTA_ID
```

```
      bta_tab[ni][1] = tid;            // Eintrag mainthreadname
      if (ni < 100) ni++;
      if (ni == 100) System.out.println("Ueberlauf BTA-Tabelle!\n");

      TA_start(tid, BName);
      MessBean log = new MessBean();
      Thread tm  = new Thread(log);
      tm.setPriority(Thread.MIN_PRIORITY);

      tm.start(); // start eigenen Thread zur Logfile-Ausgabe
    }

// -----------------------------------------------------------------

// BTA_end:   BTA-Ende (Ergebnisausgabe) kennzeichnen:
//            Messpunkt in TA-Tabelle schreiben
// Parameter: Bname: Name der meldenden Javabean
//            mainTh: Name des MainThreads
//            success: Erfolg, (false, falls Fehler entdeckt)
// ----------------------------------------------------------------------

 public static void BTA_end(String Bname, String mainTh, boolean success){
      Date ptDate = new Date();
      long ptTime = ptDate.getTime();
      int ptdut = 10000;
                                       // find BTA-Instanz:
      for (int i=0; i<ni;i++){
       if (mainTh.compareTo(bta_tab[i][1]) == 0)
         ptdut = i;
      }
         if (ptdut < 10000){             // BTA-Instanz: gefunden
          isfld[nt] = bta_tab[ptdut][0];
         }
         tifld[nt] = ptTime;
         rifld[nt] = "BTA_end";
         tgfld[nt] = Bname;
         if (success == true) okfld[nt] = "erfolgreich";
         else       okfld[nt] = "nicht erfolgreich";

         if (nt < 100) nt++;
         else
          System.out.println("BTA_end : Ueberlauf TA-Tabelle\n");
    }

// ---------------------------------------------------------------------

// TA_start:  Messpunkt vor Eintritt des Kontrollflusses in eine TargetBean
//            setzen
// Parameter: th : Name des Kontrollflusses
//            t  : Name der TargetBean
// ----------------------------------------------------------------------

  public static void TA_start(String th, String t){
```

```
      Date inDate = new Date();
      long inTime = inDate.getTime();
      int index = 10000;
                                   // find BTA-Instanz:
      for (int i=0; i<ni;i++){
       if (th.compareTo(bta_tab[i][1]) == 0)
         index = i;                       // gefunden
      }

         if (index < 10000){              // gefunden
          isfld[nt] = bta_tab[index][0];}

         tifld[nt] = inTime;
         if (nt > 0)
           rifld[nt] = "TA_start";
         else
           rifld[nt] = "BTA_start";     // 1. Eintrag
         tgfld[nt] = t;
         okfld[nt] = "-"; // success noch nicht bekannt

        if (nt < 100) nt++;
        else
          System.out.println("TA_start : Ueberlauf TA-Tabelle\n");
      }

// -----------------------------------------------------------------

// TA_stop:   Messpunkt nach Austritt des Kontrollflusses in eine TargetBean
//            eintragen
// Parameter: th : Name des  Kontrollflusses
//            t  : Name der TargetBean
//            success: Erfolg, (false, falls Fehler entdeckt)
// -----------------------------------------------------------------------

   public static void TA_stop(String th, String t, boolean success){

      Date utDate = new Date();
      long utTime = utDate.getTime();
      int indut = 10000;

                                  // find BTA-Instanz:
      for (int i=0; i<ni;i++){
       if (th.compareTo(bta_tab[i][1]) == 0)
         indut = i;
      }
         if (indut < 10000){          //  BTA-Instanz:  gefunden
          isfld[nt] = bta_tab[indut][0];}

         tifld[nt] = utTime;
         rifld[nt] = "TA_stop";
         tgfld[nt] = t;
         if (success == true) okfld[nt] = "erfolgreich";
         else      okfld[nt] = "nicht erfolgreich";
```

```
        if (nt < 100) nt++;
        else
          System.out.println("TA_stop : Ueberlauf TA-Tabelle\n");
      }

// ------------------------------------------------------------------

// add_control_flow: Neuen Thread bei Start eines SubThreads eintragen:
//                   Messpunkt setzen
// Parameter: mainth : Name des  Hauptkontrollflusses
//            netTh  : Name des neuen Kontrollflusses
// last update: 6.2.01 Version fuer BThread, StartButton v1.0
// ----------------------------------------------------------------------

    public static void add_control_flow(String mainth, String newTh){

    // mainth kann fuer jdk aus Schnittstelle entfernt werden, Aenderung in
    // BThread  noetig

      int m=0;
      boolean found = false;

