Facilitating Autonomic Computing Using Reflection

by

Dylan Dawson
Bachelor of Engineering, University of Victoria 2003

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Dylan Dawson, 2009
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
Supervisory Committee

Facilitating Autonomic Computing Using Reflection

by

Dylan Dawson
B. Eng., University of Victoria, 2003
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 potentially able to analyze its effectiveness by evaluating quality criteria and then self-tune to improve its operations.

To be able to observe and possibly orchestrate continuous evolution of software systems in a complex and changing environment, we need to push monitoring and control of evolving systems to unprecedented levels. This thesis proposes implementing monitoring and evolution in adaptive systems using autonomic elements that rely on the reflective capabilities of the language in which the system is implemented. Such monitoring will allow a system to detect anomalous internal behaviour, and infer that changes to the operating context or environment have occurred.
# Table of Contents

Supervisory Committee ........................................................................................................... ii
Abstract .................................................................................................................................. iii
Table of Contents ...................................................................................................................... iv
List of Tables ............................................................................................................................ vi
List of Figures ........................................................................................................................... vii
Acknowledgments ................................................................................................................... viii
Dedication ................................................................................................................................. ix

Chapter 1. Introduction ............................................................................................................. 1
  1.1 Autonomic Computing ......................................................................................................... 3
  1.2 Java Reflection .................................................................................................................. 5
    1.2.1 Metaobjects ................................................................................................................ 6
  1.3 Related Work .................................................................................................................... 7
  1.4 Motivation and Objectives ............................................................................................... 8
  1.5 Thesis Outline .................................................................................................................. 9

Chapter 2. Monitoring using Reflection .................................................................................. 10
  2.1 Dynamic Loading ............................................................................................................. 10
  2.2 Reflective Construction ................................................................................................... 11
  2.3 Dynamic Proxies ............................................................................................................. 13
  2.4 Monitoring with Proxies ................................................................................................. 16
    2.4.1 Proxy Chaining .......................................................................................................... 17
  2.5 Summary ......................................................................................................................... 20

Chapter 3. Structural Evolution using Reflection ..................................................................... 21
  3.1 Custom Class Loaders ...................................................................................................... 21
  3.2 Dynamic Class Replacement .......................................................................................... 22
    3.2.1 Designing for Replacement ....................................................................................... 22
    3.2.2 Implementing Replacement ...................................................................................... 24
  3.3 Summary ......................................................................................................................... 27

Chapter 4. Implementation ..................................................................................................... 28
  4.1 Reflective, Adaptive Exception Monitoring ..................................................................... 28
  4.2 Controller ......................................................................................................................... 32
    4.2.1 Effecting Behavioural Change ................................................................................... 33
    4.2.2 Effecting Structural Change .................................................................................... 34
  4.3 Logger ............................................................................................................................... 34
  4.4 Closing the Control Loop ............................................................................................... 35
  4.5 Summary ......................................................................................................................... 38
5. Analysis ............................................................................................................. 39
  5.1 Code Complexity .......................................................................................... 39
  5.2 Performance Considerations ....................................................................... 41
  5.2.1 Construction Overhead Analysis ............................................................ 43
  5.2.2 Execution Overhead Analysis – Single Proxy .......................................... 45
  5.2.3 Execution Overhead Analysis – Proxy Chains ......................................... 46
  5.3 Security Considerations ............................................................................ 47
  5.4 Limitations .................................................................................................. 48
  5.5 Summary ..................................................................................................... 49

6. Conclusions and Future Work......................................................................... 51
  6.1 Summary ..................................................................................................... 51
  6.2 Contributions ............................................................................................... 52
  6.3 Future Work ................................................................................................ 53

Bibliography .......................................................................................................... 54

Appendix A ............................................................................................................ 57
Appendix B ............................................................................................................ 58
 Appendix C ........................................................................................................ 64
Appendix D ............................................................................................................ 65
Appendix E ............................................................................................................ 66
Appendix F ............................................................................................................ 69
List of Tables

Table 1: Methods of Class for Constructor Introspection ............................................. 12
Table 1: Construction Overhead – No Proxy Delay ......................................................... 44
Table 1: Construction Overhead – 10 ms Proxy Delay .................................................. 44
Table 1: Execution Overhead – No Proxy Processing ..................................................... 45
List of Figures

Figure 1: Autonomic Element [9] .................................................................................. 4
Figure 2: Pre and Post Method Delegation Intercession ............................................. 15
Figure 3: Compositional Intercession ......................................................................... 19
Figure 4: Design for Dynamic Class Replacement [11] ............................................. 23
Figure 5: Overall System Architecture ....................................................................... 36
Figure 6: Execution Time versus Number of Proxies .............................................. 46
Acknowledgments

First and foremost, I would like to acknowledge the kind support of my supervisor Dr. Hausi Müller without whom, this thesis may never have come to fruition. As well, I would like to thank my colleagues, friends and family—especially my mother, who pushed me even harder than Hausi.

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 Canadian Consortium for Software Engineering Research (CSER).
**Dedication**

I would like to dedicate this thesis to my son and daughter, Bae and Wren Dawson.

Children are the meaning of life—I love you both right up to the moon and back.
Chapter 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 can be used to 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 to be 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. Furthermore, reflective mechanisms can be employed to implement structural and behavioural change when the context of the system has drifted to the point that evolution is necessary. The behaviour of individual objects can be dynamically modified or even replaced at run-time, increasing the flexibility and
This thesis explores how autonomic elements and reflection technology can be used to instrument Java programs from the ground up for the purpose of monitoring and effecting changes to the system's state, behaviour and even structure.

1.1 Autonomic Computing

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 longevity of the system. This effectively closes the control loop of the governing autonomic element.
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.

![Diagram of Autonomic Element](image)

**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.

Describing and implementing 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 behaviour).

1.2 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, with respect to functionality and performance, 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 behaviour. 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.

1.2.1 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.

1.3 Related Work

There are many approaches to instrumenting existing systems with the goal of obtaining information about their run-time behaviour (e.g., sequences of method invocation or profiling of execution times). In Java, the byte code 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\(^1\) to inspect and manipulate Java classes at the level of JVM\(^2\) 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 byte code 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 behaviour, 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

\(^1\) API – Application Programmers Interface
\(^2\) JVM – Java Virtual Machine
modify the behaviour of a J2EE\textsuperscript{3} application server using reflection mechanisms. Their approach was successfully able to perform self optimization in a standard J2EE benchmark test.

Aspect-oriented programming languages are also used to instrument code. For example, Briand \textit{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.

