

Monitoring Software Components and Component-Based Software

Jerry Gao, Ph.D., Eugene Y. Zhu, Simon Shim, Ph.D.

San Jose State University
One Washington Square
San Jose, CA 95192-0180
Email:gaojerry@email.sjsu.edu

Abstract

Component engineering is gaining substantial interest in the software engineering community. A lot of research efforts have been devoted to analysis and design methods for component-based software. However, only few papers address the testing and maintenance problems of component-based software. This paper discusses component traceability and maintenance issues and solutions in supporting software components of component-based software. This paper proposes a Java framework and a systematic approach to support tracking and monitoring software components in component-based programs. Application examples and the supporting system are described. Moreover, the paper introduces the concept of traceable components, including requirements, design guidelines, and architecture style. The presented results are useful to add systematic component tracking features into the current Java and EJB technology to support software components, including third-party components in software maintenance.

Keywords: Component engineering, software maintenance, software tracking, software engineering, component-based software.

1. Introduction

As the rapid increase of software programs in the size and complexity, it is very important to reduce high software cost and complexity while increasing reliability and modifiability [8]. With the advances of Internet technology, more distributed systems are built to meet diverse application needs. Currently component engineering is gaining substantial interest in the software-engineering community. As more third-party software components are available in the commercial market, more software workshops start to use the component engineering approach

to developing component-based programs for distributed applications.

Although there are many published articles addressing the issues in building component-based programs, very few papers address the problems and challenges in testing and maintenance of software components and distributed component-based programs [11][9][2][10].

Today, engineers in the real world have encountered a number of issues in testing and maintenance of component-based software [2][9][10]. Some of them related to component traceability and program tracking [2]. They are given as follows.

- *Hard to understand component behaviors in a system.* In system testing and maintenance, system testers usually have the difficulty in understanding and monitoring the component behaviors in a system due to the following reasons:
 - Engineers use ad hoc mechanisms to track the behaviors of in-house components. This has caused problem for system testers to understand the behavior of a system due to inconsistent trace messages, formats, and diverse tracking methods.
 - No built-in tracking mechanisms and functions in third-party components for monitoring their external behaviors.
 - No configuration functions for component clients to control and configure the built-in tracking mechanisms.
 - No systematic methods and technologies to control and monitor the external behaviors of the components.

- *Difficulty on component error isolation, tracking and debugging.* In a system, components that are developed by different teams may use different tracking mechanisms and trace formats. Thus, inconsistent tracking mechanisms and trace messages cause the difficulty in error detection and isolation of components.
- *High costs on performance testing and tuning for components.* Performance testing for component-based programs is a major challenge in system testing due to the fact that current component vendors usually do not provide users with any performance information. Hence, system testers and integration engineers must spend a lot of efforts to identify the performance problems and the components that cause the problems.
- *Lack in system resource validation for components.* Since most components do not provide their system resource information, it is difficult for system testers to locate the system resource problems in system testing and maintenance.

Therefore, there are two major challenges in developing distributed component-based software. The first is to design software components with consistent mechanisms and interfaces to support the tracking and monitoring of component behaviors, functions, performance, and resources. The other is to develop a systematic method and environment to monitor and analyze component behaviors and system performance in a distributed environment.

This paper addresses the component traceability issues in supporting of component-based programs. It discusses the component traceability concept, requirements, challenges, and evaluation criteria. Moreover, it examines different program tracking mechanisms. In the paper, we introduce the concept of traceable components, and propose a new tracking mechanism called event-based tracking model. We propose a Java framework and a systematic mechanism for tracking and monitoring various software components in a component-based program. Our application examples and support system show the potential advantages in supporting both in-house and third party components in component-based programs.

The structure of the paper is organized as follows. Section 2 discusses the perspectives of component traceability in terms of component behaviors, interfaces, performance, events, and status. In addition, it discusses the concept of traceable components, including tracking requirements and evaluation criteria. Moreover, it examines different mechanisms to increase component traceability. Section 3 it presents our systematic tracking solution to support software components. In this section, a new Java-based framework for tracking is provided, and the supporting environment is described. Section 4 discusses the involved design and implementation issues and our solutions. Application examples are given to demonstrate the basic idea and advantages of our solution. Moreover, we discuss the open issues on monitoring and supporting of software components in distributed environments by relating to the existing work. Finally, the conclusion remarks are given in Section 5.

2. Understanding of Component Traceability

According to IEEE Standard Directory of Electrical & Electronics Terms, "tracking" refers to the process of following a moving object or a variable input quantity, using a servomechanism [3]. A trace routine refers to a program routine that provides a historical record of specified events in the execution of a program.

Software tracking includes a) *project tracking*, b) *product tracking*, and c) *program tracking*. *Project tracking* keeps tracking of important project schedules, activities and events during the software development process. It is a fundamental activity in software management [6][5]. *Product tracking* refers to control and monitor software product in a systematic and manageable way [7]. It is the essential task in configuration management. *Program tracking* refers to the activities and effort on tracking various factors of a program, including its input data, output results, and behaviors. It is an effective way to support engineers in program understanding, debugging, software testing and maintenance.

