

Improving the Transparency of Proxy Injection in Java

Hendrik Gani and Caspar Ryan

School of Computer Science & IT, RMIT University
Melbourne, Victoria, Australia

{hgani, caspar}@cs.rmit.edu.au

Abstract

Due to their flexibility, proxies have been used for various purposes in standalone and distributed applications. More specifically, object-level proxies support fine grained capabilities and offer the potential to transparently (i.e. with minimal human intervention) inject proxy-based functionality into an ordinary application. Consequently, several solutions based upon code transformation for addressing various limitations of transparency in existing approaches are considered and evaluated. Furthermore, the automation of the majority of the required code transformation has been implemented, which along with the deployment of various (proxy-based) adaptive and test applications, demonstrates the transparency, correctness, flexibility, and practicality of the solution. As the solution was presented in terms of Java, the discussion involves Java-specific characteristics; however some of the more general concepts should be useful for addressing similar issues in other object-oriented languages.

Keywords: Proxy, Object, Transparency, Transformation

1 Introduction

In the general sense, a proxy (Gamma et al., 1995) refers to a placeholder for another entity. Due to its capability to intercept communication between the calling/source entities and the target/proxied entity, it allows deferring the creation/initialisation of the target entity, redirecting the communication target to one or more entities, performing additional capabilities before/after communicating with the target entity, etc. As such, proxies have been applied for various purposes, such as memory efficiency (Gamma et al., 1995), remote communication (Sun Microsystems, 1994-2008c), concurrent data evaluation/processing (e.g. as a *future* object (Pratikakis et al., 2004)). Proxies have also been used in component-based middleware such as EJB (Sun Microsystems, 1994-2008b), to provide various enterprise-level capabilities. Furthermore, proxies have also been widely adopted in contemporary work on application adaptation (Holder et al., 1998, Philippsen and Zenger, 1997, Rubinsztein et al., 2005, Ryan and Westhorpe, 2004, Tatsubori et al., 2001, Tilevich and Smaragdakis, 2002), since they allow certain components of an application to dynamically reconfigure themselves (e.g. changing its behaviour) transparently, i.e. without the awareness of the rest of the components in the application.

Note that although proxies can also be used at application level, e.g. web proxies, this paper only focuses on *object-level proxies* since the finer granularity allows proxies (including the functionality implemented in the proxies) to be injected/inserted into an existing Java application, i.e. by proxying *certain objects* of the application. In particular, this paper aims to improve the *transparency* of object-level proxies in order to enable the *injection* of proxy-based functionality into an ordinary application with *minimal human intervention*. This not only eases the *development* of *new* applications, since the developer can focus on the application logic (rather than the added/extra capabilities), but also assists the *extension* of *existing* applications with minimal development effort.

While object proxies are normally used to support generalised functionality that is common to different classes and applications, such as adaptation, caching, mobility, security, as demonstrated in previous work (Holder et al., 1998, Rubinsztein et al., 2005, Ryan and Westhorpe, 2004, Sun Microsystems, 1994-2008b, Sun Microsystems, 1994-2008c), the proxies themselves are not generic since each proxy class has to be explicitly authored for a specific (Java) class. Consequently, proxy classes are often *automatically generated* (at source code or byte code level) in order to avoid the routine task of writing repetitive and duplicated proxy functionality. In addition, inserting the generated proxy code into existing application code requires *modifying* certain classes of the application. The modification task should also be automated as much as possible to reduce intervention from the application developers and thus improve transparency.

Several general-purposed *code transformation* tools for *generating* and *modifying* code have been developed in the past, which include BCEL, ASM, TXL, Antlr, and AspectJ. On the other hand, the MobJex framework (Ryan and Westhorpe, 2004) includes a source code pre-compiler which is specifically targeted for inserting generic capabilities into existing code. Although these tools play an important role in leveraging development transparency; it is beyond the scope of this paper to evaluate these tools or discuss the mechanics of these tools.

Instead, this paper focuses on the code generation and modification that is required for maximising the compatibility between the proxy and the original/proxied class so that proxies can be used in place of the original objects with minimal effort. As such, the *less human effort* required for introducing proxies in an existing non-proxy application, the *more transparent* the solution. The rest of the paper is structured as follows. Section 2 provides a literature review on the approaches adopted in related work outlining room for improvement in the approaches as well as certain aspects of transparency that have not been addressed. Section 3 proposes and compares a number of approaches that address the issues identified in section 2,

followed by a discussion (section 4) of the evaluation of the transparency, correctness, and the impact that the proposed solutions have on application performance and resource consumption. Finally section 5 concludes the paper with a summary and a discussion of future work.

2 Transparency Issues

This section discusses the fundamental issues of proxy transparency that this paper aims to address. The discussion also includes how related existing work approached the issues, outlining the strengths and limitations of the approaches.

Of the reviewed object-level proxy approaches adopted in (Eugster, 2006, Holder et al., 1998, Philippsen and Zenger, 1997, Pratikakis et al., 2004, Rubinsztein et al., 2005, Ryan and Westhorpe, 2004, Sun Microsystems, 1994-2008b, Sun Microsystems, 1994-2008c, Sun Microsystems, 1994-2008a, Tsubori et al., 2001, Tilevich and Smaragdakis, 2002), the approaches proposed by Eugster (Eugster, 2006), Tilevich et al. (Tilevich and Smaragdakis, 2002), Ryan et al. (Ryan and Westhorpe, 2004), Tsubori et al. (Tsubori et al., 2001) as well as the approach adopted in JavaParty (Philippsen and Zenger, 1997) version 1.9.5 are most relevant to this work since these approaches employ automatic generation of proxy code artefacts to alleviate the developer from the burden of writing proxy-related classes. Furthermore, these approaches also utilise a certain degree of automatic code modification to minimise development effort and thus improve transparency. Nevertheless, certain transparency aspects of the existing approaches have not been addressed, while other aspects were addressed but the solutions can be improved or extended. Consequently, this paper aims to provide a more transparent approach by addressing these aspects/issues. Section 2.1 – 2.8 will discuss the fundamental issues of proxy transparency as well as how they were approached in the previous work.

2.1 Proxy Inheritance

In order to maximise the type compatibility between the proxy class and the original to-be-proxied class, a *proxy* class should also extend/inherit from the parent of the original class. Several aspects of proxy inheritance, such as proxying the attributes of the parent classes and extending *external* classes (e.g. system/core or library classes); have yet to be explicitly addressed due to the limitations of previous approaches and the lack of discussion on these aspects in previous work.

