Adding Autonomic Functionality to Object-Oriented Applications

Marc Schanne
Software Engineering
FZI Forschungszentrum Informatik
schanne@fzi.de

Tom Gelhausen
Software Engineering
FZI Forschungszentrum Informatik
gelhauto@fzi.de

Walter F. Tichy
Department of Computer Science
University of Karlsruhe
tichy@ira.uka.de

Abstract

Integrating applications with autonomic functions such as checkpointing/restart, self-healing, or self-updating is difficult and time consuming [8]. We demonstrate that autonomic functionality can be separated from applications and supplied by default implementations, thereby dramatically reducing the cost of supplying autonomy. This article proposes a proxy/wrapper technique with an additional code hook-up infrastructure to provide application adaptation with self-updating, self-configuration and self-optimization functionalities.

1. Introduction

We propose a method that enables the addition of autonomic functionality [9, 10] to object-oriented applications in compiled form with the least possible effort. Our proposal can provide the following autonomic functionality to any component of the application:

- self-updating - flexible, dynamic updating, and concurrent use of *compatible versions* of a component;
- self-recovery - support for Recovery Oriented Computing (ROC) [7] through automatic checkpointing;
- self-healing - recursive restartability [3] to cope smartly with component failures.

Dynamic updating is needed, for example when applying translets (compiled XSL transformations) [1] in a web server based XML and XSLT environment:

The content data is stored in XML files and multiple views onto the data are created via a single servlet, applying the proper translets. Changes to the content data are handled through a smart caching architecture, but an implementation change a translet requires restarting the virtual machine, thereby losing all caches and sessions. We ensure high availability of the service by dynamic replacement of a translet without restarting the web server. If the new version of the translet fails, we want the system to self-heal from this situation by searching for and using an alternate version of the translet, for instance the last known working version as a last resort.

We present a system based on a proxy/wrapper architecture that addresses these problems.

2. Proxy/Wrapper Architecture

This approach is based on class renaming and proxy/wrapper generation. It provides the targeted autonomic functionality by adding application independent supplements. They are integrated through hook-up code in the wrapper method.

The design ensures there is no need for the user to adapt his source code in any way. It requires the following prerequisites regarding the programming environment/runtime system:

- dynamic polymorphism prerequisites,
- extendable class loading infrastructure or a separate bytecode transformation step,
- reflection,
- exception handling supporting undeclared runtime exceptions.

The approach is applicable on any runtime system that fulfills the above requirements. For the remainder of this paper

Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA'03)
1529-4188/03 $17.00 © 2003 IEEE

the Java 2 platform is used as an example system meeting these prerequisites.

Some meta information should be provided to the extendable class loading infrastructure respectively the separate bytecode transformation step:

- version information for the management of different class implementations;
- member attributes irrelevant for the object state (comparable to Java `transient` attributes), reducing the amount of data to be made persistent in every checkpoint (see 'Checkpointing' in section 2.2);
- assertions (pre- and postconditions, invariants) to support the self-healing mechanism;
- further optional data, declaring properties described in section 2.5.

2.1. Application Updating

In a manner comparably to [13], the versioning problem is solved through integration in the original inheritance tree. The type safe replacement is done by class loading with transformation and separation of functional components from data storage.

The Updating Infrastructure provided by our approach has to support the following main goals:

- type-safety, proofed by a standard type-safe compiler and supported with a standard virtual machine;
- provision of arbitrary applications with the demanded functionality;
- no requirement for any source code changes;
- need for only moderate changes to the bytecode;
- application independent default implementation for version management.

It is impossible to know all references to a certain object without interfering with the runtime system. Thus, it is also impossible to update all of these references to point to a new instance of a different version of the class, if needed. Therefore, we require all references pointing to a proxy object to encapsulate a reference to the potentially changing implementation. Since we generate these proxy objects ourselves, we can register all instances created. This way, we can find all references to any instance of the original component and enable its replacement. The original structure as shown in figure 1 is transformed into the structure shown in figure 2.

