MONITORING OF JEE APPLICATIONS AND PERFORMANCE PREDICTION

Dušan Okanović, Milan Vidaković, Zora Konjović

{oki, minja, ftn_zora}@uns.ac.rs

Faculty of Technical Sciences, University of Novi Sad

Case study

DOI: **10.7251/JIT1102136O** UDC **005.334:004.4**

Abstract: *This paper presents one solution for continuous monitoring of JEE application. In order to reduce overhead, Kieker monitoring framework was used. This paper presents the architecture and basic functionality of the Kieker framework and how it can be extended for adaptive monitoring of JEE applications. Collected data was used for analysis of application performance. In order to predict application performance, regression analysis was employed.*

Key words: *continuous monitoring, Java, JMX, regression analysis*

INTRODUCTION

Degradation of software performance and quality of service over time is well known phenomenon [21]. Also, software testing, debugging and profiling in development phase are not able to detect everything that can happen after the software is deployed. New, previously unknown, errors can show up in this part of software lifecycle. It is necessary to monitor software over time in order to determine the software service levels i.e. how the software compares against service level agreements.

Although software developers usually use debuggers and profilers, there is often not enough time to properly test the software. Another problem with using profilers and debuggers is that they often induce an overhead, something the end user may find unacceptable. In order to determine how software behaves over time, in the real world, it is necessary to perform continuous monitoring of the software. The data provided by the continuous monitoring of software under production workload is much more valuable then the data obtained in the testing phase.

Monitoring system shares resources with the monitored software, causing the performance overhead. In order to control the overhead and the amount of data generated by the monitoring system, we can employ adaptive techniques. These techniques allow changing of monitoring parameters during monitoring process.

Obtained results can be used for visualization and performance analysis of software. Also, based on these results, we can predict how an application response time will change or when will some memory leak cause problem.

The main contribution of this paper is that it presents the use of open-source Kieker framework [20] with the extension for continuous monitoring of JEE applications. We created additional components that allow changing of monitoring parameters during monitoring process. By doing this, we can create flexible monitoring scenarios. As a case study, we present monitoring of a JEE application deployed on a cluster of servers. Results of this monitoring scenario are then used for application performance prediction.

In our earlier papers we presented some parts of this system. In [16] we proposed system's architecture, and in [15] we presented how this system can be applied for monitoring of applications deployed on the JBoss application server. Here, we show further improvements to the system and how the results we obtained can be used for performance analysis and prediction.

The remainder of this paper is structured as follows. Section 2 provides overview of related work in the field of performance monitoring and prediction. Section 3 presents architecture of our system, while section 4 shows its application to monitoring of one JEE test application. Performance prediction using linear regression is shown in section 5. Section 6 provides conclusion to this paper and guidelines for future work.

Related Work

Study presented in [19] indicates that performance is considered critical, but developers usually fail to use monitoring tools. In practice, application level monitoring tools, and especially open-source tools, are rarely used. The reasons for this are usually time constraints (during development), and resource constraints (e.g. performance degradation) during application use. Developers usually limit themselves to profilers and debuggers, during development.

Apart from Kieker, there are several other systems that are used for monitoring of distributed applications.

JBoss Profiler [9] is a tool based on JVMTI and JVMPI APIs. It is used to monitor applications deployed on JBoss application server [8]. The use of JVMTI/JVMPI APIs gives very precise results and low overhead. However, in order to change this tool or extend it, the knowledge of C/C# is required.

COMPAS JEEM [17] inserts software probes during the application startup. The probes are inserted into each of the layers (EJB, servlet…). The advantage of this approach is that there is no need for the application source code changes. However, a drawback of this approach is the fact that different probes must be defined for each application layer.

The system shown in [2] is used for reverse engineering of UML sequence diagrams from JEE applications. The instrumentation is performed using AspectJ, as is in Kieker. The system is limited to diagram generation and it is not suitable for monitoring. Also, the system is not able to monitor web-services, only RMI.

DynaTrace [2] and JXInsight [10] are examples of commercially available application monitoring tools. JXInsight is intended for JEE, while DynaTrace can be used for monitoring of .NET and Java applications. DynaTrace performs monitoring across multiple application tiers using PurePath technology. JXInsight is able to perform automatic analysis and detection of various problem types within applications.