The traditional proxy approach adopted in previous work, such as (Holder et al., 1998, Ryan and Westhorpe, 2004, Sun Microsystems, 1994-2008b, Sun Microsystems, 1994-2008c) uses a Java interface acting as a common interface (also known as *domain* type) for both the *proxy* and *original* class, thereby allowing the methods of the *proxy* and the *original* object to be accessed in the same manner. Ryan et al. (Ryan and Westhorpe, 2004) proposed a solution for allowing the proxy class to be used transparently in place of the original class. In this approach, an interface X is named after the original class X and declares the public methods of the original class. The original class source code itself is transformed into an implementation class X_{Impl} , containing the inserted

capabilities. Likewise, the generated proxy is named X_{Proxy} . Note that subsequent discussions will refer to the original class that is being or has been transformed as the *implementation* class as opposed to the *original* class which is the class prior to the transformation.

The approach has some limitations; the major one being the inheritance hierarchy of the original class B is not fully reflected in the domain interface because a Java interface cannot extend a class, i.e. the parent of the original class. One possible solution is to proxy the parent class A , which as a result will be transformed into an interface thus allowing the child *domain interface* B to extend the parent *domain* A . However, this is *not* practical if A is an external class (e.g. system or library class), the modification of which could affect dependent classes, e.g. those that need to access A directly rather than via a proxy.

Eugster (Eugster, 2006) proposed a flexible class structure for proxies whereby the proxy class extends from the original class thus maintaining the type compatibility between the proxy and the original class thus allowing the proxy to be used wherever the original type is expected. Although this class structure is flexible enough for accommodating the inheritance case where a proxied class A extends another proxied class B , an explicit discussion of the solution and its overhead implication was not provided in (Eugster, 2006) and hence is considered in section 3.1

On the other hand, although the work on JavaParty (Philippsen and Zenger, 1997) and J-Orchestra (Tilevich and Smaragdakis, 2002) addressed this inheritance case, the proxy class structure proposed in the work does not accommodate the case where parent class B is an *external* class. This is because these frameworks require proxy class B to extend a *special class* (i.e. $Proxy$ in the case of JavaParty and $UnicastRemoteObject$ in J-Orchestra) and modifying an external class to extend the *special class* will most likely break existing dependent code.

In order to improve proxy transparency, this paper proposes and compares several alternative proxy class structures (in section 3.1) to address various class inheritance cases. Furthermore, other related issues such as proxying parent methods with various modifiers, e.g. final, private, etc. will also be addressed.

2.2 Proxy Instantiation

In order to be able to use proxies in an ordinary Java application without explicit authoring of proxy-related code, there has to be a mechanism to transparently acquire the proxy instances. Some technologies such as *Java RMI* (Sun Microsystems, 1994-2008c) and *dynamic proxy* (Sun Microsystems, 1994-2008a), mandate the programmer to explicitly create a proxy/stub using the provided API methods. As such, any derivative of such technologies such as that proposed by Eugster (Eugster, 2006) which was based on *dynamic proxy*, also has the same limitation.

Other technologies such as EJB 3.0 (Sun Microsystems, 1994-2008b) provide a more transparent solution by utilising a concept called dependency injection, which prevents the programmer from having to explicitly instantiate proxies but instead should provide a setter method for the proxied component (e.g. a session bean). The EJB container/middleware will then create a proxy instance and pass the instance to the setter method during

the initialisation phase. Although this approach improves transparency by removing the need for explicit instantiation, applications have to be written according to the styles/policies imposed by the framework.

In contrast, the proxy approach used by JavaParty and J-Orchestra provides a fully transparent way of instantiating a proxy since the proxy is named after the original class. As such, when a client attempts to instantiate the class (e.g. using “new” operator), the corresponding proxy gets instantiated instead. Note that the instantiation of the target/original class itself is done transparently (from the client’s point of view) by the proxy. Though proxy instantiation is not an issue that requires an explicit solution in JavaParty and J-Orchestra, this might not be the case in other proxy approaches, in which case a solution is required as will be discussed in section 3.4.1.

2.3 Field Access

In Java, the client of an object interacts with the object via its fields and methods (including those that are inherited from the parent class). Methods of a target/proxied object can be invoked in a transparent manner via method call *forwarding*, i.e. forward the call from a method of the proxy to the corresponding method of the proxied object. However, this solution is not applicable for field access because unlike a method, in which the functionality can be arbitrarily defined (i.e. thus can implement the *forwarding* logic), a field only has a fixed set of operations, i.e. read and write, thus a proxy cannot delegate to the proxied object when its field is accessed. The solution to this issue involves the generation of the corresponding *field accessors* (i.e. *setter* method for writing and *getter* for reading) as well as the modification of the client code to invoke the appropriate *field accessors* instead of directly accessing a field. This solution has been discussed in sufficient details in (Eugster, 2006) and therefore section 3.4.2 only discusses outstanding field-related issues.

2.4 Reflective Operations

In addition to the standard object creation using the “new” operator, Java provides two other ways of instantiating a class. Both ways are achieved using Java reflection API. Note that the term “*reflection*” used in this paper should *not* be confused with the ability of an application to reflect and modify its components/behaviour accordingly, which is also known as *application adaptation*. Instead, the term “*reflection*” is used to refer to the ability to perform class/object-level operations (e.g. instantiation, method invocation) without the need to explicitly hardcode certain information (e.g. class/method names). As such, in the case of reflective instantiation, unlike the issue described in section 2.2, there is no reliable static/compile-time solution since the concrete type of a reflection class can only be accurately determined at runtime. Since none of the reviewed work has addressed this problem in detail, a solution will be proposed in section 3.4.1.

On the other hand, reflective method invocation does not pose a new problem since method invocation in Java is always resolved dynamically at runtime due to polymorphism regardless of whether the invocation is done via reflection. In contrast, reflective field access introduces a problem similar to reflective instantiation, i.e.

the owner/class of the field cannot be determined at compile time as will be discussed in section 3.4.2.