Component Traceability

According to Schmauch Chareles H., "traceability" refers to the ability to show, at any time, where an item is, its status, and where it has been [8].

"Traceability" of a software component refers to the extent of its build-in capability of tracking the status of component attributes and component behavior. Traceability of a component includes the following two aspects:

- *Behavior traceability* - It is the degree to which a component facilitates the tracking of its internal and external behaviors. There are two ways to track component behaviors. One is to track internal behaviors of components. This is useful for white-box testing and debugging. Its goal is to track the internal functions, internal object states, data conditions, events, and performance in components. The other one is to track external behaviors of components. It has been used for black box testing, integration, and system maintenance. The major purpose is to track component public visible data or object states, visible events, external accessible functions and the interactions with other components.
- *Trace controllability* - It refers to the extent of the control capability in a component to facilitate the customization of its tracking functions. With trace controllability, engineers can control and set up various tracking functions such as turn-on and turn-off of any tracking functions and selections of trace formats and trace repositories.

We can classify component traces into five types:

(1) *Operational trace* - It records the interactions of component operations, such as function invocations. It can be further classified into two groups: a) *internal operation trace* that tracks the internal function calls in a component, and b) *external operation trace* which records the interactions between components. *External operation trace* records the activities of a component on its interface, including incoming function calls, and outgoing function calls.

(2) *Performance trace* - It records the performance data and benchmarks for each function of a component in a given platform and environment. *Performance trace* is very useful for developers and testers to identify the performance bottlenecks and issues in performance tuning and testing. According to performance traces, engineers can generate a performance metric for each function in a

component, including its average speed, maximum, and minimum speed.

(3) *State trace* - It tracks the object states or data states in a component. In component black box testing, it is very useful for testers to track the public visible objects (or data) of components.

(4) *Event trace* - It records the events and sequences occurred in a component. The event trace provides a systematic way for GUI components to track GUI events and sequences. This is very useful for recording and replay of GUI operational scenarios.

(5) *Error trace* - It records the error messages generated by a component. The error trace supports all error messages, exceptions, and related processing information generated by a component.

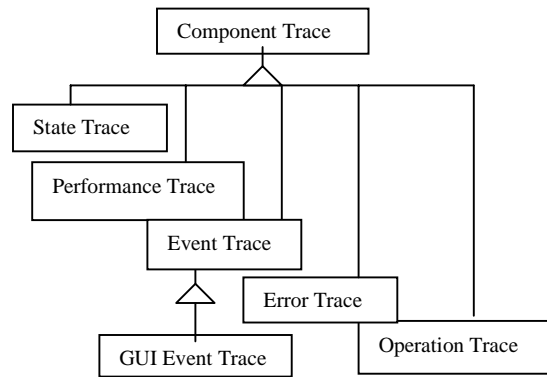


Figure 1. Different Types of Component Traces

Mechanisms to Increase Component Traceability

Since a component-based program is an integration of software components, its program traceability depends on the traceability of its components. Component observability depends on component traceability [2]. In the real word practice, we have used ad hoc mechanisms or pre-defined facilities to add tracking code into programs to increase program understandability. However, we have encountered the difficulty on supporting diverse in-house components and commercial components (COTS). This section first discusses three different systematic tracking mechanisms and examines their pros and cons.

Table 1 lists three basic approaches to add a consistent tracking capability into software components to increase component traceability, and to enhance program.

Method 1: Framework-based tracking – In this approach, a well-defined tracking framework (such as a class library) is provided for component engineers to add program tracking code according to the provided programming reference manual. It usually is implemented based on a trace program library. Component engineers can use this library to add tracking code into components. This approach is simple and flexible to use. It can be used to support all trace types, especially for error trace and GUI trace. However, there are several drawbacks. Firstly, it requires a high programming overhead. Secondly, it relies on engineers' willingness to add tracking code. Moreover, this approach assumes that component source code is available. Therefore, it is difficult to deal with commercial components (COTS) because usually they do not provide any source code to clients.

Table 1. Comparisons of Different Tracking Mechanisms

Tracking Perspectives	Framework-Based Code Insertion	Automatic Code Insertion	Automatic Component Wrapping
Source Code	Needed	Needed	Not needed
Code Separation	No	No	Yes
Overhead	High	Low	Low
Complexity	Low	Very High	High
Flexibility	High	Low	Low
Applicability	All types	OP trace, performance trace	OP trace, performance trace
Applicable Components	In-house components	In-house components	In-house components and COTS

Method 2: Automatic code insertion – This approach is an extension of the previous one. Besides a tracking framework, it has an automatic tool, which adds the tracking code into component source code. A parser-based tracking insertion tool is a typical example. To add operational tracking code, it inserts the operation tracking code into each class function at the beginning and the end of its functional body to track the values of its input data and output parameters. Similarly, it can insert the operation tracking code before and/or after each function call. Performance tracking code can be added into a component in the same way to track the performance of each function. Although this approach reduces a lot of programming overhead, it has its own limitations. First, this approach assumes

that component source code is available. This causes the limitation of using it on commercial components (COTS) due to the lack of source code. Next, it is not flexible to insert diverse tracking code at any place in components due to its automatic nature. The other problem is its complexity of the parser tool. Since a component-based program may consist of components written in different languages, this causes high complexity and costs on building parser tools.

