Monitoring in Adaptive Systems using Reflection

Dylan Dawson, Ron Desmarais, Holger M. Kienle, and Hausi A. Müller
Department of Computer Science
University of Victoria, Canada
{ddawson, rd, kienle, hausi}@cs.uvic.ca

ABSTRACT

Continuous evolution is a key trait of software-intensive systems. Many research projects investigate mechanisms to adapt software systems effectively in order to ease evolution. By observing its internal state and surrounding context continuously using feedback loops, an adaptive system is able to analyze its effectiveness by evaluating quality criteria and then self-tune to improve its operations. The goals of these feedback loops range from keeping single variables in a prescribed range to satisfying non-functional requirements by regulating decentralized, interdependent subsystems.

To be able to observe and possibly orchestrate continuous evolution of software systems in a complex and changing environment, we need to push monitoring of evolving systems to unprecedented levels. It has been established that security has to be built into a system from the ground up and cannot be added as an afterthought—the same is probably true for intensive monitoring. We propose to monitor adaptive systems with autonomic elements to enhance their assessment capabilities. In this paper, we discuss how to build monitoring into Java programs from the ground up with reflection technology to detect normal and exceptional system behavior.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Design Tools and Techniques—Object-oriented design methods; D.2.11 [**Software Engineering**]: Software Architecture—Domain-specific architectures, Patterns; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Documentation, Languages, Management, Performance, Reliability, Standardization

Keywords

Continuous evolution, self-adaptive systems, autonomic elements, feedback, control loops, monitoring, reflection, introspection, metaobjects, exceptions, Java

SEAMS 2008, May 12–13, 2008, Leipzig, Germany.

1. INTRODUCTION

Continuous evolution has emerged as a key characteristic of software-intensive and ultra-large scale systems. According to a recent study conducted by the Software Engineering Institute (SEI) [22], such systems cannot be fully specified and engineered in a top-down manner as we are used to, but are rather constructed by satisfying requirements through regulating decentralized, interdependent subsystems. In such an environment, individual subsystems have to be more self-sufficient, robust and at the same time be able to adapt due to changes in their context and operating environment. In traditional engineering of software systems, many assumptions about the context of an application are fixed at design time and as a consequence, functional and non-functional requirements can be hard-wired into the systems and thus need not be monitored for continuous satisfaction. However, for software-intensive systems, which are subject to continuous changes in context and operating environment, monitoring of requirements satisfaction will likely be the norm rather than the exception. To regulate the satisfaction of requirements, individual subsystems must adapt. For example, Litoiu discusses hierarchical control in a class of Quality of Service and Service Oriented Architecture applications, including appropriate architectures and algorithms [18].

There are many research projects investigating approaches to adapt software systems effectively [6, 16, 17]. A common feature of all approaches is *feedback (or control) loops* as core components of adaptive systems [20]. Feedback loops observe the system's internal state and its surrounding context, analyze its effectiveness by evaluating quality criteria and then adjust parameters and components to improve its operations [8].

Hitherto, most developers had no need to instrument their software with sensors and effectors to observe its hard-wired requirements. For self-adaptive software-intensive systems however, a control loop with sensors and effectors is a necessity. The autonomic computing community, spearheaded by IBM, offers the notion of an *autonomic element* to implement such control loops [16]. This architectural element seems an ideal building block with which to design software systems from the ground up with adaptive mechanisms [15]. For example, at the lowest level, autonomic elements could monitor a system's 'vital signs,' which are typically not made explicit in the source code (except, perhaps, in comments). The frequency of raised exceptions or run-time check violations could be monitored (similar to taking a person's blood pressure or pulse) and then used to assess changes in a system's health. Critical regression tests could be regularly performed while the system is in operation to observe satisfaction of selected requirements. One way to implement monitoring of internal state using such

autonomic elements is to employ reflective mechanisms offered by the underlying programming languages and run-time environments.

In this paper, we explore how autonomic elements and reflection technology can be used to instrument Java programs from the ground up to recognize normal and exceptional behavior by monitoring Java exceptions over time. Section 2 describes related work in the area of code instrumentation. Section 3 briefly introduces autonomic computing and discusses the architecture of an autonomic element. Section 4 provides background on Java reflection. Section 5 discusses our approach to instrumenting programs from the ground up using Java's reflective capabilities, especially dynamic proxy classes. Section 6 presents a small example of building a suitable Java-based infrastructure for monitoring raised exceptions that follows our approach. Section 7 closes the paper with conclusions and future work.

