

From Java to UpgradeJ: An empirical study

Ewan Tempero

University of Auckland
e.tempero@cs.auckland.ac.nz

Gavin Bierman

Microsoft Research
gmb@microsoft.com

James Noble

Victoria University of Wellington
kjx@mcs.vuw.ac.nz

Matthew Parkinson

University of Cambridge
Matthew.Parkinson@cl.cam.ac.uk

Abstract

UpgradeJ is a variant of Java that offers linguistic support for lightweight dynamic software updating (DSU), or hotswapping. UpgradeJ allows co-existing multiple versions of classes and adapts Java's type system to provide incremental typechecking.

This paper provides some preliminary, but encouraging, results of an empirical study into the applicability of UpgradeJ. By analysing how classes in popular, open-source Java applications change from release to release, we are able to estimate the proportion of those changes that could be made dynamically in UpgradeJ. Although these applications were *not* designed with DSU in mind we find that many of the changes to classes could be supported by the UpgradeJ DSU model without any significant code rewriting.

1. Introduction

Dynamic software updating (DSU) is a technique by which a running program can be updated with new code and data whilst the program is still running. It is a critical technique for so-called non-stop applications such as networks, air-traffic control systems and financial transaction processing systems, which must provide continuous service but at the same time need to be updated to fix bugs and provide unanticipated additional functionality.

Moreover, DSU techniques are also applicable to non-critical systems where interruptions, whilst possible, are highly undesirable. An interesting example of this is described by Gille and Robinson [6] who consider writing bioinformatics applications where the datasets that are loaded in are typically very large. Application development often requires many cycles of optimization, compilation and testing, but the overhead of reloading the data dramatically slows down development time. They advocate the use of DSU techniques to facilitate more agile development.

Providing DSU requires a delicate balance between a number of competing concerns such as flexibility, safety, simplicity, predictability and overhead costs. Clearly the DSU programmer would like support for as many forms of updates as possible. However, arbitrary runtime updates would make reasoning about safety impos-

sible. Given the importance of mission-critical software, simplicity of the DSU model would help assurance of update correctness as would predictability of the update results. All this should be supported in a DSU system that comes with as little runtime overhead costs as possible.

In a recent paper, Bierman et al. [2] presented UpgradeJ, an extension of Java with language-level support for class upgrades. The key features of UpgradeJ are the ability to program with multiple co-existing version of classes, to upgrade classes dynamically, and a natural extension of the Java type system to support versioning and incremental typechecking. (We provide a more detailed introduction in §2.1.)

However, the original paper leaves open the (difficult!) question of whether the design of UpgradeJ has struck the right balance of the concerns mentioned above. This paper is a first attempt to empirically answer this question. The problem we face is exactly how to test the design of UpgradeJ. There is very little code that uses UpgradeJ, or more generally, any form of DSU, available for analysis. For this study we adopted a different approach. We studied successive versions of non-DSU applications (open-source Java applications) and determined the degree to which the changes in classes between versions are supported by UpgradeJ. If there are few such changes, that would suggest the balance is wrong. In fact, we found quite a large proportion of the changes at the class level are supported by UpgradeJ.

The rest of the paper is organized as follows. In §2 we give important background to this study. In §2.1 we give an overview of the UpgradeJ DSU model. In §2.2 we consider some other studies of software evolution in the literature. In §3 we give a detailed description of the methodology we employed in this study. In §4 we state and discuss our results. We conclude with pointers to further work in §5.

2. Background and Related Work

2.1 UpgradeJ

In this section we provide a quick overview of UpgradeJ, a fuller description with examples can be found elsewhere [2]. UpgradeJ extends Java by requiring class names (other than `Object` by convention) to be annotated by a version number in square brackets after the class name.¹

UpgradeJ also adds a new statement to Java, written `upgrade ;`. When this statement is executed, the program waits to receive an upgrade (either from a command prompt or from a file—this is intentionally outside the UpgradeJ model). The upgrade is then

Copyright is held by the author/owner(s).

HotSWUp'08, October 20, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-304-4/08/10.

¹For simplicity, we represent version numbers by positive integers.

typechecked and if correct, is incorporated into the program. Either way, the program then continues execution. Having explicit control of the timing of upgrades has proved beneficial in earlier DSU work [1, 11].