Method 3: Automatic component wrapping - This approach is another extension of the first one. Unlike the second method where tracking code is inserted by parsing component source code, this approach adds tracking code to monitor the external interface and behaviors of components by wrapping them as black boxes. The basic idea is to wrap every reusable component (or third-party component) with tracking code to form an observable component in the black box view. With the tracking code, engineers can monitor the interactions between a third-party component and its application components. This approach is very useful to construct component-based software based on third-party software components, for example, EJB. Compared with the other two methods, it has several advantages. One of the advantages is its low programming overhead. In addition, it separates the added tracking code from component source code. Since no source code is required here, this method can be used for both in-house reusable components and commercial components (COTS). However, it is not suitable to support error tracking and state tracking because they are highly dependent on the detailed semantic logic and application domain business rules.

It is clear that each approach has its own pros and cons. In real practice, we need to use them together to support different types of tracking for a program and its components. To design and construct traceable components, engineers need more guidelines on component architecture, tracking interface, and supporting facilities. They encounter the following challenges:

- (1) How to design and implement traceable and observable components in a distributed system?
- (2) How to provide a well-defined tracking framework and effective tracking mechanisms with minimum programming effort and system overhead?

(3) How to support and monitor the behaviors of COTs in component-based software?

3. A Systematic Tracking Solution

This section provides a systematic solution to solve the above problems in the development of component-based software. Our solution consists of three parts: a) a well-structured Java tracking package for component developers, b) an event-based tracking model, and c) a support environment for component-based software.

Java Event-Based Tracking Model

The event-based tracking model is a systematic mechanism that supports engineers to monitor and check the behaviors of components and their interactions in a component-based program. Its basic idea is influenced by Java event model for GUI components. We consider all software components and their elements as *tracking event sources*.

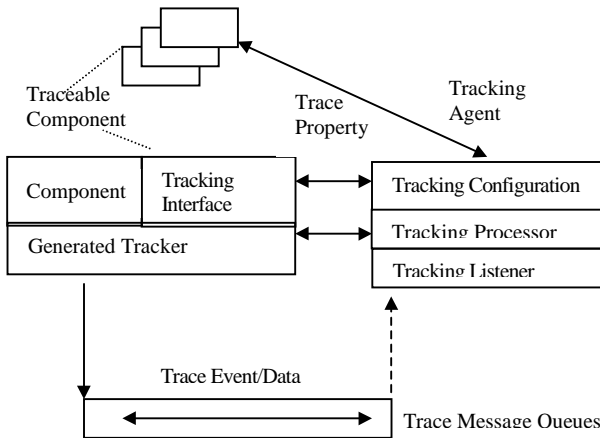


Figure 2. Structure of Event-Based Tracking Model

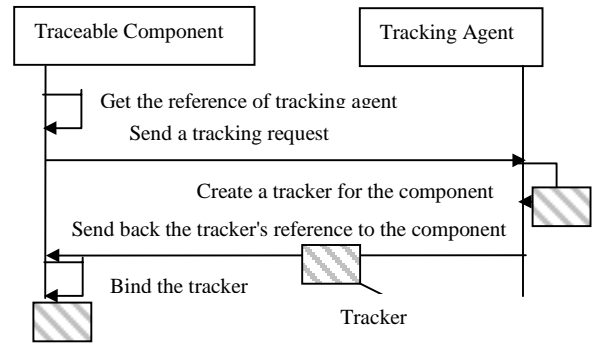
In a software component, component engineers or an automatic tracking tool can add built-in tracking code to trigger five types of tracking events. They are performance tracking, operational tracking, error tracking, state tracking, and GUI tracking events. These events are packed as tracking event messages, and added into trace message queues according to their types.

To catch different tracking events, we use a *tracking listener* to receive these events, dispatch them to a

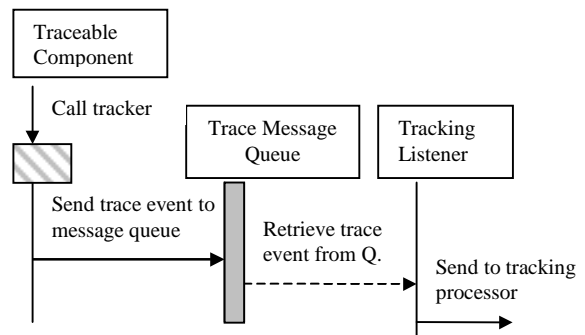
tracking processor to generate the proper trace, and store them into a specified trace repository.

