Implementing On-line Software Upgrades in Java

Martha Gebremichael

Computer Science Department
University of Maryland, College Park

Advisor:

Dr. Michael Hicks
Assistant Professor, Computer Science Department
University of Maryland, College Park

May 2003

Content

Implementing On-line Software Upgrades in Java

1. Abstract

There are many systems that have to be available without interruption and so cannot be brought down for modifications. Examples of such applications include financial transaction processors, telephone switches and air traffic control systems. On the other hand, hardly any software is deployed error-free or fully functional, thus requiring fixes and changes. Therefore, there is a need for runtime software upgrading capability for such systems. This paper discusses existing approaches of "hotswapping" – runtime updating of classes – for the Java language. The different implementations of hotswapping are compared with respect to functionality, efficiency, ease of use and limitations. In addition, an improvement for one hotswapping mechanism is suggested and implemented.

2. Introduction

Many applications have to run continuously and cannot afford to be down for improvement and maintenance. A common solution is to design the systems in a special way and/or run programs on a specially configured, redundant hardware. In this project, an alternative method is investigated where parts of the application is updated at runtime. The term "hotswapping" refers to the ability to change classes while the program is running.

One problem when updating a class dynamically is what to do with existing objects. There are four alternatives [4, 5]:

1. Barrier method - put a "barrier", blocking object creation until all existing objects of older versions have expired. Then the new version takes over and object creation resumes.

2. Recreate all existing objects using the new version - since different class versions can have different internal data structures, copying each object's state requires an understanding of the object's semantics.
3. Don't take any action at all - all new objects are created with the new version, and existing objects continue with their current versions.
4. Active partitioning – allows the user to actively select which objects to update and which to leave at the previous version.

There are different approaches to do hotswapping in the Java language. Two of the main approaches being using a library based approach to do dynamic updating and modifying the Java Virtual machine to accommodate dynamic classes. In this section, previous work on hotswapping will be introduced.

2.1 Implementation by Sun Microsystems Laboratories

The HotSwap project by Sun Laboratories [1] provides a GUI client tool that provides access to the HotSwap functionality available in the Java HotSpot™ Virtual Machine starting from JDK 1.4. The tool provides the capability to dynamically update classes at the JVM level by attaching to the running JVM, make some simple inquiries to it, and perform dynamic class redefinition on-the-fly. This is achieved by invoking a special API call that takes an existing class object and the new class version in the form of a byte array corresponding to the .class file.

The Hotswap tool uses the same standard wire protocol that debuggers use to communicate with JVMs. The Java Debugging Wire Protocol (JDWP) defines the format of information and requests transferred between the debuggee VM and the debugger front-end. Therefore, to be able to use the hotswap tool, the target JVM should be started with several options that would enable it to listen to a specific port, connect to the tool, and exchange information with it over a socket connection. These options tell the JVM to activate the internal data structures supporting debugger protocol communication, and start listening to the port in parallel with the application execution.

The core of the runtime class evolution functionality is implemented as two calls, RedefineClasses and PopFrame in the Java VM Debugging Interface (JVMDI) whose

prototypes are given below (Fig.1). The JDWP was extended to include the commands corresponding to RedefineClasses and PopFrame.

```
Typedef struct {
        jclass clazz;                   /* Class object for this class*/
        jint class_byte_count;          /* number of bytes defining class(below) */
        jbyte *class_bytes;             /* bytes defining class(in Class File Format of JVM spec) */
} JVMDI_class_definition;

jvmdiError RedefineClasses( jint classCount, JVMDI_class_definition *classDefs);
jvmdiError PopFrame(jthread thread);
```

Fig. 1 Specification of JVM Calls supporting Runtime Class Evolution

All classes given are redefined according to the definitions supplied in the RedefineClasses function. If any redefined methods have active stack frames, those active frames continue to run the bytecodes of the original method while the redefined methods will be used on new invocations(similar to method 3 above). PopFrame pops the topmost stack frame of thread's stack. Popping a frame takes you to the preceding non-native frame. When a thread is resumed, the thread state is reset to the state immediately before the called method was invoked.

2.2 Inxar Hotswapping Architecture

The Inxar (Information Exchange Architecture) hotswapping architecture [3] provides a proxy-based library to do dynamic class updating. Hotswapping is achieved through recompilation, dynamic class reloading, and object state migration throughout the life of an application.