\textbf{1.4 Motivation and Objectives}

The objective of this thesis is to explore and identify potential uses of the Java reflection API in constructing autonomic systems. Specifically, this thesis shows how the touch-points or manageability endpoints of an Autonomic Manager can make use of introspection, intercession and Java proxies to implement monitoring for the purpose of effecting behavioural and structural change in a running system. By detecting anomalous behaviour within the system, we can infer that there has been a change to the operating context or environment that the system is contained within.

\textsuperscript{3} J2EE—Java 2 Platform, Enterprise Edition
1.5 Thesis Outline

Chapter 2 describes monitoring using Java reflection, and discusses an approach to instrumenting programs from the ground up using Java’s reflective capabilities, especially dynamic proxy classes. Chapter 3 describes techniques for dynamically altering the behaviour and structure of a Java application at run-time. Chapter 4 introduces an example autonomic system based on this approach. Chapter 5 provides an analysis of the benefits, shortcomings and pitfalls of this approach. Chapter 6 closes the thesis with conclusions and future work.
Chapter 2. Monitoring using Reflection

The Java reflection API provides two important facilities for implementing monitoring: dynamic loading, which allows for the loading and usage of objects not known at runtime, and the Proxy class. When these are combined together, applications can adaptively compose monitoring behaviours—even those not implemented at design-time.

2.1 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 runtime. 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:

```java
Class myClass = Class.forName("demo.Impl");
```
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 behaviour. 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 behaviour 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].

2.2 Reflective Construction

Constructing an object that had been loaded dynamically can be accomplished in one of two ways: using the class object itself, or using the metaobjects representing the class’s constructors.
First, the `newInstance` method of `Class` creates and returns a reference to a new instance of the class represented by the `Class` object. Calls to this method will always result in constructing the object using its no argument constructor. At first glance, this may seem like an hindrance as newly created objects must now be initialized without the use of a constructor. The authors of *Design Patterns* [12] however, prescribe that this is actually desirable since developers should always program to an interface, and not the implementation. This is because an interface defines the service provided by the object, regardless of implementation, and any object that conforms to that interface can be used interchangeably.

Constructing objects this way presents two difficulties:

1. Dynamically loaded classes, specifically those that are not known at design time, may be hard to initialize.
2. In Java, it is not only possible, but sometimes desirable (as is the case with the Singleton design pattern) to make the no argument constructor inaccessible (i.e., private).

### Table 1: Methods of Class for Constructor Introspection

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor <code>getConstructor(Class[] parameterTypes)</code></td>
<td>Returns the public constructor with specified argument types.</td>
</tr>
<tr>
<td>Constructor <code>getDeclaredConstructor(Class[] parameterTypes)</code></td>
<td>Returns the constructor with specified argument types.</td>
</tr>
<tr>
<td>Constructor <code>getConstructors()</code></td>
<td>Returns an array containing all of the public constructors supported by the target.</td>
</tr>
<tr>
<td>Constructor <code>getDeclaredConstructors()</code></td>
<td>Returns an array containing all of the public constructors supported by the target.</td>
</tr>
</tbody>
</table>
Thankfully, in these cases we can still use the metaobjects representing the class’s constructors or other initialization methods. For example, we can refer to Table 1 for information on how to access the constructors of an object reflectively.

### 2.3 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:

```java
class MyClass implements IClass { ... };  
MyClass obj = new MyClass(...);

Proxy prxy = Proxy.newProxyInstance
            (  
                obj.getClass().getClassLoader(),
                obj.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 behaviours 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):

```java
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].
The use of proxy classes increases flexibility and 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.

### 2.4 Monitoring with Proxies

In Java there are two techniques to facilitate behavioural 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 the most convenient method for implementing low level monitoring. For example, instances of a single proxy class that implements Exception 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 behaviours such as tracing and profiling can be dynamically composed and enabled or disabled at run-time if each behaviour is encapsulated in a proxy. Such dynamic composition can be easily achieved by chaining proxies together.

2.4.1 Proxy Chaining

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 behaviours 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].
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 behaviour based on this knowledge [11].

2.5 Summary

One way to make applications more flexible is to design them with the ability to incorporate new code. As the context of an application shifts over time, requirements may diverge from the application’s original implementation. Dynamic loading allows applications to incorporate objects that were not available at design-time. This flexibility will allow applications to withstand changes in requirements and operating context.

Delegation through Java’s dynamic proxies provides a convenient way to intercede before and after method calls on an object. This kind of intercession allows developers to insert behaviours such as Exception monitoring, tracing, profiling and logging. Individual behaviours can be encapsulated in a single InvocationHandler from which a proxy can be created for any object in the system. This allows behaviour implementing code to be encapsulated and re-used throughout the system. Further, these behaviours can be composed together in chains to implement adaptive behaviours.

When dynamic loading and Java proxies are combined, we have the ability to introduce new behaviours and compose them together with existing ones at run-time.
Chapter 3. Structural Evolution using Reflection

In addition to providing monitoring capabilities and behavioural change, reflection also gives us the ability to effect structural change. With careful design, applications can dynamically replace objects at run-time. This increases the overall flexibility of the application by allowing developers to update objects in response to system degradation due contextual shift or the divergence of requirements.

3.1 Custom Class Loaders

Chapter 2 showed how new objects can be introduced into a running system through the use of dynamic loading via `Class.forName(String)`. This function makes use of the default system class loader to load the byte code required to construct the specified class. As discussed by Liang and Bracha, it is possible for more than one instance of a user-defined class to exist within a single JVM, but if and only if the class is loaded by an equivalent number of unique, class loaders. In this case however, the classes are not considered to be of the same type since a class is uniquely identified by the combination of class name and class loader [26]. The problem of type equivalence during dynamic class replacement can be overcome however, with the appropriate combination of custom class loaders, interfaces, abstract classes and dynamic proxies.
3.2 Dynamic Class Replacement

In systems that are intended for continuous operation, structural evolution presents a very challenging problem. By modifying the compiler or the JVM, it is possible to replace active (i.e., loaded and instantiated) classes in a running application [28]. These approaches however, are inherently complex and require systems to adopt either a custom compiler or custom JVM. Fortunately, it is possible to replace active classes under certain conditions using only the reflection API and suitable design.

3.2.1 Designing for Replacement

When considering the replacement of an active class, two important requirements must be considered. First, you must maintain references to all active instances of the class that will be replaced, and second, you must have a method for migrating instances from one implementation to another. The diagram in Figure 4 illustrates a design that will allow an active class can be replaced at run-time.

In this design, the target object is separated from the client by a Java proxy. The proxy and the abstract class both implement the target interface—this effectively hides the current implementation from the client. The abstract class is also responsible for maintaining references to all active instances of its subclasses. It is important to note that the replacement class exists in a different package than the one that it is replacing—this eliminates the need to rename the class as long as each class is loaded with a different classloader.
Another important specification of this design is that the replacement class must have a mechanism for producing a replacement object given the original object, captured by the `evolve() : IObject` method. This allows old instances to be mapped to instances of the replacement class while isolating the details of evolution to the replacement class. This is an example of the Strategy design pattern [12].