As shown in Figure 2, the event-model tracking mechanism relies on a *tracking agent* to support a collection of *traceable components* in a computer. Intuitively, a *traceable component* is a software component, which is designed to facilitate the observation and monitor its behaviors, data and object states, function performance, and interactions to others. In our solution, a traceable component contains two extra parts other than its normal functional parts:

- A *tracking interface*, which is used to set-up the connection with the *Tracking Agent*. Figure 3(a) shows a procedure to dynamically generate a plug-in tracker for a component by issuing a binding request to the *Tracking Agent* through the *tracking interface*. ***ITraceableBean*** in Figure 4 shows a tracking interface in our Java tracking package.
- A *dynamic generated tracker*, which is dynamically generated by the *Tracking Agent*. Each component receives one *Tracker* after it connected to the Agent. Since developers can use the generic interfaces to provide various tracking functions for different trace types. ***IBeanTracker*** in Figure 4 shows the details of the generic interfaces for tracking functions.



(a) Establish Connection



(b) Exchange Trace Data

Figure 3 Interaction Sequences

```

public interface IBeanTracker {
    public void traceError(String errorMsg);
    public void traceInterface(String methodName, String paramvalue);
    public void traceOperation(String methodName);
    public void tracePerformance(String label);
    public void tracePerformance(String label,boolean isBegin);
    public void traceState(String name, String value);
    public EventListener genGUIListener();
    public void traceGUI(String itemname, String event);
}

public interface ITRAgent extends ItemListener {
    // dynamically generate and return a plug-in tracker
    public IBeanTracker createTracker(String beanName);
}

public interface ITraceableBean {
    // function to store the reference to the plug-in tracker
    public void bindBeanTracker();

    // expose the trace properties
    public void getTraceProperties();
}

```

Figure 4. Tracking Package

```

public class BeanTrackerAdapter implements IBeanTracker {
    public void traceError(String errorMsg) {}
    public void traceInterface(String methodName, String paramvalue){}
    public void traceOperation(String methodName) {}
    public void tracePerformance(String label) {}
    public void tracePerformance(String label,boolean isBegin) {}
    public void traceState(String name, String value) {}
    public EventListener genGUIListener() {return (EventListener)
        (new GUIListenerAdapter());}
    public void traceGUI(String itemname, String event) {}
}

class GUIListenerAdapter extends Object
implements WindowListener, ActionListener, TextListener {
    public GUIListenerAdapter() {}

    // implement WindowListener interface
    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}

    // implement ActionListener interface
    public void actionPerformed(ActionEvent e) {}

    // implement TextListener interface
    public void textValueChanged(TextEvent e) {}
}

```

Figure 5. Adapter for the Plug-in Tracker

A *tracking agent* consists of the following three functional parts.

- *Tracking Listener* is a multi-thread program that listens and receives all types of tracking events through trace message queues, and dispatches them to *tracking processor*. Figure 3(b) shows the interaction sequences among a traceable component, tracking listener, and tracking processor

- *Tracking Processor* generates program traces according to a given trace event based on its trace type, trace message and data, and stores them in the proper trace repository.

- *Tracking Configuration* provides a graphic user interface to allow a user to discover and configure various tracking features for each traceable component.

Java Tracking Package

Current component-based technologies (such as JavaBean, EJB and CORBA) do not provide developers with systematic mechanisms and facilities to track and monitor component behaviors. Thus, developers only can use ad-hoc methods to construct traceable components. Our *Java Tracking Package* provides developers with several generic interfaces for constructing traceable Java

components. It includes

- *An interface for a component tracker* (see *IBeanTracker* in Figure 4). This interface provides the generic tracking function interface between a component *tracker* and the tracking agent. Various tracking events and requests can be passed to the *tracking agent*.
- *Agent interface* (see *ITRAgent* in Figure 4). It is a part of *tracking agent*. Using this interface, developers can issue a request to the *Tracking Agent* for a component to create a plug-in *tracker* and set-up their connection.
- *Traceable component interface* (see *ITraceableBean* in Figure 4). This interface allows developers to bind a plug-in tracker for a component. Figure 6 shows a sample implementation for a *bindBeanTracker*. The function `getTracePropertie()` can be used to discovers the trace properties of a component.
- *Tracker adaptor interface* (see *BeanTrackerAdaptor* and *GUIListenerAdaptor* Figure 5). This interface is provided to a dummy *tracker* for the case that a component tracking environment is not set up.

```
public void bindBeanTracker() {
    // the following code obtain the tracker
    Properties env = System.getProperties();

    try {
        // get the name of the tracking agent class
        String AgentClassName =
env.getProperty("TrackingAgentName");
        // initiate the tracking agent class
        Class AgentClass = Class.forName(AgentClassName);
        // obtain the static method "getTRAgent"
        Method AgentMethod =
AgentClass.getMethod("getTRAgent", null);
        // obtain the reference to the tracking agent
        ITRAgent trAgent = (ITRAgent) AgentMethod.invoke(null, null);
        // request the agent to create and return the tracker
        tracker = trAgent.createTracker("VBankBean");
    }
    catch (Exception e)
    {
        // bind the default adapter to the tracker
        tracker = new BeanTrackerAdapter();
    }
}
```

Figure 6. A sample implementation of bindBeanTracker

Tracking Environment for Component Software

As shown in Figure 7, we developed a support environment for component-based software. It consists of a number of *tracking agents* and a *tracking server*. Each machine on the network has a multiple threading *tracking agent* based on the *EJB* technology. It interacts with the plug-in trackers of components on the same machine using trace message queues (in Java Message Queue).