One of the open-source tools that is often in use is Nagios [12], is not used on an application level, but to monitor infrastructure.

This overview shows the lack of tools (especially non-commercial open-source tools) that allow continuous and reconfigurable monitoring of JEE applications with low overhead. Kieker framework in combination with JMX [20] can be used for monitoring of JEE applications. It uses AspectJ [1] – load-time weaving configuration – for instrumentation and separation of monitoring code from application code. JMX, which is in the core of JEE application server infrastructure, can be used for controlling of the monitoring process.

Performance prediction of software is a part of capacity management process [18]. Developers usually use performance monitoring to obtain data for trend analysis. Prediction is also used in proactive management of software aging.

In [22] authors present their findings in the area of software aging and propose a proactive technique called "software rejuvenation". The idea is to occasionally terminate the application and clean its internal state of accumulated errors. This should be planned and initiated based on measurement, analysis and prediction.

Nudd et al. [13] provide a methodology for detailed performance prediction through software design and implementation cycles. It has relatively fast analysis time and can be used in runtime to assist in dynamically changing systems.

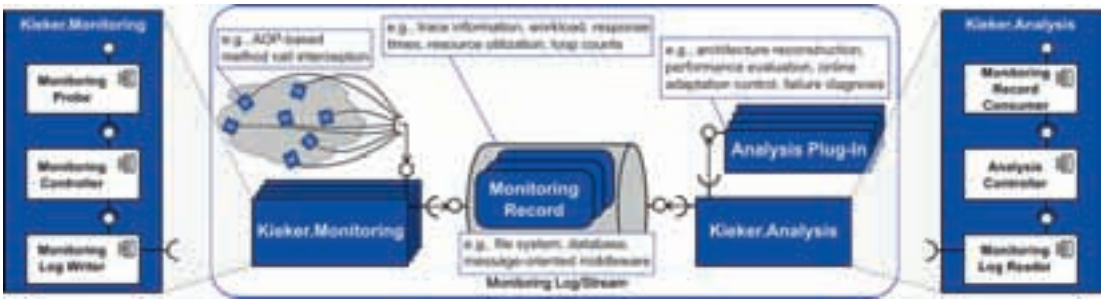
KIEKER FRAMEWORK

Kieker is a framework for continuous monitoring of all types of Java applications. It consists of:

- Kieker.Monitoring – component responsible for data collection and
- Kieker.Analysis – component that performs visualization of the data

Architecture of the Kieker framework is shown in fig. 1.

FIG. 1. KIEKER FRAMEWORK COMPONENT DIAGRAM



Kieker. Monitoring component is executed on the same computer where monitored application is being run. This component collects data on the execution of monitored applications. Monitoring Probe is a software probe that is inserted into the observed application and takes various measurements. Monitoring Log Writer stores collected data, in the form of MonitoringRecords, into the Monitoring Log. Monitoring Controller controls the work of this part of the framework.

The data in the Monitoring Log is analyzed by Kieker.Analysis component. Monitoring Log Reader reads records from Monitoring Log and forwards them to Analysis Plugin. Analysis Plugin analyzes and visualizes gathered data. Control of all components in this part of the Kieker framework is performed by Analysis Controller component.

Monitoring Log can be anything (e.g. file, database, JMS queue) because the framework does not depend on the type of storage.

Both components of the Kieker framework work completely independently. This approach allows a single computer to run monitored software, to store monitoring data in a file system or database on another computer and to perform data visualization and analysis on a third computer.

Software Instrumentation

Software instrumentation in the Kieker framework can be performed using aspect-oriented programming or by inserting pieces of code, which take measurements, create monitoring records and store these records using Kieker.Monitoring components. The drawback of the second approach is that it pollutes program code with the code that is not a part of the application. Use of aspect oriented programming is more appropriate way to perform program instrumentation. Developers can separate program logic from monitoring logic (separation of concerns). Instrumentation consists of writing aspect classes and weaving them with application classes. These aspects intercept execution of program logic at points defined using join points and add additional behavior using advices.

Among different AOP tools for the Java framework, Kieker framework uses AspectJ.

There are several ways to perform program instrumentation using AOP. Firstly, one can choose whether to instrument program code – i.e. weave aspects with application classes – during application development (compile-time weaving) or when classes are loaded (load-time weaving). Compile-time weaving is performed using AspetJ's *ajc* compiler: compiler weaves application code with aspects and generates new classes.