![Diagram](image)

**Figure 4: Design for Dynamic Class Replacement [11]**
This design uses the `newInstance():IObject` method in the `AbstractObject` class to create instances of the implementation of `ObjectImpl`, but the caller receives a proxy to that instance. In addition, the instance is stored in a static list of `AbstractObject` named instances, which is used to locate each instance of the implementation of `ObjectImpl`. The method `reload(String)` is used initially to load the implementation and subsequently to change the implementation. This design is another application of the Abstract Factory pattern [12]. Because of the use of different class loaders, the implementation classes have the same names. In some contexts, this can be an important advantage.

### 3.2.2 Implementing Replacement

Appendix C and D show the code for `AbstractObject` and `ObjectIH` respectively. First, let us examine `AbstractObject` which defines two static methods, `newInstance():IObject` and `reload(String)`.

The `newInstance()` method uses the class object of the current implementation (stored in `implClass`) to construct a new instance of the `ObjectImpl`, which is hidden behind a newly created proxy. Reloading the class involves evolving the instances. Consequently, all of the extant instances must be tracked, which is done by adding a weak reference to the proxy to a list of named instances.
The reload method loads the new implementation and evolves each existing instance of the old implementation. The method `reload(String)` is responsible for:

1. Constructing a new class loader and loading the new implementation.
2. If the directory instance variable is null, loads the first implementation. That is, `reload` is also used for the first load.
3. For reload calls after the first, evolves each instance of the object. This is done by iterating through the list of instances and invoking `evolve()`.
4. The evolve method returns a new object that is suitable for the new implementation. The new object is stored as the target of the proxy instance, which is known to the clients.
5. The list of instances is replaced.

AbstractObject has an additional nuance to it. Its instances list does not contain direct references to the proxies. If it did, the garbage collector would never free an instance of `ObjectImpl` when the client finishes with it. Instead, weak references are stored in the instances list. `WeakReference` is part of `java.lang.ref`. Weak references are constructed with a reference to another object, its referent. The referent can always be retrieved with a get method, but weak references do not prevent their referents from being garbage collected. After garbage collection of the referent, the get method on a weak reference returns null. When a class is reloaded, a new instance list is created with only weak references with non-null referents.
This design also contains an application of the Proxy pattern (that is, two distinct objects are required). If \texttt{java.lang.reflect.Proxy} were being used to implement some other pattern, \texttt{AbstractObject} might also implement the \texttt{InvocationHandler} interface. However, in this example, the invocation handler must be retargeted to different implementations of \texttt{IObject}, which implies that the \texttt{InvocationHandler} must be distinct from the target.

Appendix D presents the code for the invocation handler, \texttt{ObjectIH}. This invocation handler is straightforward in that its \texttt{invoke} method merely delegates the call to the target. It has a \texttt{setTarget(IObject)} method, which permits the \texttt{ObjectImpl} object to evolve after its class is replaced. The invocation handler adds value because it hides the real \texttt{ObjectImpl}, making the replacement transparent to clients that use the product.

It is important to note that the replacement of one implementation of \texttt{ObjectImpl} with another could be accomplished with implementation classes that have different class names. Typically however, we would like to replace a component rather than a single class. In this case, changing all the names of the classes in the component and the internal references to those classes is a tedious and error-prone process. It is best not to make all of those changes.

Also, even though the use of packages provides a compile-time namespace, it is still preferable to have distinct class loaders for each package. If both the component and its
replacement have a large number of common names, the use of distinct class loaders ensures that no reference to the original component can leak into the replacement.

### 3.3 Summary

Using the techniques introduced in Section 3.2, we have seen that it is possible to replace and active class during run-time using only the Java reflection API and a modicum of clever design. This facility is important because is allows running applications to hot swap individual objects or even entire components as the need for structural system evolution arises. In the context of autonomic computing, this step can be mapped to the ‘execute change’ phase of the autonomic control loop. When combined with the monitoring techniques presented in Chapter 2, we can see that we are one step closer to closing the autonomic control loop – indeed, we can now see how reflection can be used to facilitate the implementation of the first and last phases of the autonomic control loop. Reflection has provided us with the ability to adaptively monitor the target system for abnormal behaviour, and replace components that contribute to this errant behaviour. Chapter 4 presents a sample application, built using these techniques, that is able to reflectively monitor a system and change the behaviour of specific objects. Moreover, the system is able to dynamically load new code to introduce new behaviours or to completely replace malfunctioning objects.
Chapter 4. Implementation

The previous chapters illustrated how Java reflection and dynamic proxies can be leveraged to facilitate the design of autonomic managers. The example application in this section shows how to build such an autonomic observer to monitor Java Exceptions over long periods of time and effect behavioural and structural changes to the system as required. The assumption is that during normal operation, Exceptions are raised in predictable patterns and in bursts during exceptional behaviour. The exceptional behaviour might be due to unexpected changes in the system’s internal state or its environment. By detecting and localizing raised Exceptions, the system will be able to determine that changes in the environment have occurred. Recognizing such changes using autonomic observers will give the system a chance to adapt and evolve.

4.1 Reflective, Adaptive Exception Monitoring

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 InvocationHandler 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 behaviour 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 behaviours 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, MEDIUM, 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 (discussed in Section 5.2).

The MyInvocationHandler interface is implemented by the AbstractInvocationHandler mentioned in Section 2.4.1. The code listing in Appendix B shows that this base implementation contains a reference to the centralized Controller and a Logger. The Controller and Logger (described in Sections 5.3 and 5.4, respectively) 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 behaviour and structure of the proxy chain in response. Using the techniques
presented in Section 3.2.2 for dynamically replacing a class at run-time, the target of the proxy chain may even be hot swapped with a new implementation.

In the example application, 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, and to hide the implementation of the target.

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.

6. When it had been determined that the target object needs to be replaced, the Controller can notify that target. The target can then replace itself with a new implementation if one is available.

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 this 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.

4.2 Controller

The Controller is implemented using the Singleton design pattern [12]. This ensures that there is one, and only one, centralized manager whose purpose is to monitor and orchestrate behavioural and structural changes in the system. Code Listing 5 in Appendix E illustrates that the Controller is responsible for managing the primary proxies (and hence the proxy chains) of the objects we wish to monitor. The
Controller maintains references to all active primary proxies in an ArrayList, and supports functionality for registering and un-registering these proxies as required.