The hotswap operation involves a single ProxyClass PC and a set of Proxy instances P that are the children of PC and willfully elect to be a participant of the transaction. The hotswapping, triggered by invocation of either the Proxy.hotswap() or ProxyClass.hotswap(), is a two-phase commit protocol. The ProxyClass PC manages a Class c; each Proxy p in P manages a single proxy Object po. When an object o is hotswapped, the class c is recompiled and reloaded into memory. If the reload step is

successful, the ProxyClass instance mediating the reload will free the old Class c and replace it with the new Class c'. Once c' has been reloaded, a new object o' is constructed by reflection from c' under the control of the delegate Proxy. Section 3 shows details of this mechanism.

The class to be hotswapped is never referenced directly by the application program. Otherwise, it will not be possible to load the class using the classloader provided in the library. Instead, the dynamic class should implement an interface which is referenced in the application program.

For each `Class` that is being monitored for HotSwapping, three timestamps are watched to decide what to do: the lastModified timestamp of the sourcefile (`.java'), the lastModified timestamp of the classfile (`.class'), and a timestamp that is recorded when a class is loaded into memory. If the source file is newer than the class file, it is recompiled and reloaded.

During the hotswapping process, handling of existing objects of the dynamic class is done by method 2 described above, i.e. all existing objects are recreated using the new version of the dynamic class upon successful compilation and reloading of the class.

The Inxar project provides two implementations:

Implementation I – Hotswapping is done only when the hotswap() method is explicitly called by the application program. This can be done with JDK 1.2 or higher.

Implementation II – This approach uses *dynamic proxy classes* (`java.lang.reflect.Proxy` and related interfaces provided by JDK1.3 or higher). Rather than hotswapping only when the `Proxy.hotswap()` is called, the proxy will attempt hotswapping at every method invocation on the dynamic proxy object. The invoke() method of the InvocationHandler is called which in turn calls hotswap() and then the method of the dynamic object.

2.3 Dynamic Java Classes

The Dynamic Java Classes implementation[4] achieves dynamic evolution of programs by extending the Java class loader and modifying the Sun's Java virtual machine (JDK 1.2) to provide runtime system support for dynamic classes.

The new dynamic class loader, which extends the JVM class loader, supports replacement of a class definition and update of objects. It defines two new methods: reloadClass and replaceClass. Method reloadClass is similar to loadClass in that it reads a designated class file from the disk, creates a class object, and returns it. However, loadClass does not load classes that are already defined in the system, whereas reloadClass succeeds whether the target class was previously defined or not. Method replaceClass takes the new class and initiates instance update.

To support the library functions, minor changes in some data structures and functions internal to the JVM were made. These include:

- The Just-in-Time(JIT) compiler which generates native machine code from Java bytecode on the fly, is disabled in this implementation. If it is not disabled, the modified JVM will have to ensure that previously generated machine code does not become invalid when methods change, i.e. it must ensure that any modified methods are recompiled. Since this is not implemented, the JIT is disabled.
- Since inlined code may be invalidated by a class change, method inlining for all classes loaded by a dynamic class loader is disabled.
- Limited use of quick instruction - When a class is first loaded, its constant pool contains symbolic references, and its byte code contains indices into the constant pool for all method and data access instructions. The first time the JVM encounters any such instruction, it checks if the constant pool entry has been resolved, and resolves the entry if needed. Then, the JVM changes the instruction to a special quick instruction that does not perform the check, hence any subsequent execution of that instruction is relatively fast. In this implementation, the modified JVM only uses quick instructions that do not contain any offsets or direct references. This avoids the need to update bytecode, but has slight performance penalty.

The instance update model used in this case is global updating (method 2 above), i.e. all instances of the dynamic class must be located and updated to reflect the new class definition. To update the objects, a method similar to the mark-and-sweep technique of garbage collection is used.

2.4 Dynamic C++ Classes

The dynamic C++ classes implementation[5] is similar to the Inxar implementation where a proxy library is used to achieve hotswapping. However, it is different in two ways:

- Instances are updated differently. In the Inxar implementation, global updating (method 2) is used where all instances are updated when a class changes. In this implementation, passive partitioning(method 3) is used - objects created before the updating are unchanged, and any objects created afterwards reflect the new type and,
- Compilation is not performed in the library

To achieve hotswapping, each dynamic class has to be written as two separate parts: an abstract interface class that is known to the program at compile time and one or more implemenation classes that inherit from the interface class. This is analogous to Inxar's use of Java interfaces.

The core of this implementation is a generic template class. The template serves as a proxy for each dynamic class. As with any proxy, a program creates a dynamic class instance by creating a proxy instance instead. To use a dynamic class, the template is instantiated on the interface class. At run-time, the template locates the shared library that contains the most recent implementation class and loads this library into the program's address space. The template calls into the library to create an instance of the implementation class, and casts the instance to the type of the interface class. Finally, the public interface operations are accessed through the template using standard pointer redirection.