      Thread th = Thread.currentThread();  // HauptThread ermitteln
      String mainTh = th.getName();

      for (int j=0;j < ni; j++){            // find BTA-ID
        if (mainTh.compareTo(bta_tab[j][1]) == 0) {
          m = j;
          found = true;
        }
      }
      if (found){                       // Eintrag in BTA-Tabelle:
        bta_tab[ni][0] = bta_tab[m][0];
        bta_tab[ni][1] = newTh;
        if(ni < 100)       ni++;
        else
          System.out.println("add_control_flow : Ueberlauf TA-Tabelle\n");
      }

      TA_start(newTh, "dummy"); // Messpunkt in TA-Tabelle eintragen
    }

// -----------------------------------------------------------------

// rem_control_flow: Messpunkt bei Ende eines SubThreads setzen
//                   SubThread aus BTA-Tabelle austragen
// Parameter: mainth : Name des  Hauptkontrollflusses
//            netTh  : Name des neuen Kontrollflusses
//            success: Erfolg, (false, falls Fehler entdeckt)
// last update: 6.2.01 Version fuer BThread, StartButton v1.0
// ----------------------------------------------------------------------
```

```
    public static void rem_control_flow(String mainth, String newTh,
                                        boolean success){
    // mainth kann fuer jdk aus Schnittstelle entfernt werden, Aenderung in
    // BThread und StartButton noetig


      Thread th = Thread.currentThread();  // HauptThread ermitteln
      String mainTh = th.getName();

      if (mainTh.compareTo(newTh) != 0)     // nicht MainThread
        TA_stop(newTh, "dummy", success);   // Subthread
      else
        TA_stop(newTh, tgfld[0], success);  // Startbutton 1.Eintrag

      int m=0;
      boolean found = false;    // find BTA-ID zu MainThread

      for (int j=0; j<ni; j++){
        if (mainTh.compareTo(bta_tab[j][1]) == 0) {
          m = j;
          found = true;
        }
      }

      if (found){
        String BTA_id = bta_tab[m][0];
        found = false;
        for (int j=0;j<ni;j++){             // BTA_ID zu Subthread suchen
          if ((BTA_id.compareTo(bta_tab[j][0]) == 0) &&
              (newTh.compareTo(bta_tab[j][1]) == 0)) {
              m = j;
              found = true;
          }
         }
       }
       if (found){        // Subthread aus BTA-Tabelle austragen
         bta_tab[m][0] = "";
         bta_tab[m][1] = "";
      }
    }
// -------------------------------------------------------------------

// ATA_start:  Messpunkt bei Start eines AktivBean-Auftrags setzen:
// Parameter: BTAid  : BTA-Id der BTA, die Auftrag an Aktive Bean gab
//            newTh  : Name des neuen Kontrollflusses
// ------------------------------------------------------------------------

    public static void ATA_start(String BTAid, String newTh){

       bta_tab[ni][0] = BTAid;   // BTA_ID eintragen in BTA-Tabelle
       bta_tab[ni][1] = newTh;   // neuen Thread eintragen in BTA-Tabelle
       if(ni < 100)       ni++;
       else
```

```
            System.out.println("ATA_start : Ueberlauf BTA-Tabelle\n");

        Date inDate = new Date();
        long inTime = inDate.getTime();

        isfld[nt] = BTAid;
        tifld[nt] = inTime;
        rifld[nt] = "ATA_start";
        tgfld[nt] = "dummy";
        okfld[nt] = "-";

         if (nt < 100) nt++;
         else
              System.out.println("ATA_start : Ueberlauf TA-Tabelle\n");
      }

// ---------------------------------------------------------------------------

// ATA_stop: Messpunkt bei Ende des von der aktiven Bean ausgefuehrten
//           Auftrags setzen
// Parameter: BTA_ID  : BTA-Id der BTA, die Auftrag an Aktive Bean gab
//            newTh   : Name des neuen Kontrollflusses
//            success: Erfolg, (false, falls Fehler entdeckt)
// ---------------------------------------------------------------------------

    public static void ATA_stop(String BTA_ID, String newTh,
                                          boolean success){

        Date utDate = new Date();    // Zeitpunkt ermitteln
        long utTime = utDate.getTime();
        isfld[nt] = BTA_ID;          // Eintag in Ta-Tabelle
        tifld[nt] = utTime;
        rifld[nt] = "ATA_stop";
        tgfld[nt] = "dummy";
        okfld[nt] = "-";


        if (nt < 100) nt++;
        else
           System.out.println("ATA_stop : Ueberlauf TA-Tabelle\n");