Upon system start-up, the Controller refers to an external XML configuration file. This file tells the Controller which objects in the system are to be monitored, and how to manage the proxy chains for individual objects when they are instantiated. In most cases, each object that we wish to monitor will only have the Monitoring Proxy added to the proxy chain. This XML file can be configured however, to instruct the Controller to initially add other proxies to the proxy chain, and at what degree of aggressiveness they are to be employed.

The Controller also maintains a reference to a centralized repository. The repository is used to collect and organize Exceptions that are generated by the system. As new Exceptions are added to the repository, Event notifications are sent to the Controller which can then perform problem analysis on the Exceptions that are generated by individual objects.

### 4.2.1 Effecting Behavioural Change

When an object that is being monitored begins to throw Exceptions at a temporal interval above a configurable threshold, the Controller will revisit how the proxy chain for that object is configured. The Controller may decide to monitor with varying levels of aggressiveness, add new proxies to the proxy chain to gather more
information, or even re-organize the ordering of the proxies. Each individual proxy in a proxy chain can be reconfigured to (for example) monitor, profile, and/or trace with varying levels of aggressiveness.

4.2.2 Effecting Structural Change

Likewise, when an object that is being monitored begins throwing too many Exceptions, it may be determined that the object is not functioning properly within the current context or operating environment of the system. The Controller can send a notification to the target of the proxy chain indicating that it should replace itself (and all other extant instances of itself) with a newer version, if one exists.

In this example application, this scenario requires human intervention. A developer will be notified that an object or component in the system is no longer functioning properly and is required to make the code changes necessary to remedy the problem. This new object can be placed in an appropriate, predetermined location, loaded into the system dynamically, and hot swapped with the original version during run-time.

4.3 Logger

As with the Controller, the Logger is also implemented using the Singleton design pattern. Extensions of the AbstractInvocationHandler mentioned in Section 2.4.1 make use of this generalized logger which can be customized to log
Exceptions 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.

4.4 Closing the Control Loop

The diagram in Figure 5 illustrates the overall architecture of the reflective, adaptive Exception monitor for a single object in the system. By adding a centralized controller and logger to the techniques described in Chapters 2 and 3, we are effectively able to
implement the first and last phases of the autonomic element described in section 1.1. Specifically, the monitoring phase is accomplished through the use of reflective intercession, and the execution (i.e., effecting behavioural and/or structural change) is accomplished with reflective intercession (behavioural change) and/or dynamic class replacement (structural change).

Figure 5: Overall System Architecture

This diagram shows the target object separated from the user of this object (the client) by a proxy chain. The primary proxy is positioned at the head of the proxy chain and registers itself with the Controller. The purpose of the primary proxy is to maintain the
proxy chain and to hide the concrete implementation of the target object. The Controller refers to an external XML configuration file to determine which intercessional capabilities to add to the proxy chain. During intercession, the tracing proxy records which methods are called and when, and the monitoring proxy logs any Exceptions thrown by the target to the repository (i.e., knowledge base).

When an Exception is logged to the repository, the Controller is notified and will perform an analysis on the Exception history for the target object. If the frequency of raised Exceptions exceeds a configurable threshold, the Controller can now refer again to the XML file for instructions (or policy) regarding measures of remediation. The aggressiveness of the monitoring proxy can be increased to monitor more frequently, or for more types of Exceptions, or the proxy chain may be altered to include a new proxy, for example, a profiling proxy that records method execution times.

The information gathered by the proxy chain can then be used to manually determine if an object or component in the system needs to be updated or replaced. A developer will perform the required maintenance, and update the XML configuration file to notify the Controller that the target object or objects can be replaced, and with which object definition to replace it.
4.5 Summary

The sample application shown in the previous section of Chapter 4 illustrates how the reflective API of the Java programming language can be used to facilitate the construction of autonomic managers. Specifically, reflection can be used during the monitoring and execute change phases of the autonomic control loop. Using Java proxies, we are able to intercede on method calls for the purpose of monitoring Exceptions. As well, we are able to dynamically load and chain new or existing proxies to adapt to changing demands. When the system under observation degrades beyond a certain point, Java proxies and proper design for class replacement can further assist us by providing the capability to swap classes or even whole components within the system dynamically, at run-time. The limitations of this design are discussed further in Chapter 5.
5. Analysis

This chapter is intended to address the four main issues that have impeded the use of reflective APIs: code complexity, performance penalties, security and limitations on remote reflection. Using proxies to delegate method calls will invariably incur code complexity and performance penalties, while dynamically loading new code into a system could potentially introduce malicious code. These considerations must be made when using any reflective API. Also discussed within this chapter is the inability of Java’s reflective API to reflect on remote objects.

5.1 Code Complexity

The use of reflection in the manner described in Chapters 2, 3 and 5 requires little more than becoming familiar with the Java reflection API (which consists of only 13 classes and 9 interfaces) and a few basic design patterns such as the Factory, Singleton, Strategy and Proxy. Having said that, newcomers to this API will have to begin developing code that requires thinking at one more level of abstraction (i.e., the meta-level). There are certain pitfalls that developers will have to be aware of if they wish to use this API effectively.

As an example, developers need to be aware of the complications of passing a proxy as an argument into contexts that are expecting a real object. Consider the following code listing:
public interface Shape
{
    public float getPerimeter();
}

public class Circle implements Shape
{
    private float perimeter;

    public Circle(float radius)
    {
        this.perimeter = 2*Math.PI*radius;
    }

    public float getPerimeter()
    {
        return this.perimeter;
    }

    public boolean equals( Object obj )
    {
        if( obj instanceof Circle)
        {
            Circle c = (Circle) obj;
            return this.perimeter == c.perimeter;
        }
    }
}

When two shapes are compared for equality such as myCircle.equals(yourCircle) and yourCircle is a reference to a proxy for an object of type Circle, the comparison will always fail. This will always compare an object of type Circle to an object of type Proxy. Of course, these problems can be overcome as long as the reflection API and the associated design patterns are observed and understood. In this case, we simply access the values we wish to compare for equality through the interface instead of the concrete class because the proxy instance is bound to
the interface and intrinsically understands how handle this. The general rule is that proxied classes with methods that accept parameters that has that class as its type should be accessed through the interface, such as:

```java
public class Circle implements Shape {
    private float perimeter;

    public Circle(float radius) {
        this.perimeter = 2*Math.PI*radius;
    }

    public float getPerimeter() {
        return this.perimeter;
    }

    public boolean equals( Object obj) {
        if( obj instanceof Shape) {
            Shape s = (Shape) obj;
            return this.perimeter == s.perimeter;
        }
    }
}
```

Of course, this can be problematic if developers using reflection do not have the foresight to support the necessary accessors in the interface.