UpgradeJ supports three forms of upgrades:

1. **New class upgrade** This is the simplest form of upgrade, and allows for a new class definition to be added to the current class table. New class upgrades are most useful in combination with the other forms of upgrade.
2. **Revision upgrade** This allows for a new version of a class definition to replace an existing class definition. The new class definition is restricted to having the same fields as the old class definition, and the same “method signature” (by which we mean that objects of the new class understand the same methods and at the same types). Revision upgrades typically replace method bodies, and so correspond to bug-fix upgrades.
3. **Evolution upgrade** This also allows for a new class definition to replace an old class definition. The difference from a revision upgrade is that the evolved class may add new fields and new methods (but must still support the fields and method signature of the old class).

To allow upgrades to affect running programs, we provide new forms of annotations. As with Java, a new object is created by calling `new`, but in UpgradeJ this call must be annotated² to signify the sensitivity of the object instance to future upgrades.

1. The “exact” annotation, `=`, ensures that the instance ignores all upgrades to the class. For example, `new Foo[4=]()` creates an instance of version 4 of the class `Foo`. If class `Foo` is revised in a later upgrade, this particular object instance will not pick up any of the upgrade.
2. The “upgradeable” annotation, `+`, ensures that the instance is sensitive to all revisions to the class. Calling `new Foo[4+]()`, for example, creates an instance of version 4 of the class `Foo`. However, if class `Foo` is revised in a later upgrade, this particular object instance will use the revised method bodies for any subsequent invocations. (As a class is permitted to be successively revised we always invoke the most recent revision of the method.)
3. The “latest” annotation, `++`, also ensures that the instance is sensitive to all revisions to the class. Calling `new` with this annotation actually creates an upgradeable object of the *latest evolution* of a class. For example, assuming that the class `Foo` has been evolved to version 7, the call `new Foo[5++]()` creates an upgradeable instance of version 7 of the `Foo` class.

UpgradeJ was intended to be an experiment in lightweight support for DSU; and hence was designed to have as few runtime overheads as possible. Thus, the upgrades are at the class level: no object-level upgrading takes place. To do so requires either maintaining considerable runtime information about the heap [9] or scanning the entire heap at upgrade-time, however, supporting only “class upgrades” in UpgradeJ means that multiple versions of the same class can coexist. An important feature of UpgradeJ is that a class introduced as an upgrade is treated as a subtype of the class that it upgrades. A method expecting a `Bar` object of version 4, may always be passed a `Bar` object of version 7.

Another key design feature of UpgradeJ is its support for incremental typechecking. UpgradeJ never retypechecks any code—method bodies are typechecked at most once. Repeated typechecking potentially permits more upgrades but either requires heap-level

typechecking [1] or maintaining additional runtime data [9]. Being based only on typechecking, UpgradeJ can provide only syntactic, structural guarantees on upgrades.

2.2 Empirical studies of evolution

Most broadly, our study is on the evolution of software systems based on historical releases provided by the repositories of the applications we studied. The study of software evolution is a long and rich one, and we do not have space for more than a brief summary of the relevant material. Kagdi et al. provide a useful summary of software evolution from the point of view of extracting information from software repositories [7], and many of our comments summarise their work. In the history of software evolution research there have been many empirical studies done, considering many different artifacts relating to software, including defect reports, work reports (e.g., commits to version control), communication, and so on. The ones most relevant to us are those that focus on source code.

For our purposes, it is useful to roughly divide the study of evolution of source code into two categories: those that analyse each individual version (e.g., producing measurements from different metrics) and then compare the results of the analysis, and those that directly compare the source code—ours is the latter category. Within this category, different kinds of comparisons have been made, including at the textual or syntax level, comparison of abstract syntax trees, comparison of uses of patterns, and comparison of APIs. A common purpose for these kinds of studies is to determine how certain kinds of changes relate to such things as defect injection. In contrast, we are interested in the degree to which changes of a particular kind occur.

We will mention two specific examples. Kim et al. [8] looked at how *micro patterns*, which are “Java class-level traceable patterns” introduced by Gil and Maman [5], changed over time. They looked at micro pattern evolution over every compilable revision of three open-source Java applications. They were particularly interested as to whether the way micro pattern usage evolved over time was related to defect rates, in particular whether some forms of evolution were more bug-prone than others. Their results were inconclusive, but, like us, they looked at changes at a higher level of abstraction than syntax.