The other way to instrument the application is load time weaving. In this case, weaving of the precompiled aspects with application classes is performed during loading of classes. The disadvantage of this approach is that launching of applications takes a bit longer than in case of compile time weaving, but there is no need for source code and recompilation of the application. Load-time weaving configuration is performed with the aop.xml configuration file. In the aop.xml file we define aspects and parts of the software (classes, packages) that are to be woven together.

Developer can chose to monitor every method in every class or only designated ones. The usual way to designate methods and classes are Java annotations. `OperationExecutionMonitoringProbe` annotation and several different aspects are distributed with the Kieker framework and allow creation of different monitoring scenarios.

Regardless of the chosen scenario (compile or load time weaving, monitoring of all or only annotated methods), the aspect intercepts executed method, takes necessary measurements, lets the method execute, creates MonitoringRecord and, using Monitoring Controller, stores data into MonitoringLog. Within one application there can be multiple annotations and aspects, and they can perform various measurements.

Kieker Framework Extension

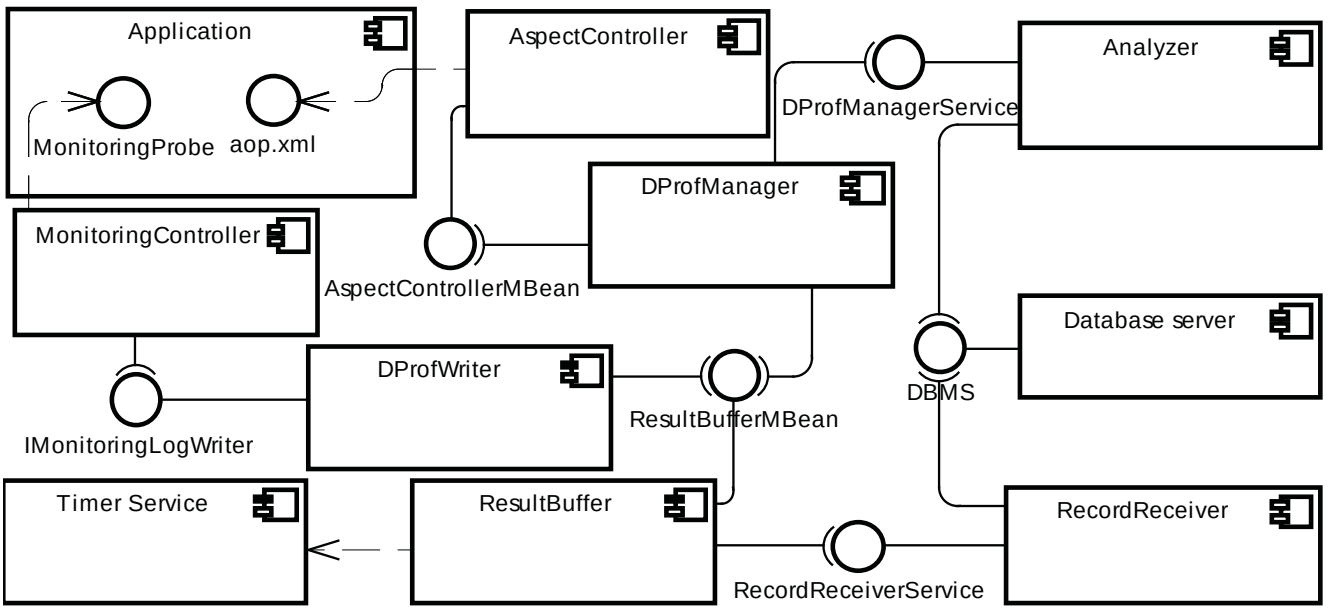
The Kieker framework was extended by implementing new MonitoringLogWriter and several new components. We call this new system the DProf.

Architecture of this part of the system is shown in fig. 2.

A new MonitoringLogWriter – DProfWriter stores all records into a special buffer – ResultBuffer. The ResultBuffer is implemented as a JMX MBean. This allows the buffer to be controlled programmatically or from any JMX console. The buffer sends monitoring records to a service running on a remote server – RecordReceiver. Records can be sent periodically in bulks or as soon as they arrive into the buffer. This remote service stores records into the database for further analysis. Essentially, the combination of the buffer, the service and database assumes the role of Kieker's Monitoring Log.