### 5.2 Performance Considerations

The reflective capabilities in Java are largely due to delaying the binding of the names of methods and fields. This is typically done at compile time, but when binding is
delayed until run-time there will be a performance impact in the form of run-time searches and checks. Within the sample application presented in Chapter 4, the performance associated with reflection can be divided into two categories [11]:

1. Construction overhead

   The time it takes to perform modifications to a class object during construction. This introduces latency when constructing a proxy class or instance, and is exacerbated by the use of the Factory pattern. This is however, just a one time cost, and should be considered insignificant in long running systems.

2. Execution overhead

   The time added to the service supplied by an object or component because of its reflective features. Within the application presented in Chapter 4, latency is introduced by delegating method calls through one or more proxies. Execution overhead will recur throughout the lifetime of the system and has the highest impact.

The execution overhead of a proxy chain is largely determined by the number of proxies in the chain, as well as the amount of pre and post-processing that each proxy performs. This run-time performance can be seen as a disadvantage to using reflection in systems that require low latency—for example, systems that are subject to real-time constraints. The same argument has also been applied to the move to high-level languages, virtual memory, object oriented programming and garbage collection. While it
is true that reflection presents a trade-off between flexibility and speed, multi-gigahertz processors and seemingly unlimited storage are becoming more and more available. As this trend continues, the return on investment of applying these abstractions becomes more and more attractive.

5.2.1 Construction Overhead Analysis

In this section, we will quantify the overhead that is associated with proxy construction. To quantify the construction overhead, we begin by measuring the time that it takes to construct a simple Object once, multiple times, and on average. This process is then duplicated with the use of a proxy to the target instead of the target itself. The results are shown in Table 2.

The overhead associated with proxy construction is highly dependent on the processing that occurs during calls to the constructor of both the target and the proxy. Since we are only concerned with the overhead however, we perform the following analysis under the constraint that the proxy constructor will be empty. The constructor of the target Object on the other hand, contains only a 0.01 second delay. This delay is introduced into the target constructor so that we will be dealing with time intervals that are significant enough to measure.
Table 2: Construction Overhead – No Proxy Delay

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct target Object</td>
<td>15.000 ms</td>
</tr>
<tr>
<td>Construct 1000 target Objects</td>
<td>15625.000 ms</td>
</tr>
<tr>
<td>Average target Object construction time</td>
<td>15.625 ms</td>
</tr>
<tr>
<td>Construct proxy to target Object</td>
<td>16.000 ms</td>
</tr>
<tr>
<td>Construct 1000 proxies to target Object</td>
<td>15625.000 ms</td>
</tr>
<tr>
<td>Average proxy construction time</td>
<td>15.625 ms</td>
</tr>
</tbody>
</table>

From the data collected in Table 2, we can conclude that the average construction overhead associated with a proxy that has an empty constructor is negligible. This result is not surprising since the overhead of construction in this case is mostly dependent on the amount of processing done in the constructor of the proxy. If, on the other hand, we introduce a 0.01 ms delay in the constructor of the proxy, we expect to see that the construction overhead will be double that of constructing the target Object alone. This is confirmed in Table 3

Table 3: Construction Overhead – 10 ms Proxy Delay

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct target Object</td>
<td>15.000 ms</td>
</tr>
<tr>
<td>Construct 1000 target Objects</td>
<td>15625.000 ms</td>
</tr>
<tr>
<td>Average target Object construction time</td>
<td>15.625 ms</td>
</tr>
<tr>
<td>Construct proxy to target Object</td>
<td>31.000 ms</td>
</tr>
<tr>
<td>Construct 1000 proxies to target Object</td>
<td>31250.00 ms</td>
</tr>
<tr>
<td>Average proxy construction time</td>
<td>31.250 ms</td>
</tr>
</tbody>
</table>
5.2.2 Execution Overhead Analysis – Single Proxy

In this section, we will quantify the overhead that is associated with proxy delegation. To quantify the delegation overhead, we begin by measuring the time that it takes to call a single method on a target Object once, multiple times, and on average. This process is then duplicated with the use of a proxy to the target instead of the target itself. The results are shown in Table 4.

Again, the overhead associated with proxy delegation is highly dependent on the processing that occurs during before and after method delegation from the proxy. Since we are only concerned with the overhead, we perform the following analysis under the constraint that the proxy will perform pure delegation, with no pre or post processing. The method of the target Object on the other hand, contains only a 0.01 second delay. From the data collected in Table 4, we can see that the effect of pure method delegation is negligible in the case that the proxy performs no pre or post method delegation intercession.

Table 4: Execution Overhead – No Proxy Processing

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute method on target Object</td>
<td>15.000 ms</td>
</tr>
<tr>
<td>Execute method on target Object 1000 times</td>
<td>15625.000 ms</td>
</tr>
<tr>
<td>Average method execution time</td>
<td>15.625 ms</td>
</tr>
<tr>
<td>Execute method on proxy to target Object</td>
<td>15.000 ms</td>
</tr>
<tr>
<td>Execute method on proxy to target Object 1000 times</td>
<td>15625.00 ms</td>
</tr>
<tr>
<td>Average method execution through proxy time</td>
<td>15.625 ms</td>
</tr>
</tbody>
</table>
5.2.3 Execution Overhead Analysis – Proxy Chains

Of course, the experiment that produced the results in Table 4 is measuring the time that it takes for pure method delegation through a single proxy. In this section, we will quantify the overhead that is associated with delegation through a proxy chain. To quantify the delegation overhead, we begin by measuring the time that it takes to call a single method on a target Object through a single proxy, and compare that to the time that it takes when the proxy chain is lengthened to 100 in increments of 10. A delay of .01 ms is introduced during pre method delegation to produce measurable results. These results are shown in a graph of Execution Time versus Number of Proxies in Figure 6.

Figure 6: Execution Time versus Number of Proxies
From the graph in Figure 6, we can see the almost linear relationship between the number of proxies in a chain, and the time it takes to delegate a method call through that chain.

5.3 Security Considerations

With respect to security, there are a few concerns that need to be addressed. First, in the Java reflection API the meta object classes Field, Method, and Constructor are all derived from the same base class, AccessibleObject. AccessibleObject provides the ability to flag a reflected object as suppressing default Java language access control checks when it is used. The access checks for public, default (package) access, protected, and private members are performed when fields, methods or constructors are used to set or get fields, to invoke methods, or to create and initialize new instances of classes, respectively [29]. At first glance, this may seem to defeat the purpose of access modifiers entirely, as setting the accessible flag on a reflected object permits the manipulation of objects in a manner that would normally be prohibited. This mechanism does however, when used with care, allow for applications with sufficient privileges to perform sophisticated operations such as Java Object Serialization.