Dig and Johnson looked at evolution of APIs, that is, the intended interface of libraries intended to be used in other applications [4]. Their overall goal was similar to ours, namely to understand how systems change so as to determine how to be support change. They were interested in whether changes to APIs were *breaking*, that is clients of them would not work correctly with the new version, or *non-breaking*. They considered two versions of 5 Java frameworks and libraries, 4 open-source and 1 proprietary. They found that most of the API breaking changes were due to refactorings. Most of the refactorings we would classify as neither revision nor evolution, although some we don’t consider (deleted classes for example).

Studies of software evolution specific to the issues of DSU are much fewer. A number of large C programs (including Linux, Apache and OpenSSH) were studied by Stoye et al. [11], who identified a handful of key trends. Recently, Cech Previtalli and Gross [3] have performed a detailed case study of the evolution of over twenty releases of Tomcat 5.5. Broadly speaking, their results are similar to ours, although a more detailed comparison is future work.

3. Methodology

Our approach for our study was to compare consecutive pairs of releases of several open-source Java applications, and determine how the classes changed as the software was developed. Because

² Although one might imagine tool support to alleviate some of this burden.

not all releases are available, we choose pairs of releases where the time interval between the releases in a given pair does not overlap with the time interval of any other pair. In the discussion below, we refer to the earlier release as the “old version” and the later release the “new version.” The applications are described in more detail below. For each pair, we produce measurements showing how the versions differ with respect to the support provided by UpgradeJ. We discuss these metrics in more detail below also.

3.1 Entities Measured

We have measured attributes of 11 open-source Java applications from the Qualitas Corpus [10]. They are summarised in Table 1. A representation of their size is given later in Table 3. For our purposes, the code measured for an “application” is only the code written for that application, that is, not including the standard library or third-party code. Each release was analysed at the bytecode level, rather than source code.

3.2 Metrics

The goal of our study is to determine how useful the support provided by UpgradeJ actually is, that is, when systems change, how often do they change in a way that UpgradeJ can represent. Generally, the kinds of changes that UpgradeJ applies to are when classes change from one release to the next, that is, it doesn’t apply to changes that consist of removing methods or classes. If a class does change, then, as far as UpgradeJ is concerned, there are three possible kinds of changes: the type of the class doesn’t change (but the implementation does), the type of the class changes but the type in the new release is a subtype of that in the old release (and implementations might change), or the type of the class changes in a way that the two types are incompatible. We have developed metrics to reflect these possibilities.

While the list above is complete, it does not take into account the possible *mechanisms* that cause the change. In particular, there is a way for a class to change without any change to its declared fields and methods, namely if its *parent* (superclass) changes. Unfortunately, changing a class’s parent does not necessarily change the class’s type, and so the analysis for this case is non-trivial. For the purposes of this study (and because we don’t expect it to happen very often) we just measure superclass changes separately.

With the points above in mind, we use the following metrics in our study:

- N** The number of *classes* (that is, not interfaces, enums, or annotations) in the *old* version of the application being measured. This includes nested classes.
- C** (Changed) The number of classes that are in both versions of the application that have changed in some way. We also use **C%**, which is **C** as a proportion of **N**.
- E** (Evolved) The number of changed classes for which their type in the new version is a subtype of their type in the old version. We also use **E%**, which is **E** as a proportion of **C** (not **N**).
- R** (Revised) The number of changed classes for which their type in the new version is the same as their type in the old version. We also use **R%**, which is **R** as a proportion of **C** (not **N**).
- A** (Adopted) The number of changed classes where their only change is that their parent was `java.lang.Object` in the old version but is something else in the new version. (Also **A%**.)
- S** (Stepchild) The number of changed classes where their only change is that their parent has changed between versions, and in neither case is `java.lang.Object`. (Also **S%**.)
- O** (Orphaned) The number of changed classes where their only change is that their parent is now `java.lang.Object` in the

new version, but was something else in the old version. (Also **O%**.)

The key points here are that the **E** and **R** categories correspond to changes that are directly supported by UpgradeJ evolution and revision upgrades respectively. The **A**, **S**, and **O** categories capture changes to superclasses.

4. Results and Discussion

4.1 Results

We do not have space to show all the measurements we have gathered for every application. Table 2 gives the measurements for one application, the azureus distributed BitTorrent client, which are both representative of all the data, and contain some interesting points. (Table 3 below gives a summary across all applications and all versions.)