In a *tracking agent*, a thread controls, records, and monitors diverse component behaviors in an asynchronous mode. On the other hand, the tracking agent communicates a tracking server to pass trace data in a given trace format. The tracking server plays a role of a central server to allow engineers to control tracking agents to collect diverse trace data and analyze them.

Java JDK.1.2.2 is used to create our tracking agent and tracking server. The distributed supporting environment for JavaBean components is set up based on an EJB server (JonAS). The Java Application Server (JonAS) is the BullSoft implementation of EJB specifications. In addition, Java Message Queue (JMQ) is used to perform the asynchronized communications between JavaBeans and a tracking agent. To create the trace data repository, we use InstantDB, a 100% Java database server from Relational Database Management System (RDBMS).

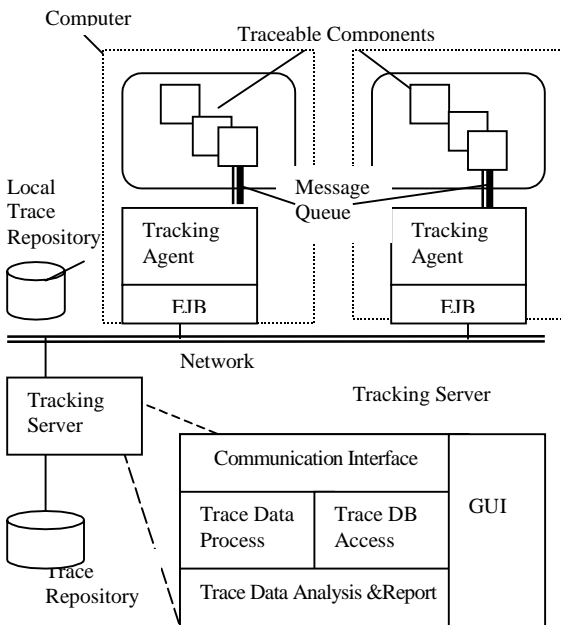


Figure 7. Distributed Tracking Environment

Trace Format

Using a standardized trace format is essential to generate consistent trace messages for each component in a distributed program. A well-defined trace format facilitates the generation of understandable trace messages. This increases component understanding, and helps error isolation and bug fixing. Our distributed tracking environment supports two types of information:

- *Trace commands* - Trace commands support the tracking server to control and communicate with tracking agents in the distributed environment. Each command includes command message ID (00), command code, time stamp, and command parameters.
- *Trace data* - Trace data include message ID (01), trace type, time stamp, component id, trace data, and trace identifier. Trace data indicates where and when the trace message is generated. A trace identifier includes component identifier, thread identifier, and object identifier. Each trace type has its specific trace data format. Tracking agents generate diverse trace data for each traceable component in a local trace repository.

4. Discussions and Summary

This section discusses several important design issues involved in our solution. They are tracking code insertion, tracking overhead reduction, technology flexibility, third-party component support, and distributed monitoring and support for components.

Tracking Code Insertion

Developers can use our solution to add tracking code into a Java component (such as EJB components, Java components or Java Beans) in two ways: automatic insertion, and user insertion. Component developers can add tracking code manually using the tracking package. Figure 8 demonstrate how to add tracking code in a *Login(..)* function for a Java class, called *VbankBean*. Three types of trace code are inserted. They are performance trace, error trace, operation trace, and interface trace. Figure 10 shows the trace results in a tracking agent. Figure 9 demonstrates how to add

```

import beantracker.tracker.*;
.... // other import package

public class VbankBean extends Object
    implements SessionBean, ItraceableBean {
    ....
    private IBeanTracker tracker;
    private boolean trackerloaded = false;
    .... // other attributes
    public boolean Login(String user, String pass) throw
    RemoteException {
        tracker.tracePerformance("Login", true);
        tracker.traceInterface("Login", user+"&"+"pass);
        tracker.traceOperation("Login");
        .... // function body
        .... try
        { .... }
        catch (Exception e2)
        { System.out.println("Cannot create Account:"+e2);
          tracker.traceError("Create account fail."); ...
        }
        tracker.tracePerformance("Login", false);
        return true;
    }
}

```

Figure 8. Tracking Code Example

```

import beantracker.tracker.*;
.... // other import package

public class loginWin extends Frame {
    public TextField txtLogin = new TextField(12);
    public JpasswordField txt Pwd = JpasswordField(12);
    public loginWin (EventListener parent, IBeanTracker tracker)
    { super("Login");
      WindowListener wlis =
      (WindowListener)tracker.genGUIListener();
      ActionListener alis =
      (ActionListener)tracker.genGUIListener();
      TextListener tlis =
      (TextListener)tracker.genGUIListener();
      Button cmdOK = new Button ("Login");
      Button cmdCancel = new Button ("Cancel");
      cmdOK.addActionListener(alis);
      cmdCancel.addActionListener(alis);
      textLogin.addTextListener(tlis);
      ....// GUI layout code
      cmdOK.addActionListener((ActionListener)parent);
      cmdCancel.addActionListener((ActionListener)parent);
      // set size and location
      addWindowListener(WindowListener parent);
      addWindowListener(wlis);
    }
}