Any Code that creates instances of `Clazz` via the standard `new` instruction directs the proxy class

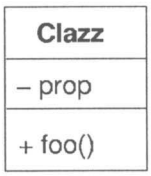

Figure 1. Original structure

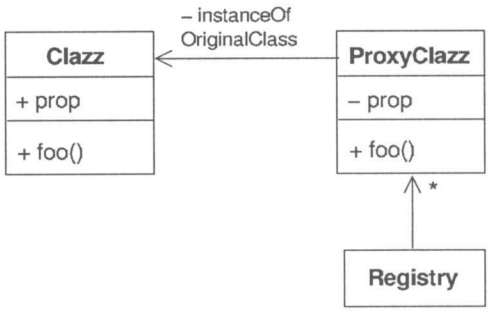

Figure 2. Transformed structure

(`ProxyClazz`) to assume the name of the original class which is renamed. Our prototype accomplishes this by using the Byte Code Engineering Library (BCEL) [5]. The original class is changed to allow the proxy class to access all member and class fields and methods. Of course, the proxy object has to be an exact blueprint of `Clazz`. The structure of `Clazz` is determined through reflection [12]. Method calls on the proxy object are simply delegated. With this changes the following code at some arbitrary position in another class will transparently use a proxy instance to access an instance of the original class (IOOC) instead of accessing it directly.

```
Clazz c = new Clazz();
c.foo();
```

There is still one problem left: synchronization. On the one hand, external code that accesses member attributes directly rather than using getter and setter methods causes the state of the proxy object to differ from the state of the IOOC. Of course, the external code expects the IOOC to be in the state of the proxy object in a subsequent method call rather than in the state in which it still resides. On the other hand, a method in the IOOC may change attributes and not inform the proxy about the state change. Consequently, subsequent direct reads on the proxy object's member attributes lead to wrong results.

One simple way of tackling the synchronization problem is to perform a full synchronization before and after every non-private method call. Thus, the implementation of an arbitrary method `foo()` in the proxy class looks like

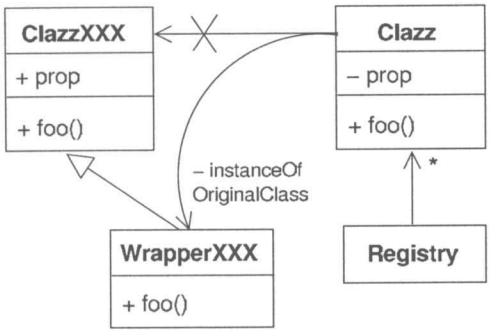

Figure 3. Additional wrapper

```
memberSyncProxyToOriginal();
instanceOfOriginalClass.foo();
memberSyncOriginalToProxy();
```

The proxy class is a blueprint of the original class with all non-private methods (delegating the execution) and with fields for all member and class attributes. The methods for synchronization are generated. It is possible to provide meta information of the attributes that partake in the object state to enable more efficient synchronization handling.

Further rules for a more complex inheritance structure of the original classes and constraints for the proxy generation are discussed in section 2.4.

2.2. Application extension

To support existing applications with the desired autonomy, program adaptation is necessary. This sections describes a solution to add application independent functionality to compiled classes.

Checkpointing The ability to restart quickly from a system crash requires that the internal state of the managed IOOCs is made persistent. The natural granularity for restarts in an object-oriented environment is defined through the methods' boundaries. Thus, it does not make sense to set more than one checkpoint before each method call.

Yet, if a method `m()` of object `o` calls method `m'()` of `o`, it is also unnecessary to set a checkpoint at the beginning of `m'()`: There is no way of (transparently) restarting `m()` at the position it called `m'()`. However, it is obvious that it does make sense to lower the checkpoint rate to those function calls where the caller is a different instance. These calls happen to be made through our proxy object, whereby the implementation of the arbitrary `foo()` method in the proxy class looks like

```
setCheckPoint();
memberSyncProxyToOriginal();
instanceOfOriginalClass.foo();
memberSyncOriginalToProxy();
```

Runtime Exceptions

> *Most software bugs in production quality software are Heisenbugs. [...] Even if the source of such bugs can be tracked down, it may be more cost effective to simply live with them, as long as they occur sufficiently infrequently [3]*

Besides such software faults, the Gartner Group estimates another 20% of unplanned downtime in business environments is due to hardware faults, of which 80% are intermittent [3, 4]. All of these errors are difficult to detect but even more difficult to fix. A rollback and restart algorithm to perform some of the demands of [3] is proposed as follows.