In interpreting Table 2 it is important to note that the “distance” between versions compared is not the same in all cases. For example, version 2.0.8.4 is almost certainly intended to be only a small change from 2.0.8.2, whereas moving from version 2.0.8.4 to version 2.1.0.0 is probably intended to be a larger change. However note that **C%** for 2.0.8.2→2.0.8.4 is 2%, whereas an apparently same level of change, 2.1.0.2→2.1.0.4, has a measurement of 14%. We also note that not every release has been measured. For example, there was almost certainly a release between 2.3.0.4 and 3.0.3.4. (We assume the azureus developers follow a common practise of using even numbers for stable releases and odd numbers for unstable ones, meaning there probably was not a public release between, for example, 2.0.8.2 and 2.0.8.4.) We have not measured the missing releases because they were not in the corpus at the time we conducted the study (and some appear to be no longer available).

Looking at the measurements in more detail, we note that the proportion of classes changed can vary from very little (2%) to most of the classes (71%), and recall that we have not shown number of classes added or removed. For **E%** and **R%**, these measurements are typically far from 0, and in fact the proportion of changed classes that are either evolved or revised is greater than 50% in most cases.

One interesting feature of the azureus results is the **S** measurement for change 2.1.0.2→2.1.0.4. Recall that this measures the number of classes whose parent was not `Object` initially and changed to another class that is not `Object`. While we speculated this kind of change would not happen much, this measurement (and others for azureus—some other applications have a small number of non-zero measurements also) shows it does happen. In this case, the 29 classes originally had `java.lang.Thread` as their parent, and this has been changed to an application specific subclass of `Thread`.

Table 3 gives a summary of the measurements for all applications and all versions, presented as ranges that the different metrics take. We have also included the values for the proportion of changed classes that are either evolved or revised (**E+R%**). Some of the **E+R%** values look very encouraging, specifically those that are 100%, however in all cases the number of changed classes was relatively small, the largest number being 19 (including for azureus 2.0.8.2→2.0.8.4). However, the largest number of classes changed (for eclipse 3.0.2→3.2), was 7207, of which 62% were either evolved or revised.

4.2 Discussion

The aim of our study is to assess whether UpgradeJ would be expressive enough for DSU programming. While there are many issues that determine how useful something is, the data we have collected shows that much of the natural evolution of Java programs could be supported as *dynamic* upgrades in UpgradeJ. In other

Application	Description	V	Earliest	Latest
ant	Java-based build tool	14	18/7/2000 (1.1)	19/12/2006 (1.7.0)
antlr	Parser generator.	10	18/9/1998 (2.4.0)	7/12/2005 (2.7.6)
azureus	BitTorrent client, providing peer-to-peer file-sharing.	11	14/3/2004 (2.0.8.2)	4/10/2007 (3.0.3.4)
eclipse	Integrated development environment	13	7/11/2001 (1.0)	18/1/2006 (3.1.2)
freecol	A turn-based strategy game.	12	30/9/2004 (0.3.0)	27/1/2008 (0.7.3)
hibernate ³	Provides object-oriented persistence.	9	30/11/2001 (0.8.1)	28/4/2008 (3.3.0.cr1)
jgraph	Framework for visualising and laying out graphs.	30	24/5/2005 (5.5)	28/11/2007 (5.10.2.0)
jmeter	Application designed to load test functional behaviour and measure performance.	8	3/2/2003 (1.8.1)	19/8/2005 (2.1)
jung	A framework that provides a common and extensible language for the modelling, analysis, and visualisation of data that can be represented as a graph or network.	17	31/7/2003 (1.0.0)	19/10/2005 (1.7.1)
junit	A unit test framework.	13	8/1/1998 (2.0)	18/7/2007 (4.4)
weka	Collection of machine learning algorithms for data mining tasks.	13	22/2/2000 (3.0.1)	18/12/2007 (3.5.7)

Table 1. Entities measured **V** is number of versions, **Earliest** is release date of earliest version measured, **Latest** is release date of latest version measured.