2. RELATED WORK

There are many approaches to instrumenting existing systems with the goal to obtain information about their run-time behavior (e.g., sequences of method invocation or profiling of execution times). In Java, the bytecode representation of a class can be instrumented before a class is loaded. This can be conveniently achieved with tools such as the Apache Byte Code Engineering Library (BCEL) [2] or ASM [1], which provide APIs to inspect and manipulate Java classes at the level of JVM instructions. For example, BCEL has been used to realize a generic framework for collecting dynamic information of Java programs [4]. Another suitable tool is Javassist [5], which offers a source-level API that allows specifying of modifications as Java source text without requiring knowledge of the underlying bytecode implementation. It enables Java programs to define a new class at run-time and to modify a class file when it is loaded.

Reflective middleware, which uses reflection to achieve openness and re-configurability of its behavior, can also be used to instrument systems. Huang *et al.* have implemented autonomic computing middleware based on underlying reflective middleware [14]. Specifically, they have built autonomic managers to observe and modify the behavior of a J2EE application server using reflection mechanisms.

Aspect-oriented programming languages are also used to instrument code. For example, Briand *et al.* have leveraged AspectJ to instrument multi-threaded and distributed Java code [3]. There are also dedicated toolkits for monitoring and testing such as the Eclipse Test & Performance Tools Platform (TPTP) project [24]. All of the above approaches have different trade-offs in terms of expressiveness, learning curve, instrumentation at compile/load/run-time, or execution overhead.

3. AUTONOMIC ELEMENTS

Autonomic Computing presents a new paradigm where computing systems manage themselves, guided by high-level objectives [16]. The metaphor is derived from our autonomic nervous system, which controls normal and exceptional body functions, from respiration to pupil dilation, through the sympathetic and parasympathetic subsystems without our conscious awareness or effort.

In an effort to define a common approach to building self-managing systems, IBM has defined an architectural blueprint for autonomic computing [9]. The architectural blueprint suggests fundamental building blocks for designing self-configuring, self-healing, self-protecting and self-optimizing software systems.

Figure 1 depicts the main building block, an autonomic element, which consists of an autonomic manager and a managed element tied together via a closed control loop. The monitor in the autonomic manager senses the managed element and its context, filters the accumulated sensor data, and stores relevant events in the knowledge base for future reference. The analyzer compares event data against patterns in the knowledge base to diagnose symptoms and also stores the symptoms for future reference in the knowledge base. The planner interprets the symptoms and devises a plan to execute the change in the managed element through the effectors. An interface consisting of a set of sensors and effectors is called a manageability endpoint. To facilitate collaboration among autonomic elements, the control and data of manageability endpoints are standardized across managed elements.

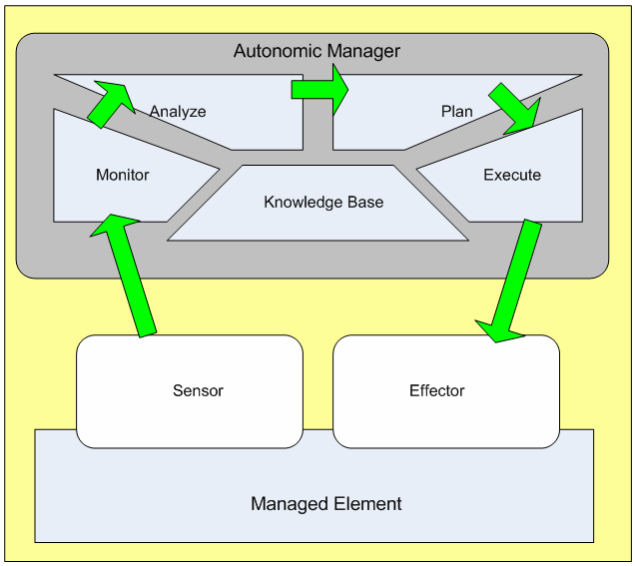

Figure 1 Autonomic Element [9]

A simple example of a managed end point could be a web service that provides weather information to subscribed users. An autonomic manager could continuously sense the output of the service and describe this output as events in the knowledge base. The analyzer could interpret these events as normal or abnormal and store its analysis (symptoms) into the knowledge base. The planner could determine an appropriate course of action based on the symptoms in the knowledge base and with guidance from a set of policy rules it must follow.