The calls to the original methods are encapsulated by a retry algorithm to give the system another chance to pass the call without throwing an unintentional exception. We identified Runtime Exceptions (i.e. NullPointerException and ArrayIndexOutOfBoundsException in Java) as the class of exceptions that are typically thrown unintentionally, thus indicating a Heisenbug. The retry algorithm provides fallbacks in the following order: (a) rollback to last checkpoint and retry with the existing instance; (b) create a new instance of the current version of the component, initialize with last checkpoint, and retry; (c) search for latent candidates (i.e. pending updates of the component, or last known working version), create an instance of the other version(s), initialize with last checkpoint, and retry; (d) initialize an internet wide lookup for different releases, load, create instance, initialize with last checkpoint, and retry; (e) self-update with an assertion enabled (see 'Assertions' below) debug version, and retry; (f) inform user/administrator via special exception about type of problem, tested versions, and violated assertions.

Assertions There are two kinds of assertions: the ones the original programmer chose to use in his code via the `assert` keyword and assertions defined via the meta information provided to the class loader. The pre- and post-conditions can be used for special implementation wrapper code [6] to obtain additional information on the source of an exception. The invariants may be checked before and after any method call. The information the system developer has to provide is comparable to the Java Modeling Language interfaces [11] and enables the definition of correct program traces [2] as well as their runtime verification through a watchdog component. The proxy's code for our arbitrary method `foo()` is as follows:

```
checkInvariants();
```

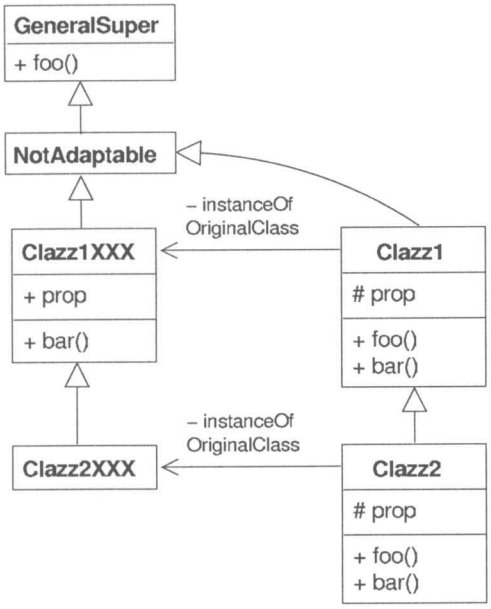

Figure 4. Proxy classes in a more elaborate inheritance structure

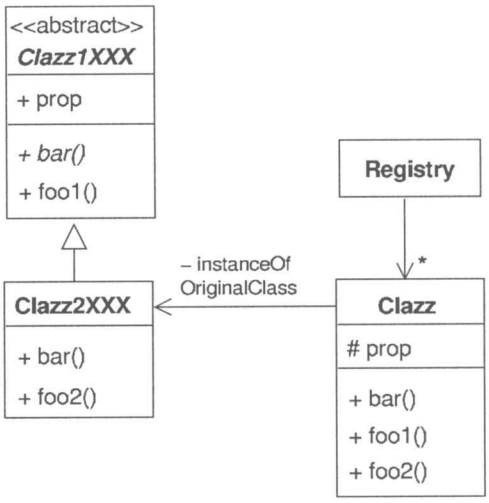

Figure 5. Transformed structure with abstract super class

```
checkPreCondition1(paramX, paramY);
checkPreCondition2();
setCheckPoint();
memberSyncProxyToOriginal();
[snip try] // retry algorithm here
instanceOfOriginalClass.foo();
[snip catch]
checkInvariants();
checkPostCondition3(paramX);
memberSyncOriginalToProxy();
```

2.3. Design Pattern Proposal

The preceding code snippet shows that the code within the proxy class grows with every new feature introduced. But even when declared for performance reasons, the use of assertions, for example, may not always be desired. Situation dependent dynamic selection as shown in paragraph 'Runtime Exceptions' seems appropriate. Consequently, these code changes cannot be done within the proxy, since we cannot alter its implementation. The wrapping code must be placed somewhere else.

We do not want to alter the code of the original class too drastically (i.e. merging all the instructions mentioned previously into the bytecode). Therefore, we need to generate an additional wrapper to add the demanded functionality. In order not to require an additional step of indirection, this proposal uses dynamic polymorphism and overrides the methods by inserting the hook-up code into the generated wrapper class (`WrapperXXX`). The class structure is depicted in figure 3.