Another concern that must be addressed arises when we consider dynamically loading un-trusted code. When security is an issue, care must be taken to customize class loaders
to prevent malicious code from entering the system. These security concerns are often over inflated. In Java, there are four components to the Java security model:

1. The Java language and the Java Virtual Machine
2. The bytecode verifier
3. The security manager, and
4. The class loader architecture

The Java security model is designed to control the execution of un-trusted code, but is ineffectual if you load classes from an untrustworthy source. If you do customize a class loader to prevent the loading of un-trusted code, that design must use the Java security model to control the permissions of the loaded code. This can be done by subclassing java.security.SecureClassLoader, which is a concrete subclass of ClassLoader with protected constructors. SecureClassLoader supports the security model by ensuring that any call to its constructors is permitted by the supervising security manager [29].

5.4 Limitations

Another important facility provided by the Java programming language is the support for remote method invocation, or RMI. RMI allows methods to be invoked transparently across a network and facilitates system distribution. Unfortunately, the Java implementation of reflection and RMI are incompatible. Developers cannot reflect on a
remote applications [27]. Remote operations via RMI are restricted to operations that can be performed on a standard Java interface. Since you cannot declare fields or static methods in a Java interface, the RMI API can not allow a client to access them.

With Java RMI, it is a trivial to use sub-classing to make objects remotable. Any object you wish to access across the network must simply extend the `java.rmi.Remote` class. Unfortunately, in the reflection API, the `Class` object is declared `final`, and hence defies sub-classing. Fortunately, the authors of [27] have examined these issues, and have submitted recommendations for changes to the reflection API. It is their opinion that reflection and RMI should be orthogonal; a program should be able to reflect upon any object, local or remote—meaning that any object, base level or meta level, should be able to be made remotable.

5.5 Summary

The issues of code complexity, performance, security and network isolation cannot be ignored by developers using the reflection API. The issue of code complexity may stand to benefit from developing tools that support the development of software that makes use of the reflection API. For example, Eclipse plug-ins can easily be constructed to help developers transform a standard object into an object that is securely accessed only through a proxy. Likewise, tool support for proxying and hot-swapping key system objects or components, once created, could be leveraged quite profitably in the realm of
refactoring. As the reflection API evolves and hardware becomes faster, the issues of remoting and performance will also be removed as barriers to the use of reflection.

With respect to construction and execution overhead, it is important to consider that their effects on performance are highly dependent on the amount of processing that must be done in the constructor of the proxy as well as during method delegation. Otherwise, proxies that are easy to construct, and perform minimal pre and post processing are of little consequence to performance.
6. 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.

6.1 Summary

This thesis has proposed an approach based on the reflective capabilities of programming languages to building monitoring and control into a software system with adaptive capabilities. We have argued that monitoring and control 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 and control 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, as well as behavioural and structural change, can be achieved with Java’s reflective capabilities, namely metaobjects and dynamic proxy classes. This thesis described how proxies can be leveraged for monitoring and how *adaptive monitors* can be realized via chaining of proxies. Furthermore, this thesis has 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 behaviour. This behaviour can be localized, and remediated through the use of proxy chains and dynamic class replacement. This thesis also describes how Java proxies and a suitable design for replacement can be used to dynamically swap individual classes or even whole components of a system during run-time. It is the intent of this thesis to show that the Java reflection API can be leveraged quite effectively in the construction of autonomic managers, specifically during the first and last phases of the autonomic control loop—monitoring and executing system change (both behaviourally and structurally) respectively.

### 6.2 Contributions

In terms of concrete contributions, this thesis has:

- Described how proxies can be leveraged for autonomic monitoring and how *adaptive monitors* can be realized via the chaining of proxies.
• Implemented and examined an example of an autonomic observer that monitors the Java Exceptions that a system under observation generates.

• Shown how Java proxies and a suitable design for replacement can be used to dynamically swap individual classes or even whole components of a system during run-time for the purpose of facilitating autonomic computing.

• Shown that the Java reflection API can be leveraged quite effectively in the construction of autonomic managers, specifically during the first and last phases of the autonomic control loop—monitoring and executing system change both behaviourally and structurally, respectively.

6.3 Future Work

For future work, we plan to construct a complete 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 behaviour, and changes in the operating context or environment.
Bibliography


Appendix A

Code Listing 1: MyInvocationHandler

```java
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);
}
```
package monitoring;
import java.lang.reflect.*;
import logging.*;
import mopex.*;
import control.*;
import java.util.*;
public abstract class AbstractInvocationHandler implements InvocationHandler,
   MyInvocationHandler
{
    protected Object prevTarget = null;
    protected Object nextTarget = null;
    protected Object realTarget = null;
    protected Controller controller = Controller.getInstance();
    protected Logger logger = Logger.getInstance();
    protected javax.swing.event.EventListenerList listenerList = new
   javax.swing.event.EventListenerList();
    protected Object parent; //the Proxy that contains this InvocationHandler
    protected MONITOR_LEVEL mon_level = MONITOR_LEVEL.LOW;
    protected boolean isPrimary = false;
    public void setParent(Proxy prxy)
    { 
        this.parent = prxy;
    }
    public AbstractInvocationHandler()
    {
    }
    public AbstractInvocationHandler( Object target )
    {
        this();
        nextTarget = target;
        if ( nextTarget != null )
        {
            realTarget = findRealTarget(nextTarget);
            if (realTarget == null)
            {
                throw new Run-timeException("findRealTarget failure");
            }
        }
    }
    protected static final Object findRealTarget( Object t )
    {
        if ( !Proxy.isProxyClass(t.getClass()) )
        {
            return t;
        }
        InvocationHandler ih = Proxy.getInvocationHandler(t);
        if ( AbstractInvocationHandler.class.isInstance( ih ) )
        {
            return ((AbstractInvocationHandler)ih).getRealTarget();
        }
    }
else
{
    try
    {
        Field f = Mopex.findField( ih.getClass(), "target" );
        if ( Object.class.isAssignableFrom(f.getType()) &&
            !f.getType().isArray() )
        {
            f.setAccessible(true); // suppress access checks
            Object innerTarget = f.get(ih);
            return findRealTarget(innerTarget);
        }
        return null;
    }
    catch (NoSuchFieldException e)
    {
        return null;
    }
    catch (SecurityException e)
    {
        return null;
    }
    catch (IllegalAccessException e)
    {
        return null; // IllegalArgumentException cannot be raised
    }
}

protected void logMessage(String s)
{
    String what = "----" + this.getClass().getName() + "----" + "\n" +
        "|\n" +
        "| " + s + "\n" +
        "|\n" +
        "----" + "\n";
    logger.Log(what);
}