```

Figure 9. GUI Tracking Example

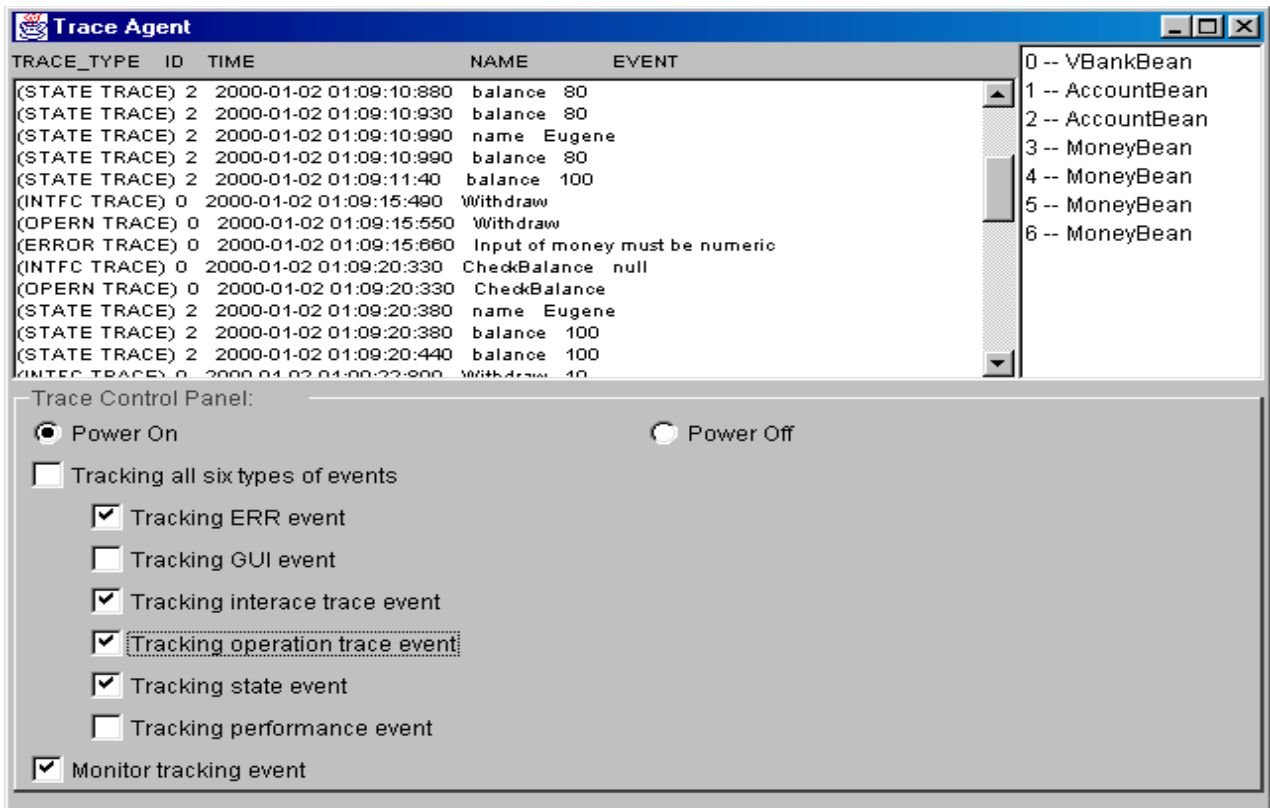


Figure 10. Trace results in a Tracking Agent.