                                    // BTA_ID aus bta-tab austragen:
        int m=0;
        boolean found = false;      // find BTA-ID in bta_tab

        for (int j=0; j<ni; j++){
          if ((BTA_ID.compareTo(bta_tab[j][0]) == 0) &&
                (newTh.compareTo(bta_tab[j][1]) == 0)){
            m = j;
            found = true;
          }
        }
```

```
      if (found){
         bta_tab[m][0] = "";  // BTA_ID austragen
         bta_tab[m][1] = "";  // zugehoerigen ATA-Thread austragen
      }
   }
// --------------------------------------------------------------------------

// logInfo:   Annahme eines Info-Strings, Schreiben in TA-Tabelle
// Parameter: info: String mit Informationen
//            target: Name der meldenden Javabean
// --------------------------------------------------------------------------

   public static void logInfo(String info, String target){
                                              // get Mainthread, Date, :
      Thread th = Thread.currentThread();
      String thName = th.getName();
      int indut = 10000;

      Date imDate = new Date();
      long imTime = imDate.getTime();

                                              // find BTA-Instanz
      for (int i=0; i<ni;i++){
       if (thName.compareTo(bta_tab[i][1]) == 0)
         indut = i;
      }
      if (indut < 10000)  // MainThread in BTA-Tabelle gefunden
      {
         isfld[nt] = bta_tab[indut][0];  // Eintragen in TA-Tabelle

         tifld[nt] = imTime;
         rifld[nt] = "Info: "+info;
         tgfld[nt] = target;
         okfld[nt] = "-";

        if (nt < 100) nt++;
        else
            System.out.println("logInfo : Ueberlauf TA-Tabelle\n");
       }
   }
// ---------------------------------------------------------------------

// getBTA_ID: Zurueckgeben der BTA-Instanz an Aktive Bean
// Parameter: keine
// returns  : Instanzname  der BTA
// --------------------------------------------------------------------------

  public static String getBTA_ID(){

                                             // find Mainthread
      Thread th = Thread.currentThread();
      String thName = th.getName();
      int indut = 10000;
```

```
                                        // find BTA-Instanz:
      for (int i=0; i<ni;i++){
       if (thName.compareTo(bta_tab[i][1]) == 0)
         indut = i;
      }
         if (indut < 10000)          // Thread gefunden
           return bta_tab[indut][0];
         else return "ERROR";

      }

// -------------------------------------------------------------------

// doLog:    Schreiben der Messdaten in ein Logfile
//           Diese Methode wird zyklisch in der run-Methode aufgerufen
// Parameter: j: aktueller Anfangsindex
//            m: aktueller Endeindex
// ------------------------------------------------------------------------

 private void doLog(int j, int m){
    if (m>j){            // EndeIndex > Anfangsindex
    if (log == null) {    // noch kein Ausgabefile-ID
      try {
       java.io.FileWriter flog = new java.io.FileWriter("kxbea",true);
       log = new java.io.PrintWriter(flog, true); // Text anhaengen
      } catch (IOException ioe) {
         System.err.println(ioe.getMessage());
      }//try
    } //if

    for(int i = j; i <= m; i++){ // in logfile schreiben
      log.println(isfld[i]);
      log.println(rifld[i]);
      log.println(tifld[i]);
      log.println(okfld[i]);
      log.println(tgfld[i]+"\n");
   }
  }// if m>j
 } //doLog
}   //class
```

Literaturverzeichnis

[Bdk11] *Beanbox Download.* Technischer Bericht, Sun Microsystems. http://java.sun.com/products/javabeans/software/bdk-download.html.

[C807] *Application Response Measurement (ARM) API* . Technical Standard C807, ACM Transactions on Graphics, Juli 1998.

[Hauc01] HAUCK, RAINER: *Architektur für die Automatisierung der Managementinstrumentierung bausteinbasierter Anwendungen* . Dissertation, LMU, 2001. To be published.

[Jb101] HAMILTON, GRAHAM (ED.): *JavaBeans(TM) API Specification.* Specification, Sun Microsystems, Juli 1997. http://java.sun.com/beans/glasgow.

[Klui00] KLUIT, ONNO: *JavaBeans Technology: Unlocking the BeanContext API* . Technischer Bericht, Juli 2000. http://java.sun.com/developers/technicalArticles/jbeans/BeanContext/index.html.

[Quin00] QUINN, ANDY: *Trail: JavaBeans(TM).* Tutorial, Sun Microsystems, 2000. http://java.sun.com/beans/docs/books/tutorial/javabeans.