The idea to describe and implement software systems with control loops is not a new concept. Over a decade ago, Shaw compared a software design method based on process control to an object-oriented design method [21]. The process control pattern described in that paper, which resembles an autonomic element, can be seen as a building block for creating software-intensive systems that are more self-aware (e.g., by continuously monitoring normal and exceptional behavior).

4. JAVA REFLECTION

The concept of reflection has been studied independently in many different areas of science and engineering and in the area of programming languages across language paradigms [7]. Examples of reflective programming languages include Lisp, Self, Smalltalk, Prolog, Python, C++, and Java. Over the past decade, reflection implementations for C++ and Java have matured enough to be practical for adaptive computing.

The reflection mechanisms of a programming language provide a running program with the ability to examine itself and its environment. To perform self-examination, a program needs an accessible representation of itself; this level of indirection is facilitated through metadata and is fundamental to a reflective system. The two main aspects of self-manipulation are *introspection* and *intercession*, which are the abilities of a program to observe and modify (respectively) its own state and behavior. Both aspects require a mechanism for encoding execution state as data. In Java this is realized with so-called metaobjects, which provide access to the representation of Java classes and are available in the `java.lang.reflect` package.

4.1 Metaobjects

The Java programming language provides reflective access to metaobjects for many important language constructs including, but not limited to: classes, methods, fields, interfaces, modifiers (e.g., public, private, static, abstract, or synchronized), arrays, the call stack, and the class loader. For example, the metaobject classes `Class` and `Method` are used to represent the classes and methods of executing programs.

Metaobjects not only provide reflective query access to the components of a program, but also provide an interface to change or adapt its structure and behaviour. During dynamic invocation, a `Method` metaobject can be used to invoke the method that it represents. Similarly, `Field` objects expose the attributes of a field (e.g., name and modifiers), allowing programs to query and modify values. This functionality allows programs to handle objects of classes that have not been specified at design time.

4.2 Dynamic Loading

Some adaptations can be accomplished by adjusting parameters, but more significant changes require modification of existing code or incorporation of new code during run-time. In Java, this can be accomplished with dynamic class loading. When combined with good object-oriented design (e.g., a plug-in architecture), dynamic loading provides additional flexibility, increasing the likelihood of accommodating changes in requirements [19].

In Java, dynamic loading can be accomplished using the reflective facility `Class.forName(String)`. This static method returns a `Class` object given a fully qualified class name. This object can then be instantiated using reflective construction as follows:

```
Class myClass = Class.forName("demo.ObjImpl");
MyObject obj = (IObject) myClass.newInstance();
```

Dynamic loading can also be enhanced with the use of custom `ClassLoaders` which govern where to search for classes to load, which class gets loaded and used, or protocols to use when finding a class [8]. A program can provide its own custom class loaders to modify the default class loading behavior. Class loading can be considered a reflective facility because the ability to create and execute a new class as well as to modify the default class loading behavior is a form of intercession. This kind of intercession permits a large increase in application adaptability, which ranges from deciding what code is used to implement a class to replacing that code even when the class is active.

Both dynamic loading and reflection facilitate delegation. Delegation provides a level of indirection between different parts of a program and allows them to vary independently from each other, while reflection increases the range of variation by making more kinds of objects available [11].

4.3 Dynamic Proxies

The Java reflection API includes a class called `Proxy` to realize so-called dynamic proxy classes. When a proxy class is created, a list of interfaces that the proxy will implement is given. Instead of instantiating and using an object `obj` for a class `C` directly, a proxy object `prxy` is created that takes `obj` as an argument:

```
class C implements I { ... };
C obj = new C(...);
Proxy prxy = Proxy.newProxyInstance (
                C.getClass().getClassLoader(),
                C.getClass().getInterfaces(),
                new MyIH(obj) );
```

The proxy `prxy` supports the same interface as the target object `obj`. As a result, proxies can be created and used transparently in place of any object in the system, including other proxies. Thus, dynamic proxies are an effective technique for adding properties and behaviors to objects.