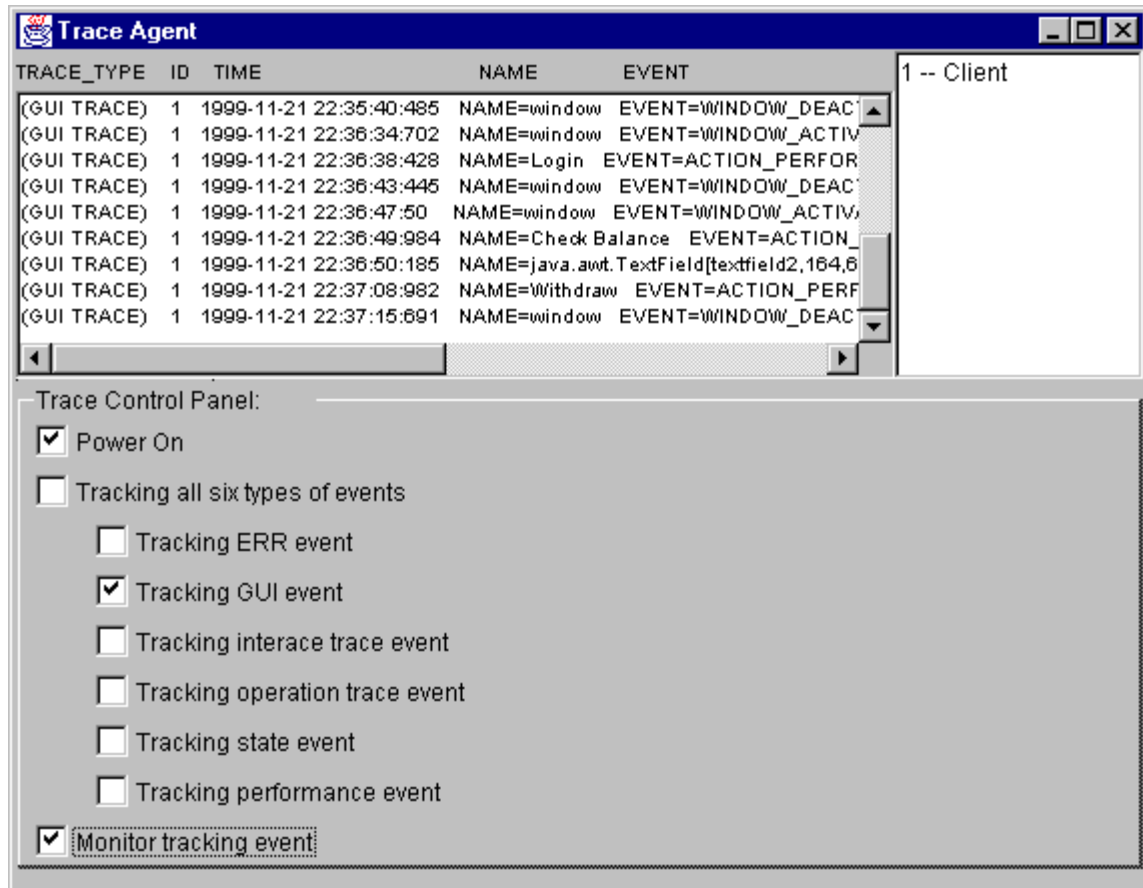


Figure 11. GUI Trace results in a Tracking Agent

GUI tracking code to monitor various GUI events in a Java-based a login Window frame (loginWin class). Figure 11 shows the GUI trace results in a *Tracking Agent*. It is clear that the trace codes (except GUI trace and error trace code) can be used to wrap third-party components to monitor their external behaviors, such as operations, interfaces, and functional performance. After we examine the source code of the EJB server (JonAS), we found that the presented mechanism can be added into the deployment tool of its container in JonAS to implement automatic tracking code insertion using the wrapping approach.

Overhead Reduction:

There are three types of overheads involved in the component tracking: programming overhead, performance overhead, and code overhead. How to reduce these overheads is very important. In our design, we minimize tracking overheads in components by using *Tracking Agent* to handle

the detailed tracking work. Thus, the only tracking overhead in components is the execution of tracker. In addition, asynchronous communication is carried between *Tracking Agent* and components' *Tracker* to reduce the delay of tracking code.

Besides, we point out other three useful methods to reduce the tracking overhead involved in normal component execution:

- Add tracking code with a prefix (such as `////` in Microsoft Visual C++) and enhance the current Java development environment to support compilation of tracking code to generate two different versions. One version contains the tracking code, and the other does not.
- Developers can customize the `bindBeanTracker()` function to bind to a default dummy `tracker` using `BeanTrackerAdapter`. Under this setting, no real tracking function codes are really

executed. The only extra overhead is the involved function invocations of these empty tracking functions in the Adapter classes. Figure 5 shows an example to bind to the default adapter.

- System supporters and developers can use the tracking configuration user interface offered in the *Tracking Agent* to turn-on or turn-off the tracking features for a component. When a tracking feature is turned-off, the only overhead is involved in the condition checking.

Advantages:

The presented solution has several important advantages:

- It is flexible to allow component users to control and configure the monitor and tracking capability of various behaviors in components. Figure 10 and 11 shows the corresponding graphic user interface.
- The proposed tracking package provides a well-structure tracking interface, which separates the implementation of tracking solution and trace formats and traceable components. This implies that a software workshop can define and customize the *Tracking Agent* and trace formats according to its technology and environment requirements. For example, they can implement trace message queues using sockets (or pipes) instead of Java Message Queue.
- It can be used to monitor the external behaviors of third-party components, including functional performance, external operations, and external interfaces.
- The two-level tracking architecture based on *Tracking Agent* and *Tracking Server* provides the scalability. For example, *Tracking Agent* on a machine is set up as a multithreading program to support many traceable components. Each thread is a lightweight process communicating with one component's tracker. In addition, the *Tracking Server* provides distributed to monitor components in a distributed environment.

Other Issues:

There are several other issues relating to monitoring of component behaviors in distributed environment: clock synchronization, event ordering, behavior and performance analysis. In the past, there is a number of published research papers addressing the issues in monitoring and debugging of distributed programs and processes [12][13][14][15][16]. Their focus is

on monitoring distributed processes and events at the process level.