2.4. Advanced inheritance constraints

The presented approach of adaptation is being developed to augment applications with autonomic functionality. The figure 4 shows all inheritance relationships in a larger inheritance tree. The proxy classes must be full blueprints of the given original classes (for simplification, only the transformed original classes are depicted) and they should be transformed without any semantic knowledge. Each class gets renamed, and if its super class is also transformed, the inheritance relationship is adapted.

The proxy class is generated by analyzing the transient closure of all super classes. If there is an abstract class as an ancestor for this class, no proxy or wrapper is generated (compare with figure 5).

The introduction of a new wrapper class allows for a further abstraction especially needed to deal with additional language features.

The keyword `final` in Java (depicted in figure 6) is used to prevent the specialization of the method `foo` in subclasses of `Clazz1`. To reproduce this behavior with the transformed class structure, additional wrappers are used to encapsulate the `final` semantics. There is no inheritance relationship between the wrappers because each class has to implement all the delegating methods specified by the transient closure of the original class with which the `final` modifier would conflict.

The Java language safety concept relies on a defined general super class (`java.lang.Object`). Generating all delegating methods of the transient closure, especially the final methods, restricts the proxy/wrapper architecture. Also, the class loading with the Sun Microsystems' JVM restricts the adaptation possibilities on library classes not located in `java.*`, `javax.*`, `org.omg.*`.

These restrictions limit the number of application classes in the inheritance tree (shown in figure 4) that could be transformed.

2.5. Further Options

So far we have shown how to integrate updating, checkpointing, a retry algorithm, and assertions into compiled classes. These are only the first steps towards enabling given applications with autonomic functionality. As examples for further options of well-known concepts that may be integrated, we discuss the fields of security, pluggability, and generic integration.

Authentication and Authorization Modern applications often contain security relevant code. This functionality is an ideal candidate for delegation to a default implementation.

Our proxy facilitates the weaving of security aspects into a given application. The functionality is added by code adaptation in the same way as already demonstrated for checkpointing and assertions.

An implementation to support user access control for any application method can be built, for example, on top of the Java Authentication and Authorization Service (JAAS) [12]. This way, existing Pluggable Authentication Modules (PAM) can be added as supplements to an existing application without changing the source code.

Pluggability Our proxy classes can be used to represent classes other then local classes. Our design allows IOOCs to be called remotely via techniques like RMI, CORBA or even SOAP. Consequently, the proxy does not even have to be a class, but could, for instance, also be a web service. Innumerable SOAP toolkits support converting classes into web services. They enable transparent replacement of components during runtime as well as providing functionality to the outer world. The proxy environment defines and controls interaction and communication between the components.

Generic Integration Given a compiled class of an existing application that is loaded through our proposed technique, its corresponding proxy can be used to enrich its interface automatically. Moreover, the proxy can be used to integrate IOOCs into a given class hierarchy. This way, the class – now represented by the proxy – may inherit

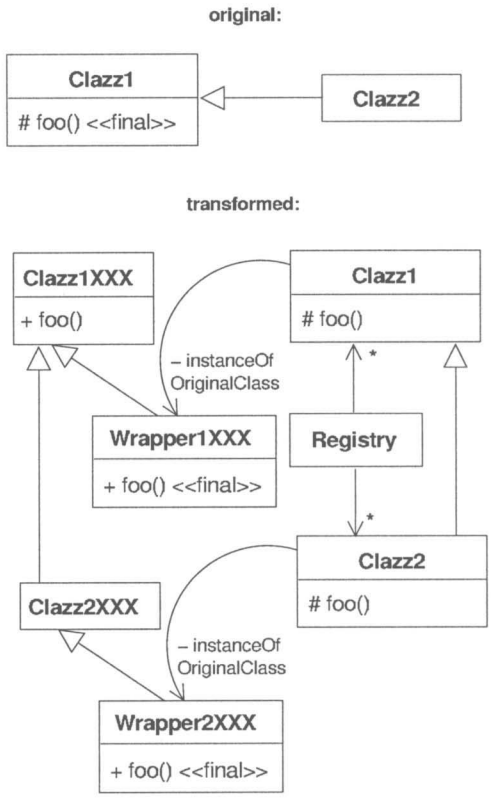

Figure 6. Special transformation to deal with the keyword `final`

all interfaces predefined in the hierarchy. Assume class `X` provides a `public void bar()` but does not (by original design) implement the interface `I`, which also declares a `public void bar()`, the proxy for `X` can automatically implement `I` when it is added to a running system that already knows `I`. This function can be regarded as a shortcut to accessing a component in the same container via SOAP/WSDL (see paragraph 'Pluggability') since the original vendor was not aware of (and thus was not able to implement) the interface another class requires. This is also a future backdoor to enable type save (semi-) automatically mapped functionality required and provided by components from different vendors using different ontologies.