Three methods are provided to manipulate the current version: activate, invalidate and activate-and-invalidate. The activate method registers a new version as the current version. Objects of older versions remain in existence. Once all objects have expired normally, the older versions are removed from the system. The activate-and-invalidate

method registers a new version as the current version but also invalidates (destroys) all objects of older versions. The invalidate method invalidates all objects of a given version.

The example provided by this implementation[5], also shown below in Fig. 2, illustrates how hotswapping is achieved. The interface *Receiver* has two implementations: *ReceiverImp* and *ReceiverDebuggingImp* where each class is compiled into its own shared library, *imp.so* and *debugimp.so*. dynamic is the name of the proxy template provided by this implementation.

```
// normal program operation – create and use
// normal packet receivers
dynamic<Receiver>::activate ("imp.so");
dynamic<Receiver> receiver;
Packet packet = receiver.receivePacket();
…
// switch to debugging mode in response to
// some external event (library name would
// be included in the external event)
dynamic<Receiver>::activate ("debugimp.so");
// now all new packet receivers will contain
// the debugging code
dynamic<Receiver> otherReceiver;
…
```

Fig 2. Hotswapping for C++ classes

3. Discussion

3.1 Comparison of Hotswapping Implementations

The different hotswapping mechanisms discussed above have their own advantages and limitations. As to which approach is best for a system depends on the needs of the application program. In this section, the implementations are compared with respect to their functionalities, efficiency and limitations.

If the application program has a fixed interface, i.e. no methods/fields are expected to be added or removed, then the current Sun implementation would be the best approach to use. This implementation is advantageous in that no modification is necessary to the application program to make a class dynamic. In addition, the programmer does not have to decide in advance which classes should be dynamic, any class can be dynamically updated without special modifications. However, the changes that can be done to the

dynamic class are limited. At the current stage, only method bodies can be updated, but in subsequent stages, the hotswapping tool will be made to handle more complicated class updates[2].