Due to the lack of a global clock in a distributed environment, we need to establish temporal relationships between occurrences of system activity. Three primary methods have been used to deal with this issue. The first involves the use of local clocks and timestamps to create a partial ordering among events[13][14]. The second creates a global snapshot by gathering and grouping of state information in the entire system [15][16]. The third uses a hardware module to solving this problem [12]. Although these methods can be applicable to component-based software in a distributed environment, the last two approaches may not be practical and flexible to deal with the complicated object interactions in component-software. In our solution, we use the local time and timestamp to find out the partial ordering in the event traces. Unlike other approaches, we focus on Java-based software components and provide engineers a well-defined event-tracking framework to construct traceable components. A flexible supporting environment (including *Tracking Agent*, *Tracking Server* and *Message Queues*) are developed based on the tracking framework.

How to understand and analyze component behaviors, interactions, relationships, and performance in a distributed environment is another open issue. Although we have seen existing research work and supporting environments addressing the maintenance issues of traditional programs and object-oriented programs [18][19][20][21]. These systems help engineers understand class relationships, object dependence, and program flows. Unlike traditional distributed systems, the performance and behavior analysis and behavior analysis of software components in a distributed environment is difficult due to the involvement of third-party components. Engineers are looking for systematic performance models and solutions to support their understanding of components, including their interactions, relationships, dependence, and performance.

5. Conclusions

In this paper, we introduce the concept of *traceable components*, demonstrate how to construct components traceable so that engineers can easily

monitor and check various behaviors, status, performance, and interactions of components. We present a Java framework and tracking model to allow engineers to add tracking capability into components in component-based software. Moreover, we discuss the design and implementation of a distributed supporting environment for component-based software. Component developers can use the presented framework and tracking mechanism to develop Java-based components and EJB-based components. Moreover, the solution is useful to support third-party software components and check various component behaviors. The solution has several advantages:

- Simple and easy to use for building traceable component with low programming effort.
- Flexible and configurable to allow system supporters to monitor various component behaviors, including GUI behaviors, performance, errors and interactions.
- Consistent trace format and lightweight tracking code.
- Scaleable and useful for both in-house and third party components.
- Changeable to fit into different requirements and technologies in organizations.

We are currently investigating new solutions of these open issues in maintenance of software components in a distributed environment, such as trace analysis models, methods and tools to support fault detection, fault isolation, and performance analysis.

6. References

1. Bernard, Londeix, Cost Estimation for Software Development, New York: Addison-Wesley, 1987.
2. Gao, Jerry, "Testing Component-Based Software", STARWEST'99.
3. IEEE Standard for Software Verification and Validation, New York, 1998.
4. Jirotko M, Goguen J., "Requirements Engineering ", Academic Press, 1993.
5. Kit Edward, *Software Testing in the Real World*, New York: Addison-Wesley, 1995.
6. McConnell, Steve. "Rapid Development: Taming Wide Software Schedules", Microsoft Press, 1996.
7. Rada Roy, "Reengineering Software - How To Reuse Programming To Build New State-of-art Software", 2nd edition, New York: Intellect Ltd., 1999.
8. Schmauch Charles H., ISO 9000 For Software Development: Revised Edition, Wisconsin: ASWC Inc., 1995.
9. Voas, J., "Maintaining Component-Based Systems", IEEE Software, Vol. No. 1999.
10. Voas, J. M. and Miller, Keith W., "Software Testability: The New Verification", IEEE Software Vol. 12, No. 3: May 1995, pp. 17-28.
11. Weyuker, Elaine J, "Testing Component-Based Software: A Cautionary Tale", *IEEE Software*, September/October 1998.
12. R. Hofmann, et al., "Distributed Performance Monitoring: Methods, Tools, and Applications", IEEE Trans. Parallel and Distributed Systems, 1994, 5(6): 585-598.
13. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications ACM Vol. 21, No. 7, pp. 558-565, 1978
14. Dieter Haban and Wolfgang Weigel, "Global Events and Global Breakpoints in Distributed Systems," 21st Hawaii International Conference of System Science, pp. 166-175, 1988.
15. K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," ACM TOCS, Vol., 3, No. 1, pp. 63-75, 1985.
16. B.P. Miller and J-D. Choi, "Breakpoints and Halting in Distributed Systems," Proceedings of the 8th DCS, pp. 316-323, 1988.
17. P.C. Bates and J.C. Wileden, "High-Level Debugging of Distributed Systems: The Behavior Abstraction Approach," Journal of Systems and Software, Vol. 3, No. 4, pp. 255-264, 1983.
18. Vaclav Rajlich, et al, "VIFOR: A Tool for Software Maintenance", Software-Practice and Experience, Vol. 20 (1), 67-77, January 1990.
19. Moises Lejter, et al, "Support for Maintaining Object-Oriented Programs", IEEE Trans. On Software Engineering, Vol. 18, No. 12, December 1992.
20. D. Kung, Jerry Gao, Pei Hsia, Y. Toyoshima, Chris Chen, "Developing an Object-Oriented Software Testing and Maintenance Environment", Communications of the ACM, Vol. 38, No. 10, pp. 75-87, Oct. 1995.
21. Wide, N. And Huitt, R. "Maintenance support for object-oriented programs", IEEE Trans. Software Engineering, Vol. 18, No. 12, pp. 1038-1044, December 1992.