// This methods allows classes to register for MyEvents
public void addMyEventListener(MyEventListener listener)
{
    listenerList.add(MyEventListener.class, listener);
}

// This methods allows classes to unregister for MyEvents
public void removeMyEventListener(MyEventListener listener)
{
    listenerList.remove(MyEventListener.class, listener);
}

// This private class is used to fire MyEvents
void fireMyEvent(MyEvent evt)
{
    Object[] listeners = listenerList.getListenerList();
    // Each listener occupies two elements - the first is the listener class
    // and the second is the listener instance
    for (int i=0; i<listeners.length; i+=2)
    {
        if (listeners[i] == MyEventListener.class)
        {
            ((MyEventListener)listeners[i+1]).myEventOccurred(evt);
        }
    }
}

public MONITOR_LEVEL getMonLevel()
{
    return this.mon_level;
}
public void setMonLevel(MONITOR_LEVEL mon_level) {
    this.mon_level = mon_level;
}
public final Object getRealTarget() {
    return realTarget;
}
public void setRealTarget(Object obj) {
    this.realTarget = obj;
}
public final Object getNextTarget() {
    return nextTarget;
}
public final void setNextTarget(Object obj) {
    this.nextTarget = obj;
}
public final void setPrevTarget(Object obj) {
    this.prevTarget = obj;
}
public final Object getPrevTarget() {
    return this.prevTarget;
}

// Adds a proxy in front of THIS proxy, ie, one closer to the primary proxy.
// Add a proxy in front of THIS proxy, ie, one closer to the primary proxy.
public final void addInFront(Proxy prxy) {

    MyInvocationHandler mih = (MyInvocationHandler) prxy.getInvocationHandler(prxy);
    //1]
    mih.setNextTarget(this.nextTarget);
    //2]
    this.setNextTarget(prxy);
    //3]
    try {
        Proxy nxtProxy = (Proxy) mih.getNextTarget();
        MyInvocationHandler nextIH = (MyInvocationHandler) nxtProxy.getInvocationHandler(nxtProxy);
        mih.setPrevTarget(nextIH.getPrevTarget());
    }
    catch (Exception ex) {
        // we are adding to the end of the chain...
        mih.setPrevTarget(this.parent);
    }
    //4]
    try {
        Proxy nxtProxy = (Proxy) mih.getNextTarget();
        MyInvocationHandler nextIH = (MyInvocationHandler) nxtProxy.getInvocationHandler(nxtProxy);
        nextIH.setPrevTarget(prxy);
    }
    catch (Exception ex) {
        // do nothing, the next target is the real target
    }
}

// Add a proxy in behind THIS proxy
public final void addInBehind(Proxy prxy)
{
    if(this.isPrimary())
    {
        throw new RuntimeException("You cannot chain a proxy in behind the Primary proxy!");
    }

    //Otherwise
    Proxy prevProxy = (Proxy) this.getPrevTarget();
    MyInvocationHandler mih = (MyInvocationHandler)
        prevProxy.getInvocationHandler(prevProxy);
    mih.addInFront(prxy);
}

//Adds to the front of the chain, ie, immediately after the primary proxy
public void addToFront(Proxy prxy)
{
    if(this.isPrimary())
    {
        addInFront(prxy);
    }
    else
    {
        //Go get the primary, then addInFront(prxy).
        Proxy prev = (Proxy) this.getPrevTarget();
        MyInvocationHandler mih = (MyInvocationHandler)
            prev.getInvocationHandler(prev);
        while(!mih.isPrimary())
        {
            prev = (Proxy)mih.getPrevTarget();
            mih = (MyInvocationHandler) prev.getInvocationHandler(prev);
        }
        //prev should be the primary at this point so...
        mih.addInFront(prxy);
    }
}

//Adds to the back of the chain, ie, furthest from the primary proxy
public void addToBack(Proxy prxy)
{
    //First, see if we are the last proxy in the chain...
    if(this.getNextTarget() == this.getRealTarget())
    {
        MyInvocationHandler mih = (MyInvocationHandler)
            prxy.getInvocationHandler(prxy);
        mih.setNextTarget(this.getNextTarget());
        this.setNextTarget(prxy);
        mih.setPrevTarget(this.parent);
    }
    else
    {
        Proxy nxtPrxy = (Proxy) this.getNextTarget();
        MyInvocationHandler nxtIH = (MyInvocationHandler)
            nxtPrxy.getInvocationHandler(nxtPrxy);
        nxtIH.addToBack(prxy);
    }
}

public final void remove(Proxy prxy)
{
}

//Searches the rest of the chain after THIS... should only be used by the
//PrimaryIH.
public final boolean contains(Proxy prxy)
{
boolean contains = false;

/* bad recursion (think about it!!)
if(!this.isPrimary())
{
    Proxy prev = (Proxy)this.getPrevTarget();
    MyInvocationHandler prevIH = (MyInvocationHandler) prev.getInvocationHandler(prev);
    return prevIH.contains(proxy);
} */

//Otherwise...
MyInvocationHandler lookingFor = (MyInvocationHandler) proxy.getInvocationHandler(proxy);

Proxy next = null;
MyInvocationHandler nxtIH = null;
boolean notDone = true;
while(notDone)
{
    try
    {
        next = (Proxy) this.getNextTarget();
        nxtIH = (MyInvocationHandler) next.getInvocationHandler(next);
        if(nxtIH.equals(lookingFor))
        {
            contains = true;
            notDone = false;
        }
    }
    catch(Exception ex)//we've reached the end of the list.
    {
        notDone = false;
    }
}
return contains;
}

public final void removeFront()
{
}

public final void removeBack()
{
}

public final boolean isFront()
{
    boolean result = false;
    if(isPrimary != true)
    {
        //Get the prev and see if it is the primary...
        Proxy proxy = (Proxy) this.getPrevTarget();
        MyInvocationHandler mih = (MyInvocationHandler) proxy.getInvocationHandler(proxy);
        if(mih.getClass().getName() == "monitoring.PrimaryIH")
        {
            result = true;
        }
    }
    return result;
}
public boolean isPrimary()
{
    return isPrimary;
}

public final boolean isBack()
{
    return this.getNextTarget() == this.getRealTarget();
}

public boolean equals(MyInvocationHandler mih)
{
    return (this.getClass().getName() == mih.getClass().getName());
}
Appendix C