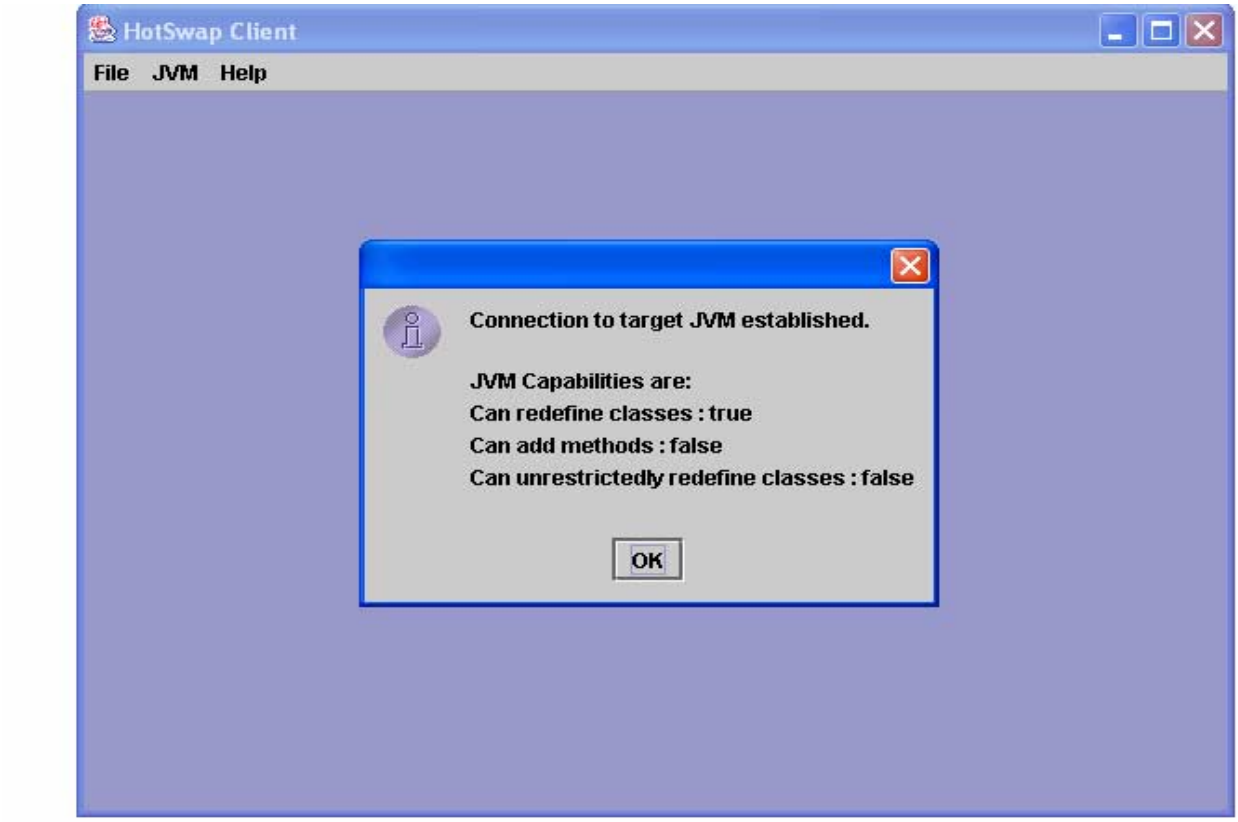

Fig. 3 The Sun client tool - after making a connection to the running JVM

Another advantage of this implementation is that it has an easy-to-use GUI client tool (Fig. 3). Using the provided menu items, the user sets the path to the old and new classes and also the port to connect to(which should be the same port used when starting the JVM). After the connection is made, it is possible to inquire what classes are loaded into the JVM, what classes have changed and finally to submit the changes to the JVM (Fig. 4)

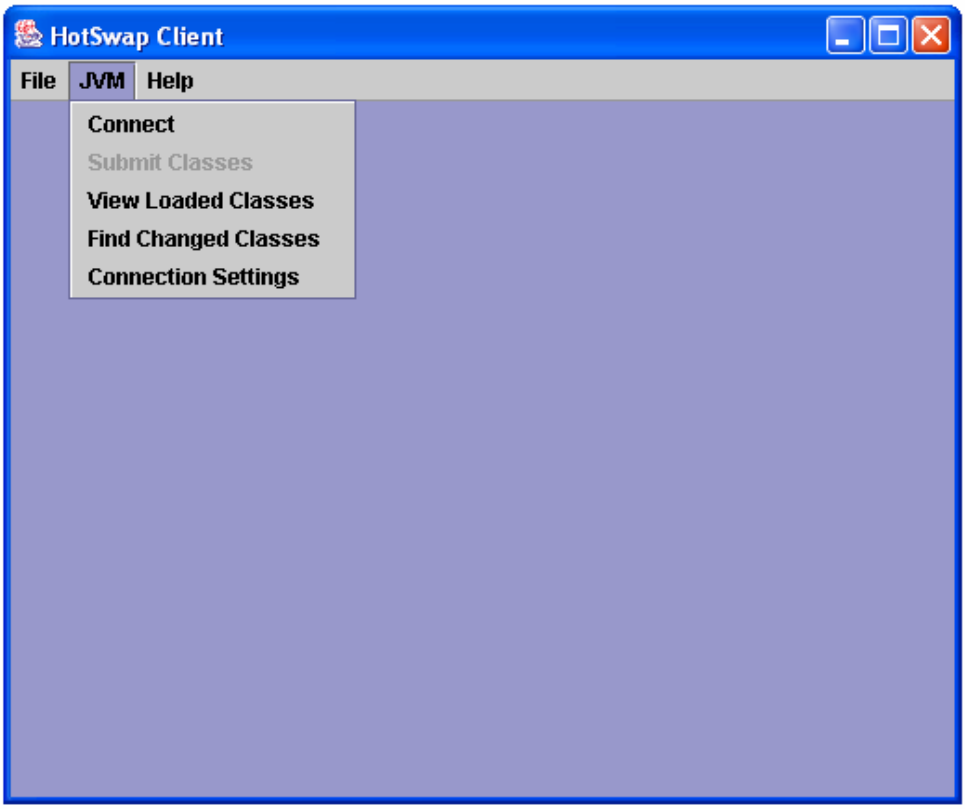

Fig. 4. Available menus after the connection is made

If the application program expects to make method/field additions, then it is better to use a hotswapping library(an interface to communicate with the JVM) and/or modify the JVM to accommodate the hotswapping. Unlike the Sun implementation, these approaches do not support unanticipated class evolution, i.e., the application must decide beforehand which classes should be made dynamic. In addition, the application program should have a means of interacting with the user/administrator, retrieving the new class version and installing it, and this makes the program less readable or harder to maintain.

Comparing the implementations discussed in Section 2, using the proxy library of Inxar is advantageous in that the application program is not bound to a specific JVM. Efficiency, however, is a problem. Every time the hotswap() method is called, even if the class is current and does not need to be compiled and reloaded, checking as much is still very expensive. In addition, the application program has to be designed in a certain way (no reference should be made to the dynamic class – it should always implement an

interface). Handling race conditions and deadlocks is also a problem in multithreaded programs.

Although the performance penalty is not that high (6% in the case of UCDavis implementation[4]), modifying the JVM to support dynamic classes has the disadvantage of requiring use of a specific VM. For example, the UCDavis implementation is done using JDK1.2 and the Java functionalities provided in JDK1.3 and later could not be used. This could be very limiting to the application program. For the programmer who develops and maintains the hotswapping implementation, modifying the JVM is much harder as it requires deeper understanding of the details of the JVM.

Another consideration when developing the hotswapping library is the model to use for instance updating. The barrier method blocks updating until all old objects have expired, and this essentially prevents updating of active classes. This could also be costly to implement as it may require reference counting. The passive partitioning model where only objects created after the class update reflect the new type, allow multiple definitions of a class to be active simultaneously. This has the disadvantage of breaking the Java name-binding semantics. Recreating all existing objects using the new class version is the most efficient but is harder to implement since different class versions can have different internal data structures. Copying each object's state requires an understanding of the object's semantics.

3.2 Detail Discussion of Inxar Implementation

Before comparing the two implementations provided by the Inxar project, let us first see the details of each implementation. For the first implementation, the hotswapping is done by including the following lines of code in the application program:

```
ProxyClassLoader loader = new ProxyClassLoader();
ProxyClass cls = loader.loadJDK12( "MyDynamic" );
Proxy proxy = cls.newInstance();
MyInterface myDynamic = (MyInterface) proxy.hotswap();
myDynamic.execute();
```

The first three statements create a ProxyClassLoader for the application, associate a ProxyClass (PC) with the dynamic class, and create a new hotswap Proxy - a holder for

the dynamic class. The ProxyClassLoader holds a set of Proxy Classes - one PC for each class, and the ProxyClass holds a set of proxies - one Proxy for each object. Fig. 5 shows the control-flow when the hotswap method is called in the first implementation of Inxar. The Proxy class returns a new instance if it is being called for the first time or if the class has changed, otherwise, it returns the old object that it caches. If the class has changed, when the execute() method is called it is executed with the new instance and shows all the changes made in the new class.

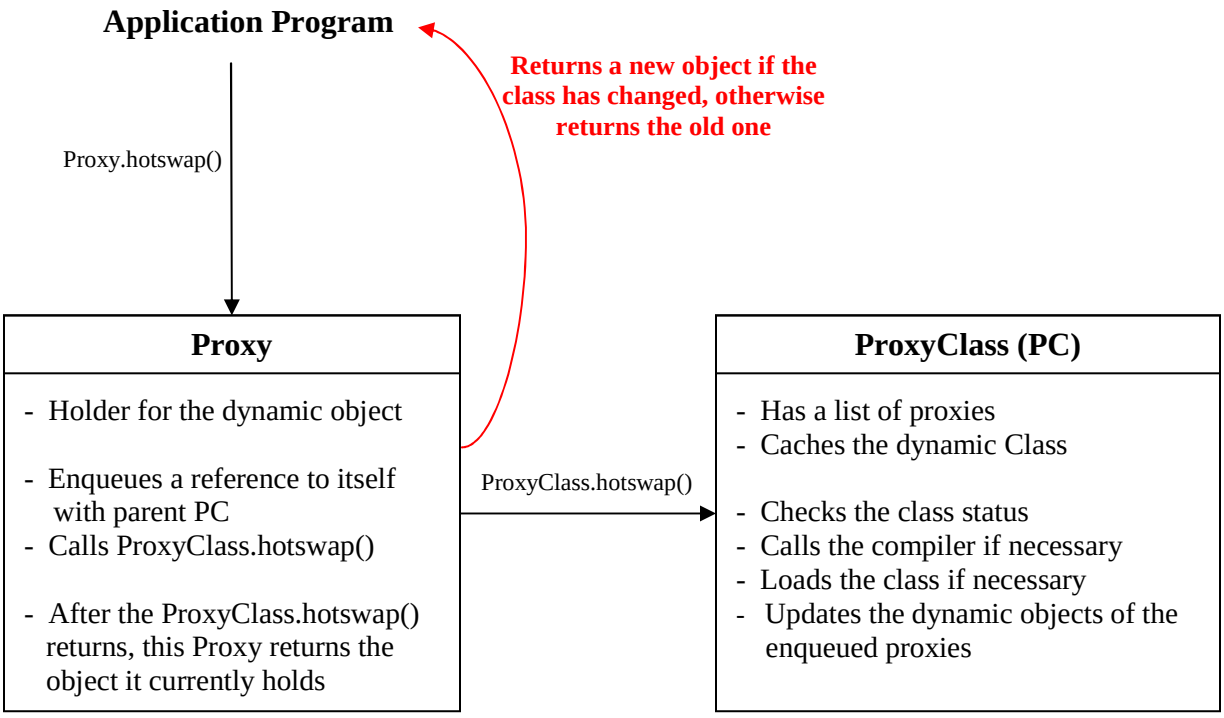

Fig. 5 Control-Flow visualization of hotswapping using Inxar Implementation I

In the case of Implementation II, the code used is slightly different.

```
ProxyClassLoader loader = new ProxyClassLoader();
ProxyClass cls = loader.load( "MyDynamic" );     // returns JDK13ProxyClass
Proxy proxy = cls.newInstance();                 // Java Proxy
MyInterface mydynamic = (MyInterface) proxy;
mydynamic.execute();
```

In this case when the statement,

```
cls.newInstance();
```

is executed, it creates a java.lang.reflect.Proxy instance and associates an Invocation Handler with it. The created Proxy implements MyInterface and the hotswap Proxy. That is why a statement like

```
MyInterface mydynamic = (MyInterface) proxy;
```

is possible. So whenever a method on an instance of MyInterface is invoked, the dynamic proxy dispatches it to the specified invocation handler (Fig. 6). It is in the invoke method of the Invocation handler that the Proxy.hotswap() method is called.

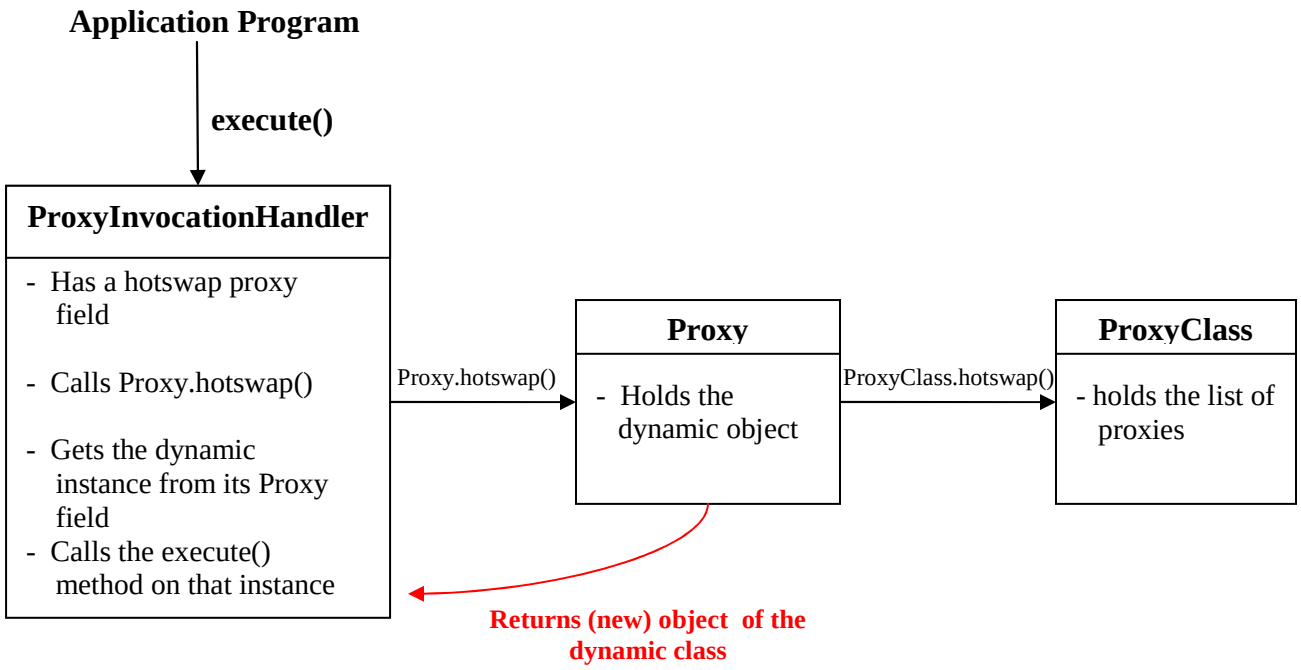

Fig. 6 Control-Flow visualization of hotswapping using Inxar Implementation II

The two implementations have their own advantages and disadvantages. In the first implementation, the hotswap() method has to be called explicitly every time the application wants to use the new instance. This makes the program less readable and could be hard to keep track of when to call hotswap(). In the second implementation the call to the hotswap() method is transparent to the application program. However, this will result in a lower efficiency because the hotswap method is called with every method call of the dynamic object. In addition, dynamic proxy class instances require more processing per invocation than non-proxy class instances, further lowering the efficiency.

If this overhead of the Java proxy class could be avoided by using an efficient proxy, it would be the best approach as it has the advantage of making the hotswap calls transparent in the application program thus making the program less prone to errors. However, with the current inefficiency of the proxy, if there are several instances of the dynamic class and/or if the methods are frequently called, using Implementation I is better. The comparison in efficiency between the different implementations is shown in Section 5 of this paper.

4. Experiments

The experiments were done to compare the runtime efficiency of various hotswapping approaches. The Inxar hotswapping architecture is modified and the experiments show the increase in efficiency due to this modification. In addition, the cost of using proxy class provided in java.lang.reflect package is evaluated. Details of the results are shown and discussed in Section 5.

4.1 Experimental Setup

The tests were done using Java 2 Platform, Standard Edition v. 1.4 on a system with the following properties:

- Microsoft Windows XP operating system
- 1.8 GHz Intel Pentium 4 Processor
- 512 MB of RAM

4.2 Modifications to the Inxar Implementation

In the Inxar implementation, the compiling of the dynamic class is embedded in the hotswapping library and the program has to constantly check the source and class files to see if the class needs compiling and re-loading. This is very inefficient for the application program. In this section, an improvement to this approach is suggested and implemented. The application program creates a thread that listens for hotswapping signals on a particular port and updates all instances of the class upon receiving such a signal. This effectively implements interrupt-driven approach as opposed to the polling method used in the original Inxar implementation.

The advantages of these modifications are as follows:

- The program does not have to monitor the dynamic class files to decide when to compile and reload the class. This increases the performance of the application program significantly (see Section 5). Creating and running the update thread does not create new performance issues because the thread blocks until a connection is made.
- The hotswapping code becomes better organized and easier to maintain.
- Better suited for controlling multi-threaded applications. The updating thread upon receiving a signal can wait until all the other threads reach a certain "clean" state and then block them to do the updating. This will help avoid deadlock as it currently happens in the Inxar implementation.

4.3 Efficiency Comparison

The classes used for the comparison are shown in Fig. 7. MyDynamic class is the class to be hotswapped and implements MyInterface. MyMain class is what is varying in each of the cases considered below. The MyInterface in the main class refers to a MyDynamic class created only through a proxy.

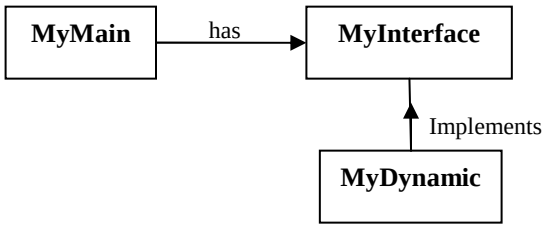

Fig. 7. Classes used in the experiment

Six cases were considered (see source codes in the Appendix)

Basic: Basic case where there is no hotswapping (MyMainBasic.java)

Inxar_I: Inxar Implementation I - the basic hotswapping using a proxy around the dynamic class(MyMainI.java)

Inxar_II: Inxar Implementation II - Using the Proxy in java.lang.reflect package in addition to the hotswapping proxy. This makes the call to hotswap

transparent in the application program (MyMainII.java)

Inxar_Hotswap: Inxar Implementation I - but calling the hotswap() before each execute() (MyMainHotswaps.java)

Modified_I: Modified Inxar Implemenatation I - Using a hotswap proxy and an updating thread that listens for an updating signal (MyMainIModified.java and UpdateThread.java)

Modified_II: Modified Inxar Implementation II - Using the hotswap Proxy and java Proxy and an updating thread to do the hotswapping upon receiving a signal. (MyMainIIModified.java)

4.4 Running Programs

To run the programs in this project (for example MyMainIModified class):

Compile the main class and the classes in the library
Executing: java -Dorg.inxar.hotswap.properties=examples.properties MyMainIModified

In case of the modified library, updating is done as follows:
http://hostName:portNum/className

The attributes in the examples.properties file have to be modified with the appropriate names and paths. The hostName is 'localhost' if the updating is done on the same machine as the application program. The portNum is the port the update thread is listening on, and the className is the dynamic class that is to be hotswapped.
The first four cases are run using the original Inxar hotswapping library, while the last two cases have to be run with the modified library. In the modified library, the only important attribute in the examples.properties file is the 'compiler.destinationpath' which is the path to the .class file of the dynamic class.

5. Result

The experiments were done for different number of times in the for loop (See source code in the appendix). The objective of doing this is to compare the percentage increase for the various cases. Ten runs of each case were considered and the average running times are as shown below.

For 10,000 loops:

Case	Actual time (in ms)	Increase from the Basic case
Basic:	3.2	0
Inxar_I:	4000	1250
Inxar_II:	25156	7861.25
Inxar_Hotswap:	25062	7831.88
Modified_I:	3.2	0
Modified_II:	31	9.69

For 100,000 loops:

Basic:	31	0
Inxar_I:	39172	1263.61
Inxar_II:	239516	7726.32
Inxar_Hotswap:	235953	7611.38
Modified_I:	31	0
Modified_II:	285	9.19

As can be seen from the results, similar trends are observed for both running times. There is about 1250 times increase from the basic case to using a simple proxy class to do the hotswapping (Inxar_I). There is such a tremendous increase because every time the hotswap() function is called, even if there is no change in the dynamic class, a check is made to see if the source file is newer than the class file, and this is a very expensive operation.

There is also about 6 times increase in time from case Inxar_I to Inxar_II. This is because Inxar_II transparently calls the hotswap() function before calling each execute method. Similar trend is seen for Inxar_Hotswap case which also calls hotswap explicitly before every execute() method.

As is seen above, using the modified version of Inxar is very efficient. The program execution proceeds in a similar manner as the basic case until the hotswap signal is received. So as expected, the running time is the same as the basic case where no hotswapping is done.

The increase in execution time from case Modified_I to Modified_II comes from using the java.lang.reflect.Proxy class. Every time a method on the proxy is called, a Method

object and an array for its arguments are created and the invoke method of the InvocationHandler is called, which then forwards the call to the underlying method.

6. Conclusion

Dynamic classes provide powerful support for the maintenance and extension of critical and long-running applications. New implementation of a class can be dynamically added to and removed from a running program, eliminating the need to take down the application when fixing bugs, enhancing performance, or extending functionality. There are different approaches to achieve dynamic updating, and each implementation has its own advantage and disadvantage.

- Rather than modifying the JVM and being tied to a specific virtual machine, it is better to use proxy library. In addition, modifying the JVM requires detail knowledge of the internal structures and implementations of the JVM.
- The modified version of the Inxar implementation is an efficient way of updating classes dynamically in that it has very high performance, is easy to maintain and gives better control for multi-threaded applications.
- Even though using the dynamic proxy classes in the java.lang.reflect package makes the application program cleaner (by making the hotswap() call transparent), its performance cost is very high and should not be used if the dynamic class' methods are frequently called.
- Regarding instance updating method, although it is harder to implement, recreating all existing objects using the new class version is the most suitable method for many applications. Passive partitioning method, where only new objects reflect the new class, breaks the Java name-binding semantics and the barrier method, where updating is blocked until all old objects have expired, prevents updating of active classes and could also be costly to implement.

7. References

1. M. Dmitriev, "Safe Class and Data Evolution in Large and Long-Lived Java Applications," PhD Thesis, Department of Computing Science, University of Glasgow, Scotland, March 2001

2. M. Dmitriev, "Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications," Published in the Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference, Tampa Bay, Florida, USA, October 14-18, 2001

3. P. C. Johnston, Inxar - Information Exchange Architecture, http://www.inxar.org/hotswap

4. S. Malabara, R. Pandey, J. Gragg, E. Barr, J.F. Barnes, "Runtime Support for type-safe dynamic Java classes", Proceedings of the 14th European Conference on Object-Oriented Programming, Sophia Antipolis and Cannes, France, June 2000.

5. G. Hjalmtysson, R. Gray, "Dynamic C++ Classes, A lightweight mechanism to update code in a running program," Proceedings of the USENIX Annual Technical Conference(NO 98) New Orleans, Louisiana, June 1998

6. J. Andersson, P. Danielsson, T. Hulten, T. Ritzau, "Dynamic Deployment of Java Applications," 1999.

7. J. Andersson, "A Deployment System for Pervasive Computing", Department of Computer and Information Science, Linkopings University, Sweden, February 2000.

8. M. Hicks, "Dynamic Software Updating," Ph.D. thesis, Computer and Information Science Department, the University of Pennsylvania. August 2001