Generally, proxies can be used whenever code needs to execute before or after certain method invocations of an interface. To achieve this, a proxy needs to be provided with an extension of an `InvocationHandler` that overrides the inherited `invoke` method [23]. For example, the above proxy can intercede and delegate method invocation as follows (ignoring exception handling to simplify the code):

```
class MyIH implements InvocationHandler {
    public Object invoke(Object proxy, Method m,
                         Object[] args)
    {
      preProcessing();
      result = m.invoke(obj, args); //delegation
      postProcessing();
      return result;
    }
}
```

The `InvocationHandler` is used to accomplish delegation by handling each method call on a proxy instance, and holding any references to the targets of that proxy instance. Overriding the inherited `invoke` method allows developers to add pre- and post-processing code surrounding method delegation (cf. Figure 2). This form of intercession allows 'wrapper' code such as for monitoring and logging to be gathered in one place. This technique greatly simplifies maintenance, testing, and debugging, because proxies keep such functionality from becoming entangled with application logic, and allows developers to reuse application-neutral wrapper code in other applications.

Because of Java's introspection of argument interfaces at the time of the proxy's creation, it is neither error-prone nor fragile to interface updates. This property yields several benefits. Since a proxy is instantiated by specifying its supporting interfaces—the corresponding implementation is created dynamically at run-time. Furthermore, a proxy can support interfaces that were not available when the application was compiled. This means that proxies can be used in combination with dynamic loading to enhance application flexibility [11].

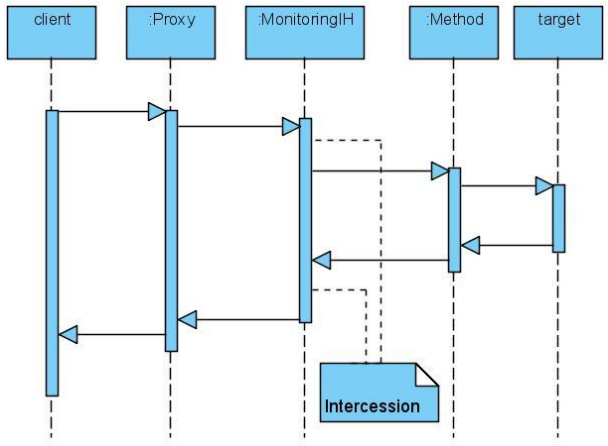

Figure 2 Pre- and post-method delegation intercession

The use of proxy classes increases flexibility or adaptability by creating modules that concentrate the code needed to give properties to an object and that may be reused in other contexts. Proxies are a flexible and modular approach to monitoring; however, as with any reflective mechanism, the use of proxies does of course incur a performance penalty for the extra level of indirection [13]. This is an important consideration when deciding on the number of proxies and the granularity of the monitoring for the system under observation.

5. MONITORING WITH PROXIES

In Java there are two techniques to facilitate behavioral or structural changes using the reflection API: (1) operations for using metaobjects such as dynamic invocation, and (2) intercession, in which code is permitted to intercede in various phases of program execution. Of these techniques, intercession—facilitated through the use of Java's dynamic proxy—provides a convenient (but not most efficient) method for implementing low level monitoring.

For example, instances of a single proxy class that implements monitoring can be used to bind to the run-time interfaces of any object that needs to be monitored, and intercede on any or all method invocations. In this way, any exceptions that a target object throws can be caught, traced, and logged with complete transparency to the objects user. The results can then be stored in the knowledge base of an autonomic element to identify, for instance, bursts or trends of raised exceptions.

Using these reflective techniques at design time, we can lay the plumbing for problem determination and localization at run time. Proxies can be used to monitor selected objects and components, even those that are not necessarily known during design and compile-time. For example, a monitoring proxy that observes raised exceptions can be selectively enabled for objects that are critical to the operation of the system. Furthermore, monitoring and other behaviors such as tracing and profiling can be dynamically composed and enabled or disabled at run-time if each behavior is encapsulated in a proxy. Such dynamic composition can be easily achieved by chaining proxies together.

