Towards Dynamic Plug-in Replacement in Eclipse Plug-in Development

Allan Raundahl Gregersen
The Maersk McKinney Moller Institute
University of Southern Denmark
Campusvej 55, DK-5230 Odense M, Denmark
allang@mmmi.sdu.dk

Bo Nørregaard Jørgensen
The Maersk McKinney Moller Institute
University of Southern Denmark
Campusvej 55, DK-5230 Odense M, Denmark
bnj@mmmi.sdu.dk

ABSTRACT
Although the Eclipse IDE offers an extremely useful built-in support for developing Eclipse plug-ins, it lacks the ability to perform dynamic updates of plug-ins in a running instance of the application being developed. Because of the nature of the Eclipse architecture and its strict class-loader delegation, plug-ins can only communicate through well-defined APIs. By applying a novel dynamic update approach to the eclipse plug-in development environment that exploits this knowledge, a new API are defined, namely the *Dynamic API*. This paper discusses some of the ordinary binary compatible changes that lead to erroneous program behaviour if not properly handled. Furthermore, it discusses how applying a dynamic update approach at development time gives developers a unique chance to experiment with dynamic updates without risking a costly shutdown of a real-life application.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces.* D2.6 [**Software Engineering**]: Programming Environments –*Integrated environments*

General Terms
Design, experimentation, verification.

Keywords
Binary compatibility, dynamic updating, plug-ins, eclipse.

1. INTRODUCTION
Recent versions of the Eclipse IDE come with a built-in support for plug-in development. This not only gives developers a great tool for writing plug-ins for the Eclipse IDE, but also brings the possibility to write entirely new applications as eclipse plug-ins on top of the Eclipse RCP. Although this plug-in development environment (PDE) already offers a lot of rich features, it does not support the ability to dynamically update a running plug-in at development time. The eclipse team has taken a step towards full dynamicity since 3.0 and now encourages well behaved dynamic plug-ins, so-called fully dynamic plug-ins, [6]. A system consisting of fully dynamic plug-ins can in theory perform in place updates of running plug-ins under some circumstances. Unfortunately it lacks the ability to migrate state, and furthermore no class of a plug-in can be directly referenced by a dependent plug-in without having to restart the workbench, which makes both updating and programming impractical. In order to support transparent and flexible live updates a more powerful mechanism is needed.

The problem of dynamic software updating is well known and in literature there have been a number of different proposals trying to solve it, e.g. [2], [4], [9], [13], [14], [15]. Although they all contribute with valuable insight to the problem domain, none of the proposals have been widespread accepted. The main reason for this might well be that they have been impractical or not been truly transparent from a developer's point of view which is supported by the statement in [17] "*Perhaps the most important lesson learned from this effort was that we significantly overestimated the amount of work developers would do in order to gain the benefits offered by an online software upgrading system*".

This paper proposes to introduce dynamic update capabilities at development time to give software engineers a unique opportunity to learn what can be done and more importantly what cannot be done when they apply code changes on the fly in a running instance of the program being developed. The novel dynamic update approach first presented in [8] is designed with the Eclipse infrastructure in mind thus it is a good candidate for inclusion in the Eclipse platform. It makes some assumptions regarding to binary compatibility, [7] on plug-in APIs in which this paper will elaborate on. Furthermore, this paper will give examples on how to deal with API changes that on the surface look unproblematic but when looking beneath show a number of subtleties that need to be tackled.

The remainder of this paper is organized as follows. Section 2 will briefly sketch the main ideas behind the dynamic update approach and thereafter present some of the API changes that need more attention regarding the approach. Section 3 will propose solutions to the problems defined followed by section 4 which gives a brief overview of practical implications. Section 5 then turns to the work related of this paper and section 6 concludes.

2. PROBLEMS OF BINARY COMPATIBLE API CHANGES
In order to understand the problems of specialized changes to plug-in APIs in relation to the specific dynamic update approach presented in [8], some introduction is needed.

2.1 Dynamic updating of active plug-ins
Component systems including the Eclipse platform are typically built on top of a runtime environment controlling the lifecycle of its components. In Eclipse its components, namely plug-ins, communicate through APIs which are defined in a separate

configuration file of each plug-in. To facilitate such communication and the ability to plug in new functionality at runtime a class loading policy, which assigns a class loader on a per plug-in basis, is required. While such a policy enables the runtime system to load new classes and thereby plug-ins, it does not support reloading already running plug-ins due to class-loader constraints held by Java. Per se Java only allows a limited reloading capability in the JPDA [18] which does not support reloading of classes in which methods, fields or inheritance relationships have been added or removed. Furthermore, additional constraints discussed as the version barrier in [16] prohibit type compatibility of any two classes loaded by different class loaders including loading of the same physical class file using two different loaders, thus excluding the possibility to just assign a new class loader with a plug-in that needs an update.

The approach in [8] overcomes these problems by letting multiple representations of classes and objects coexist for each version currently in use by any dependent plug-ins. This is done by using a technique mentioned as "In-Place Proxification" in combination with correct object correspondence handling. The idea behind the "In-Place Proxification" is to turn a running plug-in into a proxy of the new version on the fly. In order to do so code is automatically injected into every method and constructor reachable from a plug-in's API. This code contains a simple check to see if it should act as a proxy forwarding the request to the latest version in terms of a reflective call or run normally. As stated above the version barrier makes classes, and thereby their types (loaded by different class loaders) incompatible, which is why the approach keeps track of corresponding objects in so-called correspondence maps. Each live object in a currently running version of a plug-in is mapped at update time in an appropriate map to a corresponding object automatically created from the state of the former one. These maps are then used as fast lookup in the process of state migration at update time and at runtime for fast parameter and return value lookup when formal parameter types and return types have been declared in a plug-in from another version than what the caller or callee expects. Figure 1 illustrates the main ideas in the approach after applying one update of a single plug-in. It shows how communications across plug-in boundaries take place and the workflow of a method call from a class in plug-in A to a class in plug-in B given that plug-in B has gone through an update. Note in the figure that the return type of the method call has been put in as B_v1 which actually states that this type is declared by plug-in B in version 1. Furthermore the type of the parameter "a1" used in the first method call is likewise declared in plug-in B. This setup has been chosen on purpose to show how these objects cross the version barrier. What happens when the invocation reach method m in plug-in B version 1 is that the injected code checks if the class declaring the particular method is marked as a proxy, which should obviously be the case in this particular setup. Then the injected code asks the type-handler class which is central to the update manager to get a corresponding parameter value that will be type compatible with the formal parameter of the same method in plug-in B version 2. This is reflected in the figure by the arrow pointing to the upper correspondence map. Imagine what would happen if the parameter received by the original method in version 1 was directly redirected to the new version. An error would occur due to the version barrier. After retrieving the corresponding object, the new version of the method m is invoked from the injected code. As seen, the value returned is "b2" which has a type declared in plug-in B version 2. Off course, this object cannot be returned to plug-in A directly because the type of "b2" is not compatible with the one expected by plug-in A. This is why the type handler is asked to do a lookup in the reverse correspondence map to find the correct proxy version of the object which in turn gets returned by the original method m.

This concludes the brief introduction of the approach, but it should be noted that many important details concerning dependencies involving more than two plug-ins and type inheritance across plug-in boundaries and other specific problems have been left out. Please refer to [8] for more information.

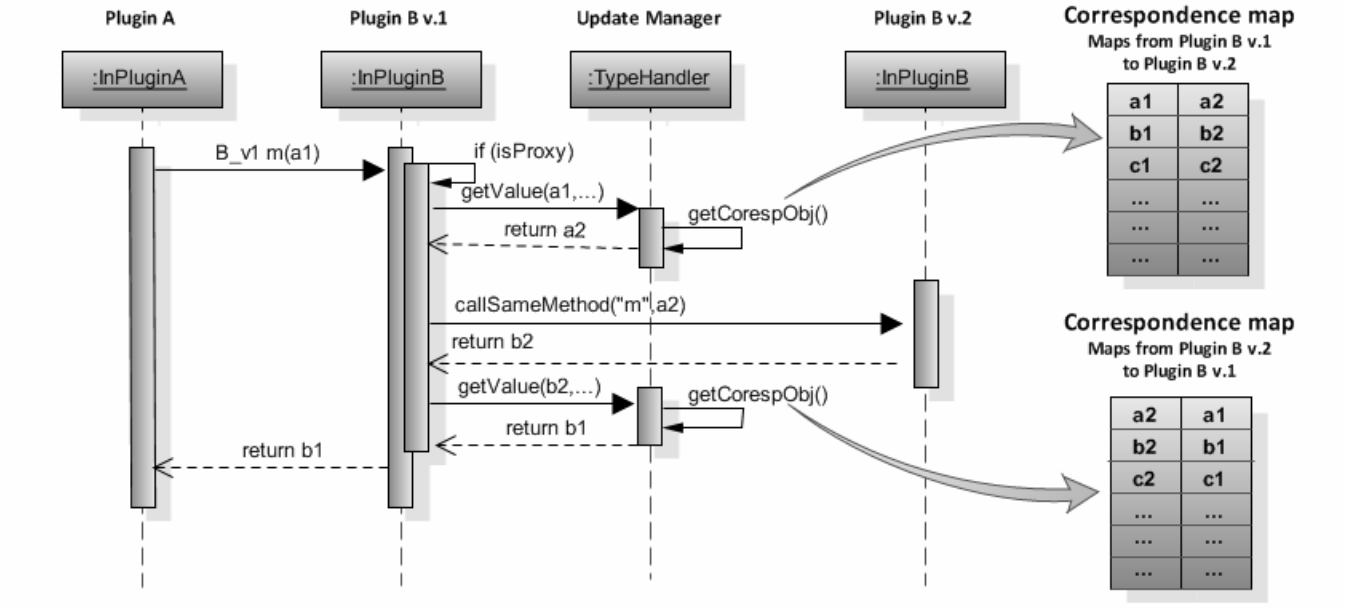

Figure 1: Workflow of dynamic update approach when plug-in B has been updated once.

2.2 The effects of changes in a new plug-in version

Having introduced the concepts of the dynamic update approach, this section turns to a discussion on how different changes to a plug-in affect the update mechanism.

As mentioned, the approach automatically injects code at load-time into the API of a plug-in in order to capture all incoming communications. Let us first argue where this code needs to be present. In figure 2 a small plug-in is illustrated.

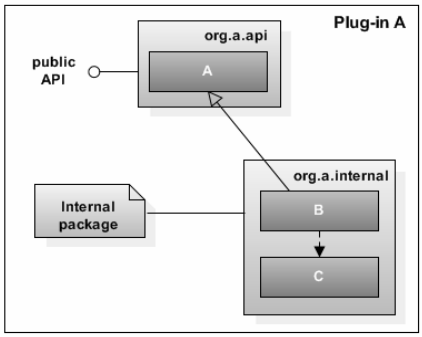

Figure 2: Plug-in with one API and one internal package.

At first sight only methods declared in class A are affected by cross plug-in communication as this is the only class in the public API. In fact, this might turn out to be true if class A does not contain any method that returns an object castable to class A. But consider if class A declares the following method:

```
public static A getCurrent();
```

The object returned from this method is either instance of class A or class B thus making class B reachable from other plug-ins. This actually results in an increase of the number of classes that need to have code injected. Throughout the rest of this paper the following terminology applies:

Direct API is the set of classes directly visible from a plug-in's API, whereas the **indirect API** covers the set of all classes not included in the direct API, and in which they are subclasses of any class returned directly from the API. Together the direct and the indirect API form the **dynamic API**.

Basically, the approach must statically analyse a plug-in and inject code in every method and constructor in the dynamic API to let a plug-in become dynamic enabled. To illustrate the effects of code changes from one version to another, the plug-in from figure 2 will work as a running example. First consider what happens if changes are made to the direct API. In fact, adding methods, fields and constructors in a new version of class A would not cause any problems as dependent plug-ins of the original version of A would still use the existing code as proxy without even knowing it. Only an updated dependent would see the additional functionality. On the contrary, consider if a method is removed from class A. This poses a problem for running dependents of this class as they expect the method to be reachable. One could argue that this type of problem has nothing to do with how the system is updated as a complete restart and deployment of the latest versions of the plug-ins will result in the same problem. However, the problem must be addressed in order to support continued execution even if a new version of a plug-in contains API changes that are not binary compatible.

More interesting scenarios show up when considering changes to the indirect API. Imagine that the developer of plug-in A decides to rename (or even remove) class B which is a perfectly binary compatible change in according to the java language specification in [7] as there are no problems from a linking point of view. The main issue here in relation to the dynamic update approach is the fact that currently running dependents could hold references to objects of the deleted classes in any previous version thus making them proxies to something that do not exist.

Another issue arises when adding new classes to the indirect API. Consider the addition of a new subclass of class A, say class D. When a dependent plug-in of a former version of plug-in A asks for an implementation of class A, the injected code might get an instance of the newly added class D from the latest version. This of course poses a problem for the injected code as it definitely fails to retrieve a compatible corresponding proxy instance simply because the class did not exist at the time of installing the dependent. To summarise, there exist two main categories of problems which need to be addressed by the approach. On one hand we have ordinary direct API breaking changes that would also pose problems for a none-dynamic update mechanism. On the other we have changes to internal parts of a plug-in resulting in possible erroneous execution if left unhandled. The following section will address these problems in turn

3. CHANGE SET AND DYNAMIC API VERIFICATION

Changes to the direct API are by default not supported in the approach. This means that if the update manager finds an API breaking change, the update is refused just like it would be in the existing update manager of the Eclipse IDE, providing that plug-in versioning follows the recommended guidelines in [5]. Part of the solution for successfully supporting dynamic updates of plug-ins is to support atomic dynamic updates of a set of plug-ins at once. This set of plug-ins, referred to as the **plug-in change set**, must enclose all plug-ins for which binary compatibility conflicts exist. Hence, a plug-in subject to a dynamic update and its dependent plug-ins with which it has binary compatibility conflicts is part of the plug-in change set. Thus, the change set is the set of plug-ins that must be redefined together for the dynamic update process to succeed. In practice though, this solution may result in postponing a particular update infinitively if e.g. a third party plug-in, of which further development has stopped, belongs to the plug-in change set. This is in fact a problem of general concern to any component platform regardless of the update mechanism used, which is also why API breaking changes are strongly discouraged.

The problems related to the indirect API are addressed somewhat differently. The overall goal of the approach is to allow all updates of plug-ins that do not break direct API compatibility (that is without having to wait for a new version of a dependent plug-in). In order to discover dynamic API incompatibility, the static code analyser (the component that also injects code into the dynamic API) must mark possible future dynamic incompatibilities so that the dynamic update manager can use this information when updating particular plug-ins. This actually just extends the idea described in [8] where such information is saved to a file in order to solve type sharing between three or more plug-ins or when inheritance across plug-ins occurs which is out of

scope of this paper. Having the right information at the time of an update, the update manager can always perform the correct operation. First consider the scenario where a class is deleted from the indirect API leaving running dependents with possible dangling references (the original objects are still there, but as proxies they cannot reference the new version of the object). This problem can be dealt with in two ways which are briefly outlined and thereafter discussed.

1. Allow the update and let dependent plug-ins continue their execution unnoticed.
2. Allow the update and perform additionally virtual updates of dependent plug-ins.

Well, of course, a third alternative could be to disallow the update in the first place which is really not an option in the sense that one of the most important goals is to support every possible none direct API breaking update. Looking at the first option it leads to a discussion of how the errors should be handled if or when they occur. Given that the approach is applied at development time one could argue that simply printing out a meaningful error would be a good choice. Indeed, the developer would be confronted with the error, but she will have no chance to deal with it at runtime resulting in a program breakdown. Instead, the approach could incorporate the second of the two options given above. The idea in a virtual update, as also described in [8] although used for a-different purpose, is to bring all dependents up to date with the plug-in causing a type-related problem as the one seen here. The idea is quite simple and mainly consists of replacing the class loaders of the dependent plug-ins so that the new types are recognized directly by all dependents. The virtual update process itself does not differ from a normal dynamic update, but a virtual update is easier in the sense that there are for sure no code changes between versions due to the fact that the same code base (same jar or simply a transient copy of the jar) is used. By using this approach no in-place proxy will ever end up in a situation where a corresponding method does not exist. However, problems may still surface because of the possible loss of state in a dependent plug-in of which a virtual update has been applied. Consider what happens in the following update scenario. The developers of plug-in A from figure 2 decide to delete class B from the indirect API and apply the update of plug-in A. Now a dependent plug-in of plug-in A, say plug-in B, holds a field of type A that before a dynamic update of plug-in B references an instance of class B. When the dynamic update process gets to the point where this particular field is migrated to the new version, it finds out that this class no longer exists. In general, there is no automatic way of replacing this field value with a corresponding object that could truly replace the former value. Therefore, the approach simply nulls such occurrences leaving a risk of unforeseen null pointer exceptions in the new client code. One could argue that following good practice for dynamic awareness as stated in [6] would solve the problem, but using this advice is not enough, because it does not take state migration into account.

Now consider the example where a class is added to the indirect API. Recall that in this case a proxy instance could not be obtained because the corresponding class was inexistent in the version used by running dependents. Contrary to when a class was deleted from the indirect API, this issue manifests itself at runtime after a dynamic update process. In fact, it shows only when the new version of a plug-in returns an instance of the newly added class to another plug-in through its direct API. More specifically, it returns to the proxy version of the direct API method called in the first place. To support this kind of change to the indirect API a mechanism to create a corresponding proxy class on the fly in the correct namespace is needed. This automatically created class should in turn be loaded by the class loader of the original version. Java provides some support for dynamic proxy generation through dynamic proxy classes [19]. Unfortunately dynamic proxy classes are limited to generating implementations of interfaces only, thus a more powerful mechanism is needed. Before choosing specific tools, in which in-place proxies can be automatically generated, it is in its place to give a brief outline of what kind of transformation it needs to support.

In-place proxy classes require a number of fields to support the dynamic update approach (all dynamic-enabled classes have these fields). All methods implemented by the newly added class should also be present in the proxy class. Furthermore, any interface implemented or super class extended by the newly added class that is not known by the “old” namespace of the original plug-in must be stripped out to avoid linkage errors or no-class-definition-found exceptions. At this point the attentive reader might wonder why the newly added class is not loaded directly with the class loader of the former version of the plug-in. In this case only unknown interfaces and extended classes should be cut off using some byte code manipulation tool and indeed it would provide a solution. The problem with this approach is the unnecessary memory footprint you get from the fields present before any dynamic enablement occurs, in addition to the application code written in every constructor and method. Therefore, it is by far better to generate a completely clean in-place proxy from the details of the class it should reference. The ASM byte code manipulation tool, [1] provides all the functionality needed to generate such classes. Furthermore, a number of relevant class and method transformations are presented in [12]. In this way the approach supports additional classes, whether they represent direct or indirect API classes, without introducing any runtime issues from a syntactic point of view. This concludes the description of solutions to the problems in relation to direct and indirect API changes.

4. PRACTICAL IMPLICATIONS

Having discussed solutions to the problems stated, this section briefly outlines how developers will use this feature in practice.

One of the main benefits of the approach is that it is practically transparent from a developer point of view. This makes it easily plug-in-able to the eclipse infrastructure, which is also reflected in the prototype implementation already spoken of in [8]. In fact the approach is completely automated and does not expect any metadata about the system other than that already present in the mandatory bundle manifest file. Adding it to the eclipse IDE as part of a development feature requires only a communication channel from the IDE instance to the currently running application. Once this channel is established it only takes one method invocation to the enhanced core plug-in in the running instance to perform the update.

Although the code injection and the actual update are completely automated the subtleties of dynamic API changes discussed can result in runtime exceptions after applying a dynamic update. In order for developers to understand them, they need some basic

knowledge of why these exceptions occur. For that reason introducing the approach at development time at first, gives developers a fair chance to familiarize with the implications.

5. RELATED WORK

In literature a lot of proposals to dynamic software updating exist. Some offer this support to languages that are not object-oriented e.g. [3], [9], thus not directly comparable to our work. Others extend a virtual machine to allow dynamic class replacement (e.g., [12, 16]). Malabarba et al. [12] propose dynamic Java classes in which a class can be replaced at runtime. Contrary to our approach, their update granularity is a single class. Due to Java's binary compatibility rules this reduces the set of applicable updates as it locks the class signature for all classes, not only those defined in an API.

A number of approaches suggest new language constructs to support runtime adaptation, e.g. [4]. Although these approaches are very useful for identifying the set of requirements for dynamic software updating, they are not likely to make it into the mainstream languages. Other techniques require a meta level programmer to write either the updated code as a separate layer as seen in [15] or to write patches/adapters as done in [2].

The approach presented in [14] shares similarities with our work as it does not require VM support nor does it need to introduce new language constructs. The approach uses a hot swapping mechanism based on indirection via object wrappers. One benefit of this approach is its negligible performance overhead. However, unlike our work, this comes at a price of less flexibility as class signatures cannot change without significant loss of state.

Bialek et al. in [2] present an approach based on partitioning Java applications to make those partitions the updateable unit. Although the approach provides a flexible solution to dynamic software evolution, it burdens the programmer with the responsibility of writing version adapters and state transfer functions, which is in deep contrast to the statement about how much work a developer would do to gain the benefits of dynamic updating capabilities.

The work done in JSR 277, [10], and JSR 291, [11], will hopefully bring a new module level into the Java language. If support for modules is added, the approach in this paper could be applied even outside the eclipse platform or any other component platform which has already implemented a module system itself.

6. CONCLUSION

This paper has suggested including a novel dynamic software updating approach into the eclipse infrastructure so that developers of eclipse plug-ins can do live updates while developing. The basic mechanism of the approach is the In-Place Proxification technique which lets already running objects become live proxies after an update. Furthermore a number of subtleties that manifest itself in the so-called dynamic API have been discussed and solutions are proposed so that any non breaking API change to a plug-in is directly supported. In addition virtual updates of dependent plug-ins are proposed as a mean to support API breaking changes, thereby enlarging the total change set when such changes occur.

7. REFERENCES

[1] ASM project web site, http://asm.objectweb.org/

[2] Bialek, R., Jul, E., J.-G. Schneider, J. –G., Jin, Y.: Partitioning of Java applications to support dynamic updates. In proceedings of APSEC'04, pp. 616-623.

[3] Chen, H., Yu, J., Chen, R., Zang, B., and Yew, P. 2007. POLUS: A POwerful Live Updating System. In Proceedings of ICSE'07, pp. 271-281.

[4] Duggan D.: Type-based hot swapping of running modules. In proceedings of ICFP'01, ACM Press, pp. 62–73.

[5] Eclipse Foundation, Inc.: Eclipse Version Numbering. http://wiki.eclipse.org/index.php/Version_Numbering

[6] Eclipse Foundation, Inc.: FAQ What is a dynamic plug-in? http://wiki.eclipse.org/FAQ_What_is_a_dynamic_plug-in%3F

[7] Gosling J., Joy B., Steele G., Bracha G.: The Java™ Language Specification, Third Edition. Prentice Hall (2005), ISBN 978-0-321-24678-3

[8] Gregersen, A. R., Jørgensen, B. N.: "Extending eclipse RCP with dynamic update of active plug-ins", in Journal of Object Technology, vol. 6, no. 6, July-August 2007, pp. 67-89. http://www.jot.fm/issues/issue_2007_07/article1

[9] Hicks M., Nettles S.: Dynamic Software Updating. In ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6. (2005) pp. 1049–1096.

[10] Java Community Process. JSR 277: Java Module System. Available at: http://jcp.org/en/jsr/detail?id=277.

[11] Java Community Process. JSR 291: Dynamic Component Support for Java SE. Available at: http://jcp.org/en/jsr/detail?id=291.

[12] Kuleshov, E.: Using ASM framework to implement common bytecode transformation patterns. Available at: http://aosd.net/2007/program/industry/index.php

[13] Malabarba S., Pandey R., Gragg J., Barr E., and Barnes F.: Runtime Support for Type-Safe Dynamic Java Classes. In proceedings of ECOOP'00. Lecture Notes in Computer Science, Vol. 1850. Springer-Verlag, (2000) pp. 337-361.

[14] Orso, A., Rao,A., Harrold M.J.: A Technique for Dynamic Updating of Java Software. In proceedings of ICSM'02, pp. 649-658.

[15] Redmond B., Cahill V.: Supporting Unanticipated Dynamic Adaption of Application Behavior. B. Magnusson (Ed.). In proceedings of ECOOP'02, Lecture Notes in Computer Science Vol. 2374. Springer-Verlag (2002) pp. 205-230.

[16] Sato Y., Chiba S.: Loosely-separated "Sister" Namespaces in Java. In proceed. of ECOOP'05. Lecture Notes in Computer Science, Vol. 3586. Springer-Verlag, (2005) pp. 49-70.

[17] Segal, M. E.: Online Software Upgrading: New Research Directions and Practical Considerations. In Proceedings of COMPSAC'02, pp. 977-981.

[18] Sun Microsystems inc.: Java Platform Debugger Architecture. http://java.sun.com/javase/6/docs/technotes/guides/jpda/

[19] Sun Microsystems inc.: Dynamic Proxy Classes. http://java.sun.com/javase/6/docs/technotes/guides/reflection/proxy.html