Code Listing 3: AbstractObject

abstract public class AbstractObject implements IObject
{
    static private ClassLoader cl = null;
    static private String directory = null;
    static private Class implClass;
    static private List instances = new ArrayList();

    public static IObject newInstance() throws InstantiationException,
                                           IllegalAccessException
    {
        AbstractObject obj = (AbstractObject)implClass.newInstance();
        IObject anAProxy = (IObject)ObjectIH.newInstance( obj );
        instances.add( new WeakReference( anAProxy ) );
        return anAProxy;
    }

    public static void reload( String dir ) throws ClassNotFoundException,
                                                InstantiationException,
                                                IllegalAccessException,
                                                NoSuchMethodException,
                                                InvocationTargetException
    {
        cl = new SimpleClassLoader( dir );
        implClass = cl.loadClass( "ObjectImpl" );
        if (directory == null)
        {
            directory = dir;
            return;
        }
        directory = dir;
        List newInstances = new ArrayList();
        Method evolve = implClass.getDeclaredMethod("evolve", new Class[]{Object.class});
        for ( int i = 0; i < instances.size(); i++ )
        {
            Proxy x = (Proxy)((WeakReference)instances.get(i)).get();
            if ( x != null )
            {
                ObjectIH aih = (ObjectIH)Proxy.getInvocationHandler(x);
                IObject oldObject = aih.getTarget();
                IObject replacmnt = (IObject)evolve.invoke(null,new Object[]{oldObject});
                aih.setTarget( replacmnt );
                newInstances.add( new WeakReference( x ) );
            }
        }
        instances = newInstances;
    }
}
import java.lang.reflect.*;

class ObjectIH implements InvocationHandler
{
    private IObject target = null;
    static private Class[] objectAInterfaces = { IObject.class };

    public static IObject newInstance( AbstractObject obj )
    {
        return (IObject) Proxy.newProxyInstance( obj.getClass().getClassLoader(),
        objectAInterfaces,
        new ObjectIH( obj ) );
    }

    private ObjectIH( AbstractObject obj )
    {
        target = obj;
    }

    public void setTarget( IObject x )
    {
        target = x;
    }

    public IObject getTarget( )
    {
        return target;
    }

    public Object invoke( Object t, Method m, Object[] args ) throws Throwable
    {
        Object result = null;
        try
        {
            result = m.invoke( target, args );
        }
        catch (InvocationTargetException e)
        {
            throw e.getTargetException();
        }
        return result;
    }
}
Code Listing 5: Controller

```java
package control;
import java.util.*;
import java.lang.reflect.*;
import logging.Event;
import logging.Repository;
import properties.*;
import monitoring.*;
import logging.*;
import monitoring.*;

public class Controller implements MyEventListener
{
    private static Controller instance;
    private ArrayList primaryProxies;

    private MyProps props;
    private String propFileName;

    private Repository rep;

    private MONITOR_LEVEL mon_level = MONITOR_LEVEL.LOW;
    private int MODERATE;// = 2000;
    private int MILD;// = 4000;

    private Controller()
    {
        rep = Repository.getInstance();
        rep.addMyEventListener
        {
            new MyEventListener()
            {
                public void myEventOccurred(MyEvent evt)
                {
                    analyzeExceptions(evt);
                }
            }
        };

        String sep = System.getProperties().getProperty("file.separator");
        propFileName = sep + "properties" + sep + "config.xml";
        setPropFileName(propFileName);
        primaryProxies = new ArrayList();

        MODERATE = Integer.parseInt(props.getProperty("moderate"));
        MILD = Integer.parseInt(props.getProperty("mild"));
    }

    public static synchronized Controller getInstance()
    {
        if (instance == null)
        {
            instance = new Controller();
        }

        return instance;
    }
}
```
public void analyzeExceptions(MyEvent evt)
{
    System.out.println("***************    ANALYZING EVENTS    ***************");
    System.out.println("Event generated by:"+evt.getSource().getClass().getName());

    ArrayList events = rep.getRepository();
    long t1 = 0;
    long t2 = 0;
    try
    {
        Event e1 = (Event) events.get(events.size()-1);
        t1 = e1.getTime();
        Event e2 = (Event) events.get(events.size()-2);
        t2 = e2.getTime();
    } catch(Exception e)
    {
        //not enough events to analyze...
        return;
    }

    //the time between successive exceptions
    long t3 = t1-t2;
    if ( t3 > 0 && t3 <= MODERATE )
    {
        //we monitor most aggressively.
        this.setMonLevel(MONITOR_LEVEL.HIGH);
    }
    else if ( t3 > MODERATE && t3 < MILD )
    {
        //we monitor with moderate agression.
        this.setMonLevel(MONITOR_LEVEL.MED);
    }
    else if ( t3 > MODERATE && t3 < MILD )
    {
        //we monitor least aggressively.
        this.setMonLevel(MONITOR_LEVEL.LOW);
    }
}

private void setMonLevel(MONITOR_LEVEL mon_level)
{
    this.mon_level = mon_level;

    for(int i=0; i<primaryProxies.size(); i++)
    {
        Proxy prxy = (Proxy) primaryProxies.get(i);  //Will be the primary proxy.
        MyInvocationHandler pih = (MyInvocationHandler) prxy.getInvocationHandler(prxy); //IH of the Primary
        Proxy nextPrxy = (Proxy) pih.getNextTarget(); //The next Proxy in the chain
        MyInvocationHandler nih; //The next IH in the chain
        boolean proxyTarget = true;
        while(proxyTarget)
        {
            try
            {
                nih = (MyInvocationHandler) nextPrxy.getInvocationHandler(nextPrxy); //The next IH in the chain
                nih.setMonLevel(mon_level);
                nextPrxy = (Proxy) nih.getNextTarget();
            } catch(Exception ex)
            {
                proxyTarget = false;
            }
        }
    }
}
public void registerPrimaryProxy(Object obj)
{
    if(!primaryProxies.contains(obj))
    {
        primaryProxies.add(obj);
        Proxy prxy = (Proxy)obj;
        MyInvocationHandler mih = (MyInvocationHandler)
            prxy.getInvocationHandler(obj);
        mih.addMyEventListener
        {
            new MyEventListener()
            {
                public void myEventOccurred(MyEvent evt)
                {
                    analyzeExceptions(evt);
                }
            };
        }
    }
}

public void unRegisterProxy(Object obj)
{
    primaryProxies.remove(obj);
}

public void setPropFileName(String fileName)
{
    propFileName = fileName;
    props = new MyProps(propFileName);
}

public MyProps getProps()
{
    return this.props;
}

public void myEventOccurred(MyEvent evt)
{
    this.analyzeExceptions(evt);
}

Appendix F

Code Listing 5: config.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="key">value</entry>
    <entry key="db.class">com.app.db.StubCustomerDB</entry>
    <entry key="moderate">2000</entry>
    <entry key="mild">4000</entry>
    <entry key="monitoring">true</entry>
    <entry key="handlers">monitoring.ProfilingIH;monitoring.TracingIH;
    monitoring.ExceptionMonitoringIH</entry>
</properties>
```