Analyzer component analyzes gathered data and sends new monitoring parameters to DProfManager. DProfManager controls ResultBuffer and AspectController. The configuration of monitoring system is performed through the aop.xml. AspectController performs monitoring system reconfiguration by adding and removing clauses from aop.xml.

Fig. 2. Extensions for Kieker framework



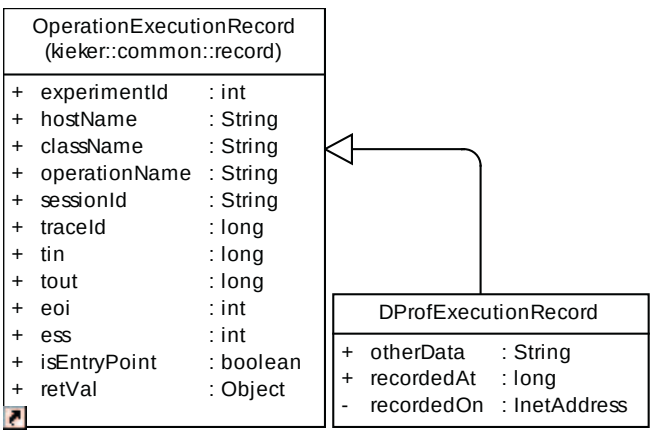
The system can be configured to:

- Record normal results – this is usually used to determine normal values of monitored parameters.
- Find which component is not in accordance with the expected values. In this case, the system monitors only top-level (interface) methods of components. If there is discrepancy with the expected values, the system turns on monitoring in the next level. The last method that has values different than expected is identified as the source of the problem.
- Find which component consumes selected resource the most. The process is similar to the previous. The difference is that there are no expected values. We only try to find on which level, which method consumes the most of the designated resource.

Another extension of the framework is addition of the new type of Monitoring Record – `DProfExecutionRecord`. It extends the standard Kieker's `OperationExecutionRecord` by adding new attributes. Attribute `recordedOn` holds the IP address of the computer where the record was created. Attribute `recordedAt` holds the time in milliseconds when the record was created. Because the original `OperationExecutionRecord` holds only information about response time, we have added the attribute `otherData`. It holds performance information of any other parameter, such as memory, CPU, network.

`OperationExecutionRecord` class is shown in Fig 3.

FIG. 3. OPERATIONEXECUTIONRECORD CLASS



CASE STUDY – FINDING PERFORMANCE BOTTLENECKS

The use of the Kieker framework for monitoring of distributed JEE applications will be demonstrated on the software configuration management (SCM) application described in [14] deployed on a JBoss 5.1.0 server. This is a JEE application responsible for tracking of applications and application versions.

The application is implemented using EJB technology. *Entity* EJBs [4] are used as O/R mapping layer. They are accessed through the *stateless session* EJB (SLSB), modeled on the *façade* design pattern [5]. SLSBs are annotated to work as JAX-WS web services as well.

Application client is the Java Swing [7] application which uses web services to access the application.

Listing 1. represents a part of the `OrganizationFacade` class. `createOrganization` method invokes `checkOrgName` method, retrieves object of `City` class by its id and creates a new entity EJB. All of these methods are annotated with `@OperationExecutionMonitoringProbe`.

Listing 1. Stateless session EJB `OrganizationFacade` class

```
@Stateless
public class OrgannizationFacade
     implements OrganizationFa-
cadeService {
     // ...
     @OperationExecutionMonitor-
ingProbe
     public Organization
createOrganization(String org-
Name,
     String address, String
email, long cityId) {
     checkOrgName(orgName);
     City c = entityManager.
find(City.class, cityId);
     Organization org =
     new Organization(orgName,
address, email, c);
     entityManager.persist(org);
```

```
      return org; }
      @OperationExecutionMonitor-
ingProbe
      public void checkOrgName() {
      // zip code check
      // ...
      }
}
```