3. Current Status and Future Work

So far we have successfully implemented application updating as well as application extensions, namely checkpointing, assertions, and runtime exception healing in our system. The developed prototype implements the introduced infrastructure with the class loader and a local lookup for new class versions and additional meta information. Our

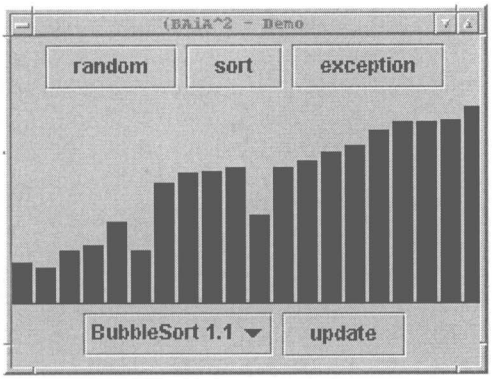

Figure 7. Sample Application

future intention is the integration of these concepts and additional further options to add autonomic functionality to object-oriented applications.

A sample application shown in figure 7 is used with the infrastructure and tests the updating and exception healing mechanisms. The GUI application follows the model-view-controller pattern. The GUI also shows the restrictions of our approach with Sun Microsystems' Java VM. Only the non-graphical parts (model and algorithms) are useful for dynamic updating without any further indirections.

Perhaps the modification of the Java VM and the implementation of the native class loading could be a possible enhancement of our platform- and VM-neutral approach.

Our first goal is to determine the autonomic properties we would like to demand of an advanced runtime architecture. We deem performance considerations as secondary, since we do not want to solve problems that might be solved automatically according to Moore's law within the next few years.

4. Conclusions

We presented a software technical solution to add autonomic functionality to existing applications that does not depend on any source code adaptation. Our prototype is currently implemented based on the Java class loader architecture since this approach allows powerful just-in-time bytecode transformation in conjunction with a bytecode engineering toolkit like BCEL [5]. Yet the solution is not limited to the Java world in any way. By providing a separate bytecode transformation step, it is possible to attribute arbitrary classes automatically.

Currently, we are working on the types of features that can be added to an arbitrary application through our proposed technique. We envision a kind of autonomic functionality framework that enables fine-grained management of autonomy added to arbitrary applications.

References

[1] Scott Boag. *Xalan-J 2.0 Design*. The Apache Software Foundation, in progress edition, 2000. http://xml.apache.org/xalan-j/.

[2] Mark Brörkens and Michael Möller. *Trends in Testing Communicating Systems*, chapter JASSDA TRACE ASSERTIONS - Runtime Checking the Dynamic of Java Programs, pages 39–48. March 2002.

[3] George Candea and Armando Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[4] Timothy C. K. Chou. Beyond fault tolerance. *IEEE Computer*, pages 31–36, April 1997.

[5] Markus Dahm. Byte Code Engineering with the BCEL API. Technical Report B-17-98, University Berlin, April 2001.

[6] Michel de Champlain. The Contract Pattern. In *Proceedings of Pattern Languages of Program Design 4 (PLoPD4)*, October 1998.

[7] Armando Fox. Toward Recovery-Oriented Computing. In *Proceedings of the 28th VLDB Conference, Hong Kong, China*, 2002.

[8] W. Wayt Gibbs. Autonomic Computing. *Scientific American.com*, May 2002. http://www.sciam.com/.

[9] Paul Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. IBM, October 2001. http://www.research.ibm.com/autonomic/.

[10] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, pages 41–50, January 2003.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, and Clyde Ruby. *JML Reference Manual*, draft, 1.17 edition, October 2002.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification: Chapter 4. The class File Format*. Sun Microsystems, 2nd edition, 1999.

[13] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A Technique for Dynamic Updating of Java Software. In *Proceedings of the IEEE International Conference on Software Maintenance (CSM 2002)*, October 2002.