5.1 Chaining Proxies

Chaining proxies together allows us to realize *adaptive monitoring*, which allows for the reconfiguration of monitors during run-time. Initially, for instance, we may want to monitor only the exceptions generated by the system. Bursts of exceptions, however, may trigger more aggressive monitoring such as tracing of method invocations and profiling of execution hot-spots.

Constructing such a monitor can be accomplished with proxies via implementing the `InvocationHandler` interface to perform exception logging, tracing, profiling and instantiating a corresponding proxy object whenever the target objects needs that kind of monitoring. The intercessional capabilities of the proxies can be turned on and off as required. Depending on system demands, individual proxies can be made to intercede or not, or can be made to intercede with varying levels of aggression.

Figure 3 demonstrates that arranging proxies in a chain has the effect of composing the properties and behaviors implemented by each proxy. The structure of the chain however, requires careful design. When a client makes a call to what they perceive to be the real target, they are actually operating on a proxy. Likewise, individual proxies normally work under the assumption that their target is the real target, not another proxy [11].

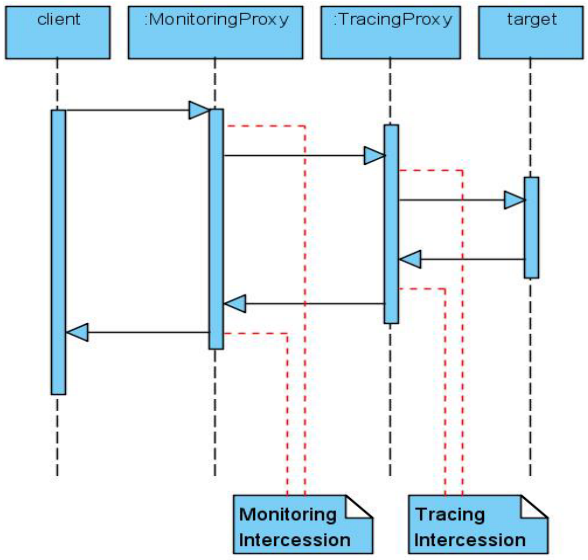

Figure 3 Compositional intercession

If the target of a proxy is another proxy, the `InvocationHandler` may behave under an incorrect assumption. To overcome this difficulty, we can make use of an abstract class, `AbstractInvocationHandler`, from which

we will derive other handlers for all chainable proxies. This abstract class has the ability to recursively search the chain of proxies to locate the 'real' target at the end of the chain, and can make decisions about their intercessional behavior based on this knowledge [11].

6. EXAMPLE: AN ADAPTIVE EXCEPTION MONITOR

The previous sections illustrated how Java reflection and dynamic proxies can be leveraged to facilitate the design of autonomic managers. The example in this section shows how to build such an autonomic observer to monitor Java exceptions over long periods of time. The assumption is that during normal operation exceptions are raised in predictable patterns, but in bursts during exceptional behavior. The exceptional behavior might be due to unexpected changes in the system's internal state or its environment. Recognizing such changes using autonomic observers will give the system a chance to adapt and evolve.

The reflective, adaptive exception monitor is implemented in Java by providing intercessional processing after method delegation to any object in the system. Specifically, it is able to transparently inspect each exception generated by specified objects. Exceptions are logged in the knowledge base of the autonomic element for future pattern or symptom analysis. The monitoring proxy is also designed to work in a chain so that other proxies can be composed together (e.g., tracing or profiling).

The key to the development of a reflective, adaptive exception monitor resides in the implementation of a specialized `InvocationHandler`. Implementing the `Invocation-Handler` interface allows us to write code that can intercede during Proxy method delegation to any Java object in the system. This is accomplished through reflective access to the `Method` metaobject of the target object as illustrated in Figure 2. Currently, monitored Java exceptions are captured through post-method invocation intercession.

The monitoring `InvocationHandler` will be able to perform adaptive monitoring of exceptions generated by any Java object. Its key capabilities include:

- The ability to be turned on and off;
- The ability to react to changing demands (e.g., bursts);
- The ability to detect normal and abnormal system behavior over long periods of time; and
- The ability to be composed together with other handlers.