Versions	N	C	C%	E	E%	R	R%	A	A%	S	S%	O	O%
2.0.8.2→2.0.8.4	929	19	2%	1	5%	18	95%	0	0%	0	0%	0	0%
2.0.8.4→2.1.0.0	986	370	38%	49	13%	166	45%	0	0%	0	0%	0	0%
2.1.0.0→2.1.0.2	1253	253	20%	61	24%	105	42%	6	2%	0	0%	0	0%
2.1.0.2→2.1.0.4	1378	190	14%	37	19%	80	42%	0	0%	29	15%	0	0%
2.1.0.4→2.2.0.0	1408	997	71%	114	11%	556	56%	0	0%	0	0%	0	0%
2.2.0.0→2.2.0.2	1573	452	29%	64	14%	256	57%	0	0%	0	0%	0	0%
2.2.0.2→2.3.0.0	1710	550	32%	106	19%	225	41%	0	0%	0	0%	0	0%
2.3.0.0→2.3.0.2	2317	249	11%	52	21%	118	47%	0	0%	0	0%	0	0%
2.3.0.2→2.3.0.4	2338	320	14%	44	14%	148	46%	0	0%	0	0%	0	0%
2.3.0.4→3.0.3.4	2410	1658	69%	241	15%	514	31%	1	0%	0	0%	0	0%

Table 2. Measurements for azureus.

App	N	C	E%	R%	E+R%
ant	95-938	14-565	1-48	14-99	29-99
antlr	35-169	5-102	0-40	27-93	64-100
azureus	929-2410	19-1658	5-24	31-95	46-100
eclipse	5585-17594	62-7207	2-35	31-69	53-92
freecol	215-689	39-316	15-90	5-64	30-94
hibernate	61-1048	29-446	10-39	0-65	10-80
jgraph	71-73	1-25	0-100	0-100	67-100
jmeter	328-630	25-280	12-40	32-85	59-97
jung	101-558	7-136	0-57	29-86	54-100
junit	36-93	8-63	0-42	22-83	37-94
weka	147-1640	11-1059	6-68	16-91	50-100

Table 3. Summary of results. All numbers given are *ranges* taken by the measurements for the applications over all the versions examined. E+R% is the sum of E and R as a proportion of C.

words, the applications could be upgraded “in flight” *without* the need for stopping the application, persisting any important state (if possible), installing the new version, and then restarting the new version of the application with any persisted state reinstated (again, if possible). For the classes that change between releases it was usually the case that more than half of them were either evolution or revision upgrades, using the UpgradeJ classification.

The number of changed classes does not appear to strongly affect the degree to which changes are evolution or revised. Even when the number of changed classes is in the thousands, the majority of those changes are supported by UpgradeJ. This is encourag-

ing as none of these applications were written with DSU in mind and yet we find most of the changes are in fact supported by UpgradeJ.

In the data in Table 2, the proportion of classes that are revised is always larger than the proportion evolved, suggesting that such changes are the norm. However this is not true in general, although it is usually true, and even when it is true the proportion of classes that are evolved is often significant.

We also have detailed data as to why a given class got the measurement it did, and in the future could look at (for example), of the members changed in a given class, what proportion of the changes is consistent with revision or evolution. For example, it is possible that a class has 10 methods with changed implementation and 1 method with a parameter added. This class would not be considered revised (or evolved), even though in some sense it was very close to being so. Preliminary analysis suggests that changes to members (changing the type of a field or the signature of a method) are quite rare. Removal of methods tends to be more common than removal of fields, although in both cases it is rarely the case that more than 2 fields or methods are removed. It also appears to be the case that it is very rare that a field is removed but a method is not.

The goal of this study was to determine the degree to which changes to individual types were supported by UpgradeJ, and our results suggest that such changes are the majority. However, if there is one change that is not supported by UpgradeJ then the whole revision cannot be dynamically updated. So, for azureus for example, only 2.0.8.2→2.0.8.4 could be dynamically updated. All of the other new releases had changes not supported by UpgradeJ.

Whether UpgradeJ is sufficient to support dynamic updates of a whole system requires a better understanding of the changes we observed that are not supported by UpgradeJ as discussed above. We also note that the systems we studied were modified without DSU in mind. There are design strategies supported by UpgradeJ that allow for quite extensive changes, as discussed in the original paper [2]. Had we found significantly fewer changes supported by UpgradeJ in the real systems we studied, it would be harder to believe a different design strategy would allow UpgradeJ to be useful, as it is, we found the results quite encouraging.

4.3 Threats to Validity

To extend our conclusions to beyond just those systems studied, we must also consider how representative the studied systems are. All of the systems we studied are successful, open-source, and in Java. Even for this restricted population our sample is fairly small. Nevertheless it does cover a variety of domains, development teams, and sizes, and so we believe our results are more widely applicable.