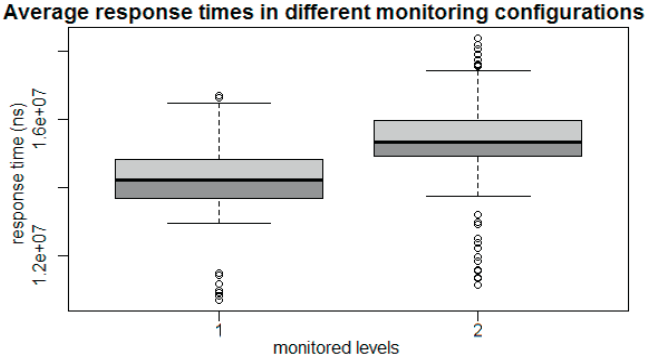
The testing will be conducted by repeatedly invoking `OrgannizationFacade. createOrganization (...)` method. These invocations are supposed to generate data which will be used for program performance analysis.

Initially, the system is configured for monitoring of methods in the top level – `createOrganization` method in this case. The system is configured to analyze monitoring data every two hours and change monitoring parameters, if needed.

In the first pass, results show that `createOrganization` method takes to long to execute. The monitoring system then included second level of methods into monitoring configuration. After two hours, the results were analyzed again. They have shown that average execution time of the `checkOrgName()` method is above expected. This method required refactoring, in order to meet demands.

Fig. 4. shows how response time changes when monitoring of another level is added to monitoring configuration.

FIG. 4. COMPARISON OF RESPONSE TIME WHEN ONE OR TWO LEVELS OF METHODS ARE MONITORED



We can see that the response time increases if another level of methods is added to monitoring configuration. By using adaptive monitoring technique, our system behaves as human tester would. It monitors only one level of methods, and turns on monitoring of lower level only if a problem is detected. This way, the total overhead is reduced.

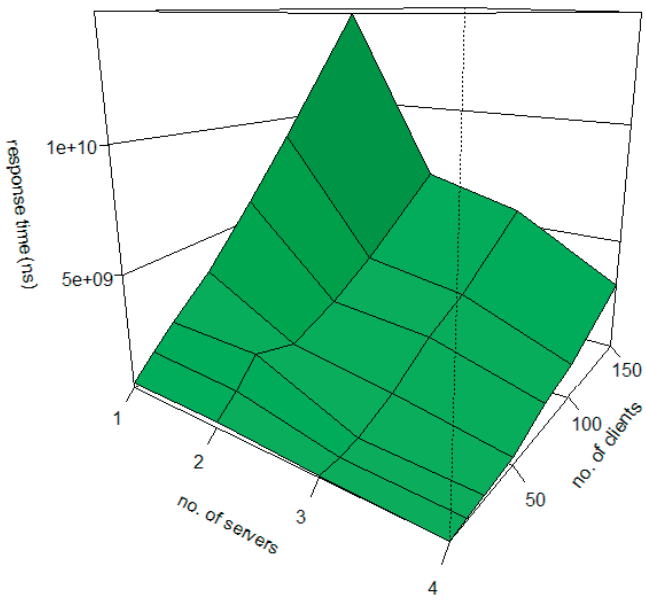
CASE STUDY – RESPONSE TIME PREDICTION

We have deployed our test application, from previous case, on a cluster of four servers and generated different workloads. We wanted to see what happens with the response time when we increase workload and number of servers in cluster.

Results are as expected: the response time increases with the number of clients and decreases with the number of servers.

Obtained results are shown in Fig. 5.

FIG. 5. AVERAGE RESPONSE TIME FOR DIFFERENT SERVER CONFIGURATIONS AND WORKLOADS



In order to predict how response time would change if the number of clients is increased or if we add another server, we employed the regression analysis. A model, in which we have two independent variables – the number of servers and the number of clients, and one dependent – response time, was created.

The analysis of this model shows that these two independent variables explain 83.4% of response

time variance. The rest of the average response time is influenced by some external factors. In this case, these can be hardware glitches, network traffic and cluster load balancer influence.

Both of the predictors are significant (the *p* value is less then 0.01) and the model provides us with coefficients for prediction shown in the table 1.