Exceptions generated by the system are logged sequentially in time for each object for which a monitoring proxy is employed. As bursts of exceptional activity are recorded, the monitoring proxy will increase the aggressiveness with which it monitors. Likewise, when the system is operating normally, the monitor may choose to decrease its aggressiveness. Increases and decreases in aggressiveness can range from not monitoring at all, to simply logging the few exceptions that are generated under normal conditions, to logging every exception generated by every object and finally to employing the use of other proxies to chain other intercessional behaviors together such as tracing and profiling.

Code Listing 1 in Appendix A shows the interface to a custom `InvocationHandler`, `MyInvocationHandler`. This interface specifies how proxies can be created and composed together in a chain. The methods `addToFront()`, `addToBack()`, `contains()`, and `remove()` illustrate that the proxy chain will exhibit functionality commonly associated with a linked list. This interface also specifies that proxies constructed with this type of handler can operate with varying degrees of aggressiveness expressed with a Java enumeration:

`MONITOR_LEVEL {HIGH, MEDUIM, LOW, NONE}.`

Another important facility specified here is the ability to register event listeners for each proxy. This allows for adaptive orchestration of the entire proxy chain through a centralized `Controller`.

The `MyInvocationHandler` interface is implemented by the `AbstractInvocationHandler` mentioned in Section 5.1. Code Listing 2 in Appendix A shows that this base implementation contains a reference to the centralized `Controller` and a `Logger`. The `Controller` and `Logger` both work to close the control loop by monitoring events generated by the chain of proxies (such as the logging of an exception), and modifying the behavior of the proxy chain in response.

In the example application we have created, the following processes take place:

1. When a client attempts to instantiate a specific target object using the Factory design pattern [12], the object is created and a proxy to that object is generated and returned transparently to the client.
2. During instantiation, the proxy that is created (called the primary proxy) registers itself with the controller. The purpose of the primary proxy is to maintain the head of the proxy chain.
3. After a primary proxy has been registered with the `Controller`, the `Controller` can decide which proxies to add to the chain. In our example, only an exception monitoring proxy is initially chained.
4. The `Controller` receives an event notification each time the exception monitoring proxy transparently logs another exception.
5. The `Controller` can then analyze the exceptions that have been logged and adjust the `MONITOR_LEVEL` with which the object is monitored or chain additional proxies for tracing and profiling.

It is important to note that the Controller can also dynamically load new `InvocationHandlers` that were not specified at design-time for the purpose of chaining new types of proxies.

In our implementation, the monitoring proxy can intercede and inspect exceptions as they are generated for any object in the system. Exceptions are initially logged in the least aggressive mode. Only one in every three exceptions generated by a specific object is inspected. (This is for illustration purposes only. A more realistic example might be to inspect only application-defined exceptions.) The time between successive exceptions is then used to control the aggressiveness of the monitor. Large durations of time between exceptions will cause the monitor to maintain its

least aggressive monitoring mode (i.e., logging only one in three exceptions). Shorter durations may move the monitor from mild through moderate to highly aggressive monitoring modes, where most or all exceptions are logged and inspected for the purpose of problem determination and localization.

The adaptive measures taken can easily be configured for control either through simple techniques such as observing thresholds, or more advanced techniques involving event correlation, event grouping, and scenario recognition. In the most aggressive monitoring mode, every exception that is generated is logged, and the monitor may now begin to employ the use of proxy chains to capture more system wide event data.

6.1 Logging

Extensions of the `AbstractInvocationHandler` also make use of a generalized logger that can be customized to log to one or more repositories (i.e., knowledge base of an autonomic element) simultaneously such as standard out, flat text files, relational databases, or web services.

The generalization of the logging component also allows us to log significant system events in standards compliant formats such as the Common Base Event (CBE) or Web Services Distributed Management (WSDM) formats [10, 25] to facilitate further analysis. Information that is captured for logging purposes can include but is not limited to:

- The object that generated the exception;
- The exception itself (including the stack trace);
- The time the exception was raised; and
- Profiling and tracing information.

Another benefit of the generalized logger is that it is customizable for usage by multiple components in the system. A monitor may use it to log exception information to a web service, while a tracing component might use it to simply log to a flat text file or another repository.