2.5 Static Attributes

Static attributes (i.e. fields and methods) in Java refer to those that belong to a class rather than the instance. In traditional applications, there is only a copy of these attributes which are shared by instances of the same class. In a standard Java Virtual Machine (JVM), the attribute sharing only works within the scope of a JVM instance/process. Consequently, this presents an issue in distributed applications, in which *static fields* are *not* shared between different machines since each machine loads a separate copy of the class, therefore the value of these fields are not consistent across machines.

J-Orchestra addressed this issue by generating an additional proxy class *SP* to manage the synchronisation of the static fields of a class. Any attempt to access a static field will be delegated to an *SP* instance to synchronise the values of the distributed static fields. Additionally, this approach also provides the flexibility of implementing extra functionality in *SP*. Since the issue of static field synchronisation has been appropriately addressed in the work, it will not be discussed any further.

2.6 Private Methods

Generally, proxies are used for inter-object (i.e. between methods of different objects/instances) rather than for intra-object communication which is more appropriately addressed using an Aspect Oriented Programming (AOP) technology such as AspectJ. Nevertheless, *private* methods might need to be proxied since even though accessing these methods is restricted to code residing in the same class, it is possible that the attributes are accessed by a different *instance* (of the same class), thus this case is qualified as inter-object communication.

Proxying a private method requires the method to be exposed (e.g. making it public) so that it is accessible from the proxy. However, doing this could change the semantics of the method declaration. For example, if a class *x* has a child class *y*, exposing a private method of *x* could cause the method to be unexpectedly overridden by the child class *y* if a method with the same name exists in *y*. This semantic is inconsistent with that of the original class because private methods are not supposed to be polymorphic, i.e. they cannot be overridden.

This issue was addressed by Eugster (Eugster, 2006) by introducing a new non-private method (called *stub method*) which simply delegates/forwards to the original private method. The stub method is given a unique name, e.g. prefixed with the *qualified class name* (i.e. package + class name), so that it does not conflict/override a method of the parent class. Then, the original private method *m()* is proxied according to the following call sequence: `XProxy.m()`, `XImpl.X_m_stub()`, `XImpl.m()`. However as will be discussed in section 3.2.3, this solution needs to be extended in order to cope with the issues introduced by proxy inheritance and parent method proxying.

2.7 Self-referencing

A common issue in any proxy-based approach is that passing a self-reference (i.e. using “*this*” keyword) of a

proxied object to another object, will expose the proxied object, thus undesirably allowing direct access to the object instead of via a proxy. This could break the semantic of the object interaction since the functionality supported by the proxy would not get executed in subsequent inter-object communication anymore.

In EJB (Sun Microsystems, 1994-2008b), this issue is addressed by requiring the developer to *manually* acquire the handle/proxy of a particular EJB component via the relevant API method, prior to passing/returning the proxy to other components. In contrast, the Java RMI framework (Sun Microsystems, 1994-2008c) *automatically* replaces a remote object with a stub/proxy when the object gets passed or returned from a remote machine. This approach is transparent to the developer however it has the shortcoming in that the automatic object substitution is only performed during remote invocation, which although is appropriate for *remote* communication, might not be appropriate for other (i.e. *local*) proxy-based functionality.

J-Orchestra proposed that code transformation was used to substitute the “`this`” reference with the relevant proxy whenever the reference is explicitly used. However, there was little discussion on the substitution rule, i.e. conditions that need to be satisfied for an expression to be substituted. Consequently, the substitution rule and conditions will be discussed in section 3.3.1. Moreover, an extension to the original substitution technique that allows self-references to be correctly passed/returned from any class in the inheritance hierarchy of the implementation class will also be presented.

2.8 Identity Semantics

Cases that can cause issues related to the identity semantics of a proxy object were identified in (Pratikakis et al., 2004) as follows: checking reference equality using “`==`” operator, synchronising thread executions, and testing an object type via “`instanceof`” keyword.

The first two cases only cause a problem when there are multiple proxy instances referring to a single object, since even though the proxies represent the same implementation object, in actuality they are not the same object. In some systems, having multiple proxies for a single object is inevitable due to the specific functionality provided by the proxy. For example, in the case where a proxy is used to keep track of the interaction (e.g. call frequency, duration) between two objects, there should be at least one proxy instance for each caller-callee pair. Similarly, in a distributed system, there could be multiple proxies referring to a remote object from different hosts.

Although accessing the attributes of a proxy exhibits the same behaviour as in the case of the original application, i.e. accessing the original/target object, a different result will be returned when two proxy references are tested for equality, e.g. using `==` operator. Consequently a solution as well as the required code transformation will be discussed in section 3.4.3.

Synchronising multiple execution threads on a proxied object might not work if the synchronisation is performed on different proxy instances. This issue has been addressed in (Tilevich and Smaragdakis, 2002) and therefore a further discussion is beyond the scope of this paper.

Another identity-related issue is the use of the

“`instanceof`” operator to check whether a certain object is of a certain type. A transparent proxy approach should maintain type compatibility between the proxy class and the original class so that the use of “`instanceof`” keyword returns correct results.

3 Addressing Proxy Transparency

This section presents several solutions involving code transformation for addressing the transparency issues discussed in section 2 as well as other issues arising from the solutions. Although some code transformation tools/technologies are more suitable for the proposed transformation tasks than others, it is beyond the scope of this paper to provide an exhaustive evaluation; nonetheless obvious advantages and limitations of certain technologies will be mentioned in section 3.2.

This section firstly describes the overall structure of the proposed proxy approaches then follows with some discussions on specific aspects of the transparency solution, which are classified and structured according to the classes that are transformed: the *proxy* class, the *original* class, and the *proxy clients* (i.e. classes that use the proxy). The configurability aspect will also be addressed in order to enable the application of domain-specific knowledge to reduce the overhead that results from the proposed transparency solution. In reality the application deployer should have a reasonable amount of knowledge on the deployed application.

Since a common application of proxies is to provide middleware supported functionality (e.g. remote components in an application server (Sun Microsystems, 1994-2008b)), the proposed proxy approach is designed with such application in mind. As such, where necessary the supporting framework or middleware is discussed.

3.1 Proposed Class Structure

In supporting a proxy class hierarchy that is compatible with the original/proxied class, this section proposes three alternative class structures, each of which is based on an existing approach.

The *first approach* improves upon the approach adopted in JavaParty (Philippsen and Zenger, 1997) and J-Orchestra (Tilevich and Smaragdakis, 2002), in which the generated *proxy* class is named after the *original* class `X`, whereas the *implementation* class is named `XImpl`. In the case where a proxied class `B` extends another proxied class `A`, the proxy class `B` extends the parent proxy `A` whereas the original class `BImpl` extends `AImpl` as illustrated in Fig 1. The main difference between this approach and the one adopted in JavaParty and J-Orchestra is that a proxy class does not extend a class other than the one extended by the original class, thus class type compatibility can be maintained. This approach has the advantage of requiring simple code transformation, in particular when compared to the *third alternative*, which will be discussed in detail in section 3.2. However, the main limitation of this approach is that it does *not* support dynamic and transparent wrapping/unwrapping of proxies, i.e. dynamically swap a proxy with the implementation object and vice versa, without breaking existing code. This is because there is no common type between the proxy and the implementation classes. Note that the wrapping and unwrapping

functionality referred here is dynamic, i.e. can be performed at any point in the application execution, and thus is different from the static wrapping/unwrapping feature discussed in (Tilevich and Smaragdakis, 2002).

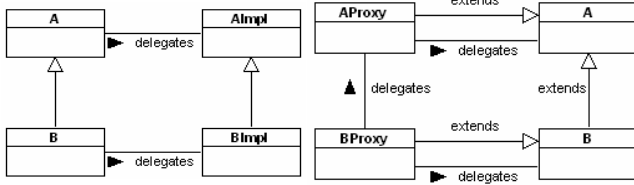


Fig 1. An Example of Approach 1 Fig 2. An Example of Approach 2

Depending on the application/utilisation of the proxies, wrapping/unwrapping might or might not be required. For example, certain client/source objects may need to interact with a target object directly (i.e. without going through the proxy) due to various reasons including: 1) improving performance and efficiency, 2) enforcing semantic correctness by preventing the execution of the proxy functionality which could undesirably modify certain states/data, and 3) addressing the issue caused by unmodifiable client code (e.g. native code) (Tatsubori et al., 2001, Tilevich and Smaragdakis, 2002), i.e. the inability to apply solutions that require modification of the client code, such as proxying field access (section 2.3). Determining when to wrap/unwrap is beyond the scope of this paper since it is specific to the supported proxy functionality. In comparison to the first approach, the wrapping/unwrapping capability is supported in the second and third approaches.

The *second approach* extends the class structure proposed by Eugster (Eugster, 2006) to explicitly address the main issues in proxy inheritance. In this approach, the implementation class x is named after the original class x , while the proxy class $xProxy$ extends the implementation class x for *type compatibility*. In a normal inheritance scenario (depicted in Fig 2), the child proxy $BProxy$ extends the implementation class B , which then extends the parent implementation class A . As such, $BProxy$ is compatible with both implementation classes (i.e. A and B), whereas $AProxy$ is compatible with class A because it extends A . In order for $BProxy$ to proxy/handle the methods of its parent implementation class A ; it declares the parent methods and delegates the execution to $AProxy$.

Note that even though the proxy class $BProxy$ can directly access the functionality implemented in the B via inheritance, in practice, the proxy does *not* execute the functionality/methods but rather it overrides the methods which then forward/delegate the execution to a separate instance of B (i.e. *implementation object*). Otherwise, if $BProxy$ simply uses the inherited functionality without delegating to an instance of B , the solution could degenerate into a decorator design pattern (Gamma et al., 1995). In this case, the role of the $BProxy$ is to decorate (i.e. extend the functionality of) the implementation class B , which is not what this paper aims to address.

The drawback of this approach is that since $BProxy$ extends B , it inherits all the fields declared in B , which are never used because as previously mentioned the execution of the functionality is delegated to a separate instance. Even though precautionary steps can be taken to ensure that the fields will never get initialised, they still consume

a certain amount of memory. Another downside of this approach is that since the proxy automatically inherits the capabilities of the implementation class (e.g. remote communication capability that is achieved by specifying `Remote` as one of the implemented interfaces), there could be confusion or uncertainty over whether the proxy really requires/uses the capability. In the best case, this only reduces code readability and thus complicates development and maintenance of the application/framework. In other cases, this could introduce overheads or even change the semantics of the original application.

The *third approach* is based upon the solution proposed in (Ryan and Westhorpe, 2004), with the primary difference being an abstract *class* is used as the *domain type* instead of a Java *interface* (Fig 3 and Fig 4). This allows the domain class to extend the original parent class to maintain type compatibility which was not possible using an interface as was mentioned in section 2.1. Note that the use of an abstract class for the domain type is *not* novel in itself, but the utilisation of such an approach to accommodate the compatibility between the proxy class and the original class is non-trivial and has not been explicitly addressed in previous work. This paper favours the use of the third alternative since it allows wrapping and unwrapping, thus supporting a wider range of proxy functionality than the first approach, with less overhead than the second approach as discussed in section 4.2.3.

The issues related to the *structural compatibility* of both the proxy class and the original/implementation class in the *third approach* will be addressed in section 3.2 and 3.3. *Structural compatibility* refers to the equivalence between the structure of the proxy class and the original class, which include class attributes (e.g. fields, methods), attribute modifiers (e.g. protected, final), and parent type hierarchy (i.e. super class and interfaces). Additionally, sections 3.3/3.4 will address the *semantic compatibility* of the proposed approach in order to ensure that the use of proxies does not change the intended behaviour of the application. Although focusing on the *third approach*, some solutions such as those discussed in sections 3.2.3, 3.3.1, 3.3.2, 3.4.2, and 3.4.3 are generic and as such are applicable for the *first* and *second approaches* as well.

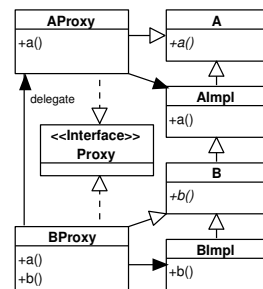


Fig 3. Approach 3: extending another proxied class A

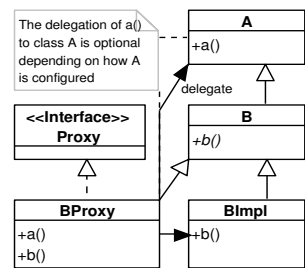


Fig 4. Approach 3: extending a non-proxied class A

3.2 Proxy Class Generation

This section discusses aspects of the adopted approach (i.e. the *third approach*) that influence the structural properties of a proxy class, which as a result might require changes on the *domain* interface and the *parent* classes (i.e. all classes that are directly or indirectly extended by the original class). However as will be revealed in this section, the modification/transformation of the parent classes is

minimal and more importantly does not break dependency with other existing classes. Although it is not the aim of this paper to evaluate different transformation tools and technologies, it is worth noting that the parent classes being transformed might *not* belong to the application (i.e. belong to external libraries), and thus it is often more practical to transform these classes at byte code level as the source code might *not* be easily accessible or even available. Also note that since Java does *not* allow transformation of *system class* (i.e. core Java class) byte code at runtime (more specifically at class-load-time), the transformation has to be performed at post-compile time.

3.2.1 Proxy Class Inheritance

The following addresses the issues in proxy inheritance, some of which have been described in section 2.1. These issues include proxying the attributes of the parent classes, extending external classes, and preventing unnecessary parent class initialisation. As mentioned in section 3.1, in the adopted/third approach, both the implementation and proxy classes extend the domain class, which further extends the appropriate parent of the original class. As shown in Fig 3 and Fig 4 (in section 3.1), the proxy class implements the `Proxy` interface so that they can be distinguished from the implementation objects and can be accessed in a polymorphic way, e.g. by the supporting framework. Note that common proxy/extra functionality, e.g. caching, is delegated to a `ProxyDelegate` instance.

In the case where an ordinary (non-proxied) class `B` extends a proxied class `A`, the child class `B` should be modified to extend the *implementation* class `AImpl` of the parent class. However note that although this scenario involves a proxy as well as class inheritance, it is not related to the inheritance hierarchy of a proxy class. On the other hand, there are two fundamental proxy inheritance cases that should be considered: a proxied class extending another proxied class (Fig 3) and a proxied class extending a non-proxied class (e.g. original class) (Fig 4).

In the former case, the domain class (`B`) needs to extend the implementation class of the parent (`AImpl`) as shown in Fig 3. Note that the solution is also applied to the parent class `A`, therefore `AImpl` and `AProxy` also extend `A`. Since the child proxy (`BProxy`) initially only forwards calls to methods of the child class (`BImpl`), the proxy needs to be extended to further allow access to methods of the parent implementation class (`AImpl`). This could be done by inserting the support into the child proxy but this could result in a lot of duplication. Having the child proxy (`BProxy`) inherit from the parent proxy (`AProxy`) is also not an option since `BProxy` already extends another class. As such, a delegation approach is used; wherein *overridden* methods are delegated to the corresponding methods of the parent proxy i.e. `BProxy` delegates the methods of domain class `A` to `AProxy`.

The second inheritance scenario, in which a proxied class extends a non-proxied class, is generally applied when the parent class belongs to an *external* library since structurally modifying it could affect other dependent classes. In this scenario, the domain class (`B`) first extends the original parent class (`A`) as shown in Fig 4. However since the parent class is unmodified and does not have a proxy, the delegation solution used in the former case

cannot be used. Instead, if any of the parent methods need to be accessible from the child proxy; these methods should be inserted into the child proxy (`BProxy`).

With regard to one of the identity issues described in section 2.8, since the proxy class extends from the domain class which is named after the original class, the use of the “instanceof” operator in the original code can be left unmodified since the relevant expression will return the same result as in the case of the original code.

3.2.2 Parent Constructor and Initialisation

In the two inheritance cases, the *domain* class should have a matching constructor declaration for each constructor of the original *parent* class, i.e. with exactly the same signature/arguments. These constructors are needed to forward calls from the constructors of the *implementation* class (i.e. via the “super” keyword) to the corresponding constructors of the *parent*.

Similarly, an additional constructor should also be inserted into the *parent classes*, i.e. domain class and all other classes up in the inheritance hierarchy. This is based on a solution proposed in (Eugster, 2006) whereby the inserted constructor receives a single argument with a unique class type, e.g. `myframework.CreationInfo` to ensure that its signature does not conflict with any of the existing constructors. The purpose of introducing a new constructor in the *parent classes* is to allow the proxy class to *explicitly* invoke the new constructor instead of relying on the *default* Java behaviour, which is to invoke an empty argument list constructor of the parent class, which might *not* exist. Even if the constructor exists, the initialisations done in the constructor could have a negative impact on performance, efficiency and transparency (e.g. modifying certain data/states).

For the same reason, declaration-time field initialisations (sometimes referred as instance initialisers) should be *prevented*. This type of initialisation is similar to the initialisations done in the constructor except that they are performed earlier (i.e. prior to the constructor execution). In order to solve this issue while still maintaining the same semantics, these initialisations have to be moved to the beginning of every constructor in the class *except* for the newly inserted constructor.

3.2.3 Method Modifiers

This section discusses how methods with various modifiers (i.e. *final* and various visibility modifiers) are proxied in the proposed approach.

Any *final* (non-overridable) method of the parent class `A` that is proxied by the child proxy `BProxy` should be made *non-final*. This is because to proxy a method of the parent class `A`, the child proxy `BProxy` needs to override the method. The *final* keyword is used mainly for improving code quality (e.g. maintainability, readability) and since the code transformation works on a separate copy of the code, removing this keyword during the transformation should not affect the quality of the original code. However, doing so could potentially introduce security issues, but these can be prevented if the application never use classes from untrusted sources, which is the responsibility of the application and the supporting middleware, and thus it is beyond the scope of this work to address this issue.

In order to retain the accessibility semantics/constraints of proxied methods, the visibility modifiers (e.g. public, protected) of the original methods should be copied/reflected in the *domain*, *implementation*, and *proxy* classes. However, as mentioned in section 2.6, proxying a *private* method requires the insertion of a non-private stub method, which enables the call sequence: `XProxy.m()`, `XImpl.X_m_stub()`, `XImpl.m()`. Note that `XProxy.m()` should *not* be private to allow invocation from other objects, i.e. instance of `XImpl`. This could cause a problem when a child proxy exists (e.g. `YProxy` extends `XProxy`), since `XProxy.m()` might *accidentally* be overridden by a method of `YProxy` that happens to have the same name. To avoid this problem, the proxy method has to be renamed, thus the call sequence becomes: `XProxy.X_m_stub()`, `XImpl.X_m_stub()`, `XImpl.m()`. As a consequence, any code that invokes the original private method should be modified accordingly as will be discussed in section 3.3.2.

3.3 Implementation Class Transformation

This section discusses the modifications of the *implementation* class required to maximise transparency in the proposed proxy approaches. Unlike the proxy class, the modification of the implementation class only affects the internal implementation of the *original* class, and thus does not syntactically affect other classes.

3.3.1 Self-referencing

This section presents the rule for identifying the statements/expressions that need to be modified for the purpose of substituting a self-reference “this” of an implementation object with a proxy instance. This is so that a direct reference of the object never gets passed to other objects, which as described in section 2.7 could break the semantics of the proxied object interaction. Note that when substituting an occurrence of a “this” keyword in `AImpl`, it cannot be assumed that an instance of `AProxy` should be used as the substitution. This is because the “this” reference might refer to the child class, e.g. `BImpl`, in which case an instance `BProxy` should be used instead. As such, runtime type checking should be performed to determine the concrete type of the self-reference “this”.

A negation is used for specifying the substitution rules, which consist of a list of cases/conditions where an occurrence of the self-reference should *not* be wrapped/substituted with a proxy. As such, during the code transformation, every occurrence of the “this” keyword should be checked against the conditions, and if none of them holds, the keyword gets substituted with proxy instantiation code. The rules are as follows:

- 1) Method invocation, e.g. `this.call()`;
- 2) Field access, e.g. `this.var`;
- 3) Explicit class instance method invocation, e.g. `OuterClass.this.call()`;
- 4) Explicit class instance field access, e.g. `OuterClass.this.var`;
- 5) Implicit `toString()` call, e.g. `println("Self: " + this)`.

This negation logic serves as a safe guard where in exceptional cases (such as those when it cannot be determined whether a substitution is required), a self-reference will nonetheless be wrapped to guarantee consistent proxying behaviour. As an example of such cases, proxies are not needed when the self-reference is only passed between methods of the same object. However

it is difficult to identify this case without sophisticated code analysis, which sometimes needs to be complemented with runtime checking for more accurate substitution decisions. As such, in the absence of the code analysis and runtime checking, the self-reference will be wrapped/proxied which generally does not change the application semantic, but instead introduces small performance and memory overheads. It should be noted however that it is uncommon for properly written Java code to pass a self-reference within the same object, as the reference is always directly accessible.

3.3.2 Proxying Private Methods

As mentioned in section 3.2.3, any code that accesses a proxied private method needs to be modified to instead call the corresponding stub method. Note that since private methods are only accessible from within the declaring class, the code transformation only needs to be performed on the declaring class (i.e. implementation class). The transformation involves modifying `obj.m()` so that the “obj” reference is firstly checked whether it is a proxy as shown in Fig 5. The checking is necessary because “obj” could be a direct reference to the implementation object, e.g. due to the unwrapping capability described in section 3.1. If “obj” is a proxy, the relevant stub method should be invoked instead. In contrast, code in the parent class A is modified in a similar manner, with the difference being the stub method that gets invoked, i.e. `A_m_stub()` instead of `B_m_stub()` as shown in Fig 6.

<pre>// Orig code in class X B obj = ... obj.m(); // Transformed code (obj instanceof Proxy) ? obj.B_m_stub() : obj.m();</pre> <p style="text-align: center;">Fig 5. Private Child Method</p>	<pre>// Original code in class Y B obj = ... (A obj).m(); // Transformed code (obj instanceof Proxy) ? obj.A_m_stub() : (A obj).m();</pre> <p style="text-align: center;">Fig 6. Private Parent Method</p>
---	--

Note that a similar solution is required for the field accessors (i.e. setter and getter methods as discussed in section 2.3) of a *private field*, although in this case, the insertion of a stub method is not strictly required since the field accessor itself can serve the role of a stub method, i.e. exposing the visibility of the private field and having a unique name to avoid overriding a parent method.

3.4 Client Class Transformation

This section addresses the transparency issues that are most appropriately solved by transforming *client classes*, i.e. any *application* classes (including implementation class) that access another implementation class via the proxy. The modification only applies to the body (i.e. internal implementation not external structure) of the client class, therefore does not affect other classes.

3.4.1 Proxied Class Instantiation

This section addresses transparent proxy instantiation (as introduced in section 2.2) for the proposed proxy class inheritance structure. The transparency is achieved by modifying the instantiation of a proxied/original class (i.e. using “new” operator) so that the corresponding *implementation* and *proxy* classes are instantiated and the *proxy* instance is returned to the *client*. The transformation

involves traversing potential client classes and searching for the relevant instantiation code. The potential client classes are limited to those that belong to the application, since instantiating a class using the “new” operator requires the class name to be known prior to runtime.

On the other hand, this assumption does not apply to reflective instantiation (i.e. using Java reflection API) which allows dynamic resolving of class types, thus enabling instantiation to be performed by external classes, such as framework classes. As such, addressing reflective instantiation requires traversing all classes in the execution class path to search for classes that need to be transformed, even though only few classes require the transformation.

Furthermore, due to the supported dynamic class resolving, a static substitution/modification of code does not suffice. Instead, the solution should also involve runtime type checking. In this solution, firstly the original instantiation code should be replaced with the invocation of a newly introduced/inserted method `reflectionNew` as shown in the example in Fig 7. At runtime, the method (as depicted in Fig 8) determines whether the class to be instantiated is a domain class, in which case the corresponding implementation and proxy objects will be created and the proxy object will be returned to the client.

<pre>// Original code java.lang.Class c = c.newInstance(); // Transformed code java.lang.Class c = reflectionNew(c);</pre> <p>Fig 7. Replacing Instantiation</p>	<pre>Object reflectionNew(Class c) { if (isDomain(c)) { Class ic = getImplClass(c); return createProxy(ic.newInstance()); } return c.newInstance(); }</pre> <p>Fig 8. Proxying Created Instance</p>
--	---

One drawback of the solution which requires code modification/substitution is that there is no straight forward solution for the case where the client code is not modifiable, e.g. native code. However, it is very unlikely that a Java class is instantiated from native code, since due to it being not portable, native code is used sparsely for low-level platform-specific operations.

3.4.2 Field Access of Proxied Objects

<pre>// Original code Java.lang.reflect.Field f = ... value1 = (Integer) f.get(obj); // Transformed code value1 = (Integer) ((obj instanceof Proxy) ? ((Proxy) obj).getFieldValue(f) : f.get(obj));</pre>	<p>Fig 9. Proxying Reflective Field Access</p>
--	---

As mentioned in section 2.3, proxying field access has been addressed in previous work, the solution of which requires transformation of client classes. Due to the static nature of field access (i.e. non-polymorphic), the code transformation can be performed at compile time and most of the time only the application classes need modifying since these fields are never accessed by external classes unless done via Java reflection API. In order to handle reflective field access, the relevant statement firstly needs to be checked to determine whether it is accessing a field of a proxy, in which case the inserted `getFieldValue` method of the proxy will be invoked instead, as depicted in Fig 9. The method call is then forwarded to the corresponding field accessor as described in section 2.3.

Due to the same reason outlined in section 3.4.1, it is not possible to proxy a field that is accessed from

unmodifiable code (e.g. native code). Nevertheless, a proxy can be unwrapped (i.e. substituted with the target object) before it is passed to the native code, in which case the field of the target object will be accessed directly.

3.4.3 Reference Equality Checking

In order to address the proxy reference equality checking issue discussed in section 2.8, when such a statement is detected (Fig 10), a special method will be invoked to assist the equality checking by firstly ensuring whether the *second object* being compared is a proxy as depicted in Fig 11. If it is, both of the proxies being compared will be unwrapped so that the actual implementation objects can be tested for equality. Note that the proxy identification (i.e. the “instanceof” expression) shown in Fig 10 only works if the compared objects are not primitive values and as such a compile-time code analysis is required to determine whether the original code should be modified.

<pre>// Original code if(o1 == o2) { ... }</pre> <pre>// Transformed code if(((o1 instanceof Proxy)? ((Proxy) o1).refEq(o2) : (o1 == o2))) { ... }</pre> <p>Fig 10. Handling Comparison</p>	<pre>boolean refEq(Object o) { if(o instanceof Proxy) { return this.getImpl() == ((Proxy) o).getImpl(); } return false; }</pre> <p>Fig 11. Testing Proxy Comparison</p>
--	--

3.5 Reducing Transparency Overhead

This section addresses the configurability of the proposed proxy approaches, particularly to reduce the additional overheads that are introduced as a result of making them more transparent. Firstly, the transformation should allow the application deployer to explicitly specify potential *client* classes. Doing this improves the efficiency of the code transformation process since the code transformer only needs to inspect a smaller set of classes.

In addition, the transformation should also optionally allow the deployer to explicitly include/exclude individual methods of a proxied class in the generated proxy class for the purpose of reducing the storage overhead of both the transformation and the execution of the application. The flexibility to exclude certain methods from the proxy class is especially important if the parent class is a library class, since library classes tend to contain more methods than necessary. For instance, proxying a class that extends the `java.awt.Frame` from Java 5.0 library will generate at least 300 proxy methods, of which only a very small subset generally get used by the application. Note that depending on the functionality implemented in the proxy, more memory might be required for each proxied method, e.g. memory required for recording frequency of method calls.

4 Evaluation

The proposed solution proposed in section 3 has been applied to the development of various applications to demonstrate both the transparency and practicality of the solution, as well as providing a testbed for evaluating its: 1) correctness; 2) performance and 3) resource utilisation, as will be described in sections 4.1 and 4.2

4.1 Practical Application

The MobJeX framework (Ryan and Westhorpe, 2004) was used as a practical application of the proposed solutions, whereby the majority of the solutions were implemented in

mobjexc, a source code transformation tool specifically developed for inserting various capabilities, including those for facilitating adaptation via *object mobility* (Ryan and Westhorpe, 2004) and *object swapping* (Rubinsztein et al., 2005), into existing applications. The MobJeX framework also consists of several middleware containers for supporting the execution of adaptive applications. Adaptation via object mobility allows objects of the application to be dynamically migrated to different hosts to find the optimal object layout depending on the execution context of the application. On the other hand, adaptation via object swapping allows the application to swap a certain object with another compatible object to change the behaviour of the application in response to various factors including the availability of certain resources (e.g. memory availability) or the limitation of a certain execution environment (e.g. screen size).

Most of the transformation tasks required for the proposed solutions, which include both source-code and byte-code transformation, have been implemented, with the exception of the transformation required for the issues of proxy reference comparison, field access, private methods, and reflective instantiation. This is because it involves the complexity of requiring a certain degree of automatic code analysis, which is currently not supported by the code transformation tool (i.e. mobjexc). Arguably, the first two issues do not occur very often since firstly, in object oriented programming, logical equality checking (i.e. via `equals` method) is generally preferred over reference comparison. Similarly, directly exposing the fields of a class is not considered a good practice as it is better to use setter/getter methods which provide the flexibility to embed future functionality (e.g. value validation) into the methods. In contrast, while the last two issues are not uncommon, they are usually isolated in a specific method or class (e.g. reflective instantiation is normally handled by a factory class (Gamma et al., 1995)), thus when required, manual modification can be performed on the relevant method or class.

MobJeX has been used to deploy and run various applications, including a mobile-device compatible (e.g. PDA) application for managing photo statistics, a Taxi Dispatching System (TDS) and a prototype of a virtual world application. Since none of these applications require the unimplemented transformation tasks mentioned in the previous paragraph, the deployment of these applications do *not* require manual modification of proxy-related code. Only several lines of declarative configuration, approximately ranging between 10 – 40 lines, were required, although more configurations/fine-tuning might be required for more optimal execution.

4.2 Experimentation

Though not fully implemented, the correctness, performance, and resource usage of the *entire solution* was evaluated, with minor code modifications related to the issues described in section 4.1 performed *manually*. Although evaluating the efficiency of the transformation process could give an indication of the usability of the system and the complexity of the proposed solution; this was not done because the evaluation result will be heavily affected by the underlying design and implementation of mobjexc, which is not the focus of this paper.

The experimentation focuses on the proposed *third approach* described in section 3.1, however, a brief comparison with the *second approach* will be provided in order to further justify why the third approach is preferred. The experiments were run on a Core Duo T2400 machine running a Windows XP. Sun JVM 1.4.2_14 was used for the experiments due to the requirement imposed by the MobJeX framework to maintain compatibility with the Java versions supported by constrained mobile devices. In order to obtain more accurate memory overhead measurement, requests for garbage collection were frequently sent to the JVM.

4.2.1 Experiment 1: Correctness

The aim of this experiment is to evaluate the correctness of the solutions described in section 3. Several test applications were written to explicitly evaluate the *structural* and *semantic* correctness (as described in section 3.1) of the proposed proxy approach. These applications consist of a client accessing one or more proxied classes to test various scenarios, such as self referencing, field access, etc. The invocation of the inherited and overridden parent methods was also tested. Furthermore, the tests also cover accessing different class attributes with different modifiers. Each application differs in terms of the proxy inheritance hierarchy:

- 1) A proxied class (P) extends a non-proxied class (NP) which extends another non-proxied class (NP).
- 2) P extends P which extends NP
- 3) P extends P which extends P

Note that none of these scenarios involves a case where NP extends P, since as mentioned in section 3.2.1, it not related to the inheritance hierarchy of a proxy class. Several complex capabilities supported by the framework, such as object mobility, concurrency management, etc., were injected into the proxied classes to ensure that the insertion of the capabilities did not affect the correctness of the solutions. In addition, a separate test consisting of clients residing in different packages was conducted to more rigorously validate the visibility semantic of the proxied class. Where possible, the tests were verified automatically by using assertion statements, otherwise manual verification was done by inspecting the produced source code and tracing through the printed log statements. The results of the tests confirmed that the transformed (i.e. proxy-based) applications exhibit the same behaviour as the original versions and proxies were indeed used where they were expected to be used.

4.2.2 Experiment 2: Performance Overhead

Experiment 2 aims to measure the performance overhead that is introduced by the proposed approach, specifically when instantiating a proxy and invoking its methods. The experiment includes two inheritance cases: a proxied class (P) extends a non-proxied class (NP) and a proxied class (P) extends another proxied class (P). The proxied classes only support basic operations, i.e. store and return a single integer value, while the proxy classes were generated with no additional functionality as they simply delegated method calls to the implementation class. In order to evaluate the performance overhead of the proposed *third approach*, its performance was compared with the *traditional* (interface-based) proxy approach (described in

section 2.1). The results show a consistent trend for both inheritance cases, with the second case (i.e. P extends P) introducing slightly higher overheads. The results show that the instantiation of a proxied class (including its proxy) using the “new” keyword introduces an additional overhead of 0.002 milliseconds, which is 5% more than the time required in the traditional approach. On the other hand, the invocation of a method introduces an additional overhead of 0.0002 milliseconds, which is also 5% more than the traditional approach.

The results of the experiments involving field and reflection-based proxying cannot be compared with the *traditional* approach since the operations are not supported in the traditional approach, and instead the results are compared with the original application in order to quantify the overhead of reflection-based operations compared to the non-reflection counterparts. Firstly, field access roughly takes the same time as method invocation since it is essentially an invocation to the getter/setter of the field. In both the proposed *third approach* and the *original application*, reflective method invocations introduce equal reflection overhead; however the reflective instantiation and field access impose more overhead for the transparent approach due to the required runtime checking, with 20% more overhead for instantiation and 9% for field access.

In comparison to the second approach, the performance of the *second* and *third* approaches are roughly equal even though as shown in section 4.2.3 the second approach consumes more memory. Even when tested using a class with 50 fields, the proxy instantiation in the second approach does not introduce noticeable extra overhead.

4.2.3 Experiment 3: Storage Overhead

This experiment aims to evaluate the impact of the adopted proxy approach on both volatile (e.g. RAM) and non-volatile (e.g. disk) storage. Non-volatile storage consumption is determined by the size of the proxy, domain, and implementation classes, whereas volatile storage consumption is affected by the size of both the classes and the instances. The setup of the experiment is similar to the setup used for evaluating the performance overhead, with the exception that the client class only contains code for instantiating proxied objects since method invocation and field access are irrelevant.

The results show that the *third approach* consumes more volatile memory than the *traditional* approach in both inheritance cases (i.e. P extends NP and P extends P). The overhead was influenced only by fields in the *parent* class. As the number of (integer) fields grows, the overhead also increases, i.e. 1 field = ~4 bytes; 10 fields = ~40 bytes, 20 fields = ~80 bytes. On the other hand, the overhead of the second approach is influenced by both the fields in the *child* and the *parent* classes:

- Case 1 (i.e. P extends NP): 1 field = ~8 bytes; 10 fields: ~80 bytes, 20 fields: ~160 bytes
- Case 2 (i.e. P extends P): 1 field: ~24 bytes; 10 fields: ~136 bytes, 20 fields: ~256 bytes

In terms of class size, the *third approach* has the largest overhead followed by the *traditional approach* and then the *second approach* as shown in Table 1. As classes are only loaded *once* during the execution of an application, the class size advantage (i.e. smaller class size) of using the second approach is easily offset by the memory

consumed by the instances of the proxy. For example, in the virtual world application prototype mentioned in section 4.1, approximately 4 proxies get instantiated for each proxied class, which, in average has 12 fields.

(results in bytes)	1 Method	10 Methods	20 Methods
Traditional	6,477	8,109	9,949
Second	6,218	7,669	9,299
Third	6,841	8,455	10,275

Table 1. Total Size of Generated Classes

5 Conclusion

This paper has presented and evaluated solutions to various limitations of transparency in existing proxy-based approaches, demonstrating that transparency can be improved with acceptable overheads. Furthermore, the automation of the majority of the required code transformation has been implemented, which along with the deployment of various adaptive and test applications, demonstrates the transparency, practicality, correctness, and flexibility of the solutions. While the specific application of inserting mobile object adaptation capabilities was chosen due to its high complexity, due to the generality of the solution, it should also be applicable to other domains as will be investigated in future work.

6 References

- EUGSTER, P. (2006) Uniform proxies for Java. *Conference on Object Oriented Programming Systems Languages and Applications*. Portland, Oregon, USA ACM New York, USA.
- GAMMA, E., HELM, R., JOHNSON, R. & VLISSIDES, J. (1995) *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley.
- HOLDER, O., BEN-SHAUL, I. & GAZIT, H. (1998) System Support for Dynamic Layout of Distributed Applications. Technion - Israel Institute of Technology.
- PHILIPPSEN, M. & ZENGER, M. (1997) JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9, 1225-1242.
- PRATIKAKIS, P., SPACCO, J. & HICKS, M. (2004) Transparent proxies for java futures. *Conference on Object Oriented Programming Systems Languages and Applications*. Vancouver, BC, Canada ACM New York, NY, USA.
- RUBINSZTEJN, H., ENDLER, M. & RODRIGUES, N. (2005