TABLE. 1. REGRESSION MODEL COEFFICIENTS

	Coefficients		Standardized Coefficients	t	Sig.
	B	Std. Error	Beta		
Intercept	$1.939 \cdot 10^9$	$4.965 \cdot 10^8$		3.906	0.000
Number of users	$6.221 \cdot 10^7$	4562536.524	0.820	13.636	0.000
Number of servers	$-1.254 \cdot 10^9$	$1.877 \cdot 10^8$	-0.402	-6.682	0.000

The following equation was derived from the table 1.:

$$\widetilde{RT} = 6.221 \cdot 10^7 \cdot N_{usr} - 1.254 \cdot 10^9 \cdot N_{ser} + 1.939 \cdot 10^9$$

($\widetilde{RT}\widetilde{RT}$ is estimated response time, $N_{usr}N_{usr}$ is number of users and $N_{ser}N_{ser}$ is number of servers). By using this equation, we can estimate (with the satisfying precision) how response time will change (with the respect to the calculated errors for every coefficient) if we vary the number of users and servers.

Regression results show that we can use this model for performance prediction with satisfactory precision.

CONCLUSION

This paper presents the use of the DProf system for continuous monitoring of distributed Java applications and the use of monitoring data for performance prediction.

It describes the Kieker framework, its architecture and configuration. The Kieker was used for monitoring of one SCM application which was implemented using EJB and web-services technologies. Additional components, implemented using JMX technology, allow for development of the reconfigurable application monitoring system. During the monitoring, it is possible to change monitoring parameters. The system can also be configured to change monitoring parameters automatically in order to provide more precise data or to reduce performance overhead.

We have applied the regression analysis in order to estimate application performance. The result was the model which allows us to predict what will happen to application performance if the number of clients changes or if we change the number of servers the application is deployed on.

Future work will focus on further improvements of monitoring system. Also we will try to apply other machine learning techniques in order to improve performance prediction model.

References

[1] AspectJ,h ttp://www.eclipse.org/aspectj/

[2] Briand LC et al. (2006) Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9), 642–663.

[3] Dynatrace, http://www.dynatrace.com/en/

[4] EJB 3.0, http://java.sun.com/products/ejb/

[5] Gamma E et al. (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Proffesional, Boston, USA.

[6] Grottke M et al. (2006) Analysis of Software Aging in a Web Server. *IEEE Transactions on Reliability*, 55(3), 411-420.

[7] Java Swing, http://java.sun.com/javase/6/docs/technotes/guides/swing

[8] JBoss Application Server, http://www.jboss.org/jbossas

[9] JBoss Profiler, www.jboss.org/jbossprofiler

[10] JXInsight,h ttp://www.jinspired.com/products/jxinsight/

[11] Kiczales G. et al. (1997) Aspect-Oriented Programming. In Proceedings of ECOOP, pp. 313, Vienna, Austria

[12] Nagios,w ww.nagios.com

[13] Nudd GR et al. (2000) Pace-A Toolset for the Performance Prediction of Parallel and Distributed Systems. *International Journal of High Performance Computing Applications*, 14(3), 228-251.

[14] Okanović D and Vidaković M (2008) One Implementation of the System for Application Version Tracking and Automatic Updating. In Proceedings of the IASTED International Conference on Software Engineering 2008, pp 62–67**,** Innsbruck, Austria

[15] Okanović D and Vidaković M (2011) Performance Profiling of Java Enterprise Applications. In Proceedings of the International Conference on Internet Society Technology and Management, on CD, Kopaonik, Serbia,.

[16] Okanović D et al (2011) Towards Adaptive Monitoring of Java EE Applications. In Proceedings of the 5th International Conference on Information Technology, on CD, Amman, Jordan

[17] Parsons T et al. (2006) Non-Intrusive End-to-End Runtime Path Tracing for J2EE Systems. *IEEE Proceedings – Software*, 153(4), 149–161.

[18] Rudd C and Lloyd V (2007) Service Design. The Stationery Office, UK

[19] Snatzke RG (2008) Performance survey 2008. (available at http://www.codecentric.de/export/sites/www/resources/pdf/performance-survey-2008-web.pdf)

[20] Sullins BG and Whipple MB (2002) JMX in Action. Manning Publications, USA

[21] van Hoorn A et al. (2009) Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical report, Institut für Informatik, Oldenburg, 2009.

[22] Yilmaz C et al. (2005) Main Effects Screening: A Distributed Continuous Quality Assurance Process For Monitoring Performance Degradation in Evolving Software Systems. In Proceedings of the 27th International Conference on Software Engineering, pp 293-302, St. Louis, USA

Submitted: October 25, 2011
Accepted: December 31, 2011