7. CONCLUSIONS AND FUTURE WORK

The need to regulate large, complex, decentralized systems by monitoring and tuning the satisfaction of requirements is upon us. We need numerous sensors, monitors, analysis/planning engines, and effectors to be able to observe and control independent and competing organisms in a dynamic and changing environment. A subsystem, which is instrumented with autonomic managers to monitor assertions, invariants, regression tests, or the hysteresis of non-functional requirements will be more self-sufficient, robust and, at the same time, be able to adapt to changes in its context and operating environment. This greatly enhances the ability of a system to determine and localize problems.

In this paper we have proposed an approach based on the reflective capabilities of programming languages to build monitoring into a software system with adaptive capabilities. We have argued that monitoring should be built into an adaptive system from the ground up with autonomic observers. Addressing monitoring is important because it constitutes a tie between the managed element (i.e., the software system under observation) and the autonomic manager (cf. Figure 1). Thus, monitoring should be a concern during the whole system life cycle, including requirements engineering, design, architecture, and, implementation. Furthermore, there is a need for standard interfaces and event formats.

Monitoring in a Java environment can be achieved with Java's reflective capabilities, namely metaobjects and dynamic proxy classes. In this paper, we have described how proxies can be leveraged for monitoring and how *adaptive monitors* can be realized via chaining of proxies. Furthermore, we have discussed an example of an autonomic observer that monitors the Java exceptions that the system under observation generates. This observer monitors the system continuously over a long time period. The assumption is that during normal operation exceptions are raised in predictable patterns, but happen in bursts when the system exhibits anomalous behavior.

For future work, we plan to build an autonomic system that uses our monitoring approach. Alternatively, we may be able to take a suitable existing Java system and inject proxies into it as well as augment it with an autonomic observer. Such a system could then serve as a basis for a case study to show the feasibility and trade-offs of our approach. For example, it would allow us to quantify the overhead that is imposed by proxies and other reflective monitoring mechanisms. Furthermore, it would allow us to study our hypothesis that monitoring raised exceptions can be used to indicate normal vs. anomalous system behavior.

8. ACKNOWLEDGMENTS

This work was funded in part by the National Sciences and Engineering Research Council (NSERC) of Canada (CRDPJ 320529-04 and CRDPJ 356154-07), IBM Corporation, and CA Inc. via the CSER Consortium. We also would like to acknowledge comments by Qin Zhu and Lin Lei on this paper.

REFERENCES

[1] ASM home page, WebObject Consortium, http://asm.objectweb.org/

[2] BCEL home page. The Apache Jakarta Project, http://jakarta.apache.org/bcel/

[3] Lionel C. Briand, Yvan Labiche, Johanne Leduc, Tracing Distributed Systems Executions Using AspectJ. *21st IEEE International Conference on Software Maintenance (ICSM 2005),* pp. 81-90, Budapest, Hungary, September 2005.

[4] Anil Chawla and Alessandro Orso. A Generic Instrumentation Framework for Collecting Dynamic Information, *ACM SIGSOFT ACM Software Engineering Notes*, Vol. 29, No. 5, pp. 1-4, September 2004.

[5] Shigeru Chiba. Javassist home page, http://www.csg.is.titech.ac.jp/~chiba/javassist/

[6] Dagstuhl Seminar 08031 on Software Engineering for Self-Adaptive Systems, January 13-18, 2008, http://www.dagstuhl.de/08031

[7] François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-oriented Programming. *Proceedings of the IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI,* pp. 29-38, August 1995.

[8] Guy Dumont and Mihai Huzmezan. Concepts, Methods and Techniques in Adaptive Control, *Proceedings American*

Control Conference (ACC 2002), Vol. 2, pp. 1137-1150, 2002.

[9] IBM Corporation. An Architectural Blueprint for Autonomic Computing, White Paper, 4th Edition, June 2006, http://www.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf

[10] IBM Corporation. CBE Common Base Event, IBM developerWorks, http://www.ibm.com/developerWorks/library/specification/ws-cbe/

[11] Ira Forman and Nate Forman. *Java Reflection in Action,* Manning Publications, October 2004.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, November 1994.

[13] Brian Goetz. Java theory and practice: Decorating with dynamic proxies, IBM developerWorks, August 2005, http://www.ibm.com/developerworks/java/library/j-jtp08305.html

[14] Gang Huang, Tiancheng Liu, Hong Mei, Zizhan Zheng, Zhao Liu, and Gang Fan. Towards Autonomic Computing Middleware via Reflection, *Proceedings 28th ACM COMPSAC Conference on Computer Software and Applications (COMPSAC 2004),* Vol. 1, pp. 135-140, 2004.

[15] Duncan Johnston-Watt. Under New Management, *ACM Queue,* Vol. 4, No. 2, March 2006.

[16] Jeff O. Kephart and David M. Chess. The Vision of Autonomic Computing, *IEEE Computer,* Vol. 36, No.1, pp. 41-50, January 2003.

[17] Jeff Kramer and Jeff Magee. Self-Managed Systems: An Architectural Challenge, *FOSE 2007: 2007 Future of Software Engineering, 29th ACM/IEEE International Conference on Software Engineering (ICSE 2007),* Minneapolis, Minnesota, USA, pp. 259-268, May 2007.

[18] Marin Litoiu, Murray Woodside, and Tao Zheng. Hierarchical Model-based Autonomic Control of Software Systems. *ICSE 2005 Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005),* Workshop at *27th ACM/IEEE International Conference on Software Engineering (ICSE 2005),* St. Louis, Missouri, USA, pp. 34-40, May 2005.

[19] Qusay H. Mahmoud. Understanding Network Class Loaders, Sun Developer Network, October 2004, http://java.sun.com/developer/technicalArticles/Networking/classloaders/

[20] Hausi A. Müller, Mary Shaw, Mauro Pezzè. Visibility of Control in Adaptive Systems, *Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008),* Workshop at *30th ACM/IEEE International Conference on Software Engineering (ICSE 2008),* Leipzig, Germany, May 2008. In press.

[21] Mary Shaw. Beyond Objects, *ACM SIGSOFT Software Engineering Notes*, Vol. 20, No. 1, pp. 27-38, January 1995.

[22] Software Engineering Institute. *Ultra-Large-Scale Systems: The Software Challenge of the Future*, 134 pages, ISBN 0-9786956-0-7, July 2006, http://www.sei.cmu.edu/uls/

[23] Sun Microsystems. Reflection, http://java.sun.com/javase/6/docs/technotes/guides/reflection

[24] TPTP home page, Eclipse, http://www.eclipse.org/tptp/

[25] WSDM Web Services Distributed Management, OASIS, http://www. oasis-open.org/committees/tc_home.php

APPENDIX A—Selected Java Code for Adaptive Exception Monitor

Code Listing 1: MyInvocationHandler

```
import java.lang.reflect.*;

public interface MyInvocationHandler
{
    public Object createProxy( Object obj );
    public void addMyEventListener(MyEventListener myEventListener);

    public MONITOR_LEVEL getMonLevel();
    public void setMonLevel(MONITOR_LEVEL mon_level);

    public void setParent(Proxy prxy);
    public Object getNextTarget();
    public Object getPrevTarget();
    public Object getRealTarget();

    public void setNextTarget(Object obj);
    public void setPrevTarget(Object obj);
    public void setRealTarget(Object obj);

    public boolean isFront();
    public boolean isPrimary();
    public boolean isBack();

    public void addInFront(Proxy prxy);
    public void addInBehind(Proxy prxy);
    public void addToFront(Proxy prxy);
    public void addToBack(Proxy prxy);
    public void removeFront();
    public void removeBack();

    public void remove(Proxy prxy);
    public boolean contains(Proxy prxy);

}
```

Code Listing 2: AbstractInvocationHandler

```
import java.lang.reflect.*;

public abstract class AbstractInvocationHandler
                implements InvocationHandler, MyInvocationHandler
{
    protected Object parent; //the Proxy that contains this InvocationHandler
    protected MONITOR_LEVEL mon_level = MONITOR_LEVEL.LOW;

    protected Object prevTarget = null;
    protected Object nextTarget = null;
    protected Object realTarget = null;

    protected Controller controller = Controller.getInstance();
    protected Logger logger = Logger.getInstance();
    protected EventListenerList listenerList = new EventListenerList();

    // ...

}
```