The granularity at which we studied the evolution of these systems was the “release” level, rather than every revision checked into the version control system. There may be something going on at the finer resolution that we cannot detect. For example, it is conceivable that a class that we show as being revised between to releases actually went through a number of changes and at some point two revisions of it were incompatible with each other. This is a direction for possible future work. However the resolution in some sense corresponds to a commitment made by the developers, and so our results are representative of the final decisions, even if they are not of intermediate ones.

We identified classes by their fully-qualified name. Classes whose names were changed would therefore be recorded as the old name being deleted and the new name being added.

Because we analyse bytecodes and not source code, it is possible that changes we have recorded as being implementation-only changes (revisions) were in fact caused by a change to the compiler. This is certainly possible, although we note that it is likely that if a compiler changed, then *every* method’s (bytecode) implementation would have changed and the measurements would have reflected that. The same argument applies to techniques such as obfuscation. So long as the obfuscation techniques are the same, our results are valid. (In fact we have not encountered application of obfuscation in the open-source applications we studied.)

A problem we encountered with hibernate was that its package structure changed twice between the “1” sequence of releases and the “3” sequence. Because we deal with full-qualified class names, changes in the package structure mean changes in the class name, and so we could not compare releases with such changes at all. It also appears that the difference sequences are to some degree developed independently, and so it may not make sense to consider (for example) the last release of the “2” sequence to be the predecessor to the first release of the “3” sequence. As it happens, due to the change in the package structure we don’t compare versions from different sequences, but a similar development philosophy may be used in other applications.

5. Conclusions

DSU is a technique for upgrading software “in flight”. Whilst there are a number of low-level ways for achieving this, we are more interested in high-level approaches that combine simplicity and predictability with safety. However, it is far from straightforward to assess whether such a system is expressive enough to allow the DSU programmer to build DSU software.

In this paper we have tested the expressivity of UpgradeJ: a lightweight, language-level proposal for DSU based on Java. We

have taken a number of versions of 11 open-source Java applications (in total, 150 complete application builds) and analyzed the way these applications have evolved. Our primary focus is to determine whether the way classes evolved in these applications would be considered as valid upgrades in the UpgradeJ system.

This is a particularly tough test as none of these applications were written with DSU in mind. In our experience, DSU programs are architected differently to offer maximal upgradeability. That said, the sort of changes that can be observed between successive versions of non-DSU programs provide us with probably the best clue as to how DSU software might evolve. What we found was that for 6 of the applications, for every new release, at least 50% of changes to classes could be expressed with UpgradeJ. Even for the other applications, for most releases, changes to classes were more likely to be supported by UpgradeJ than not.

Whilst preliminary, our study does suggest that the lightweight support for DSU in UpgradeJ is sufficient for much of the typical object-oriented software evolution seen “in the wild”. Clearly, this claim requires further study—in particular we should like to review the changes that are not directly supported by UpgradeJ. It may well be the case that a simple rewriting of the code would make them directly supported. More detailed analysis of the codebase is required here. Another possibility is that such an analysis of the codebase might reveal other useful forms of DSU that are not currently supported by UpgradeJ. Similar corpus studies could also be used to investigate DSU support in other languages, or to evaluate extant DSU mechanisms.

Acknowledgments

This work was done while Tempero was a visiting researcher at the BESQ centre at the Blekinge Institute of Technology, Sweden, whose support he gratefully acknowledges. We also thank the anonymous reviewers for their helpful comments.

References

- [1] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proceedings of USE*, 2003.
- [2] G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proceedings of ECOOP*, pages 235–259, 2008.
- [3] S. Cech Previtali and T. Gross. A case study for aspect-based updating. In *Proceedings of RAM-SE*, 2008.
- [4] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution*, 18(2):83–107, 2006.
- [5] J. Y. Gil and I. Maman. Micro patterns in Java code. In *Proceedings of OOPSLA*, pages 97–116, 2005.
- [6] C. Gille and P. N. Robinson. Hotswap for bioinformatics: A STRAP tutorial. *BMC Bioinformatics*, 7(64), 2006.
- [7] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution*, 19(2):77–131, 2007.
- [8] S. Kim, K. Pan, and E. J. Whitehead Jr. Micro pattern evolution. In *Proceedings of MSR*, 2006.
- [9] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of ECOOP*, pages 337–361, 2000.
- [10] Qualitas Research Group. Qualitas corpus release 20080603. <http://www.cs.auckland.ac.nz/~ewan/corpus/>.
- [11] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutantis: safe and predictable dynamic software updating. In *Proceedings of POPL*, pages 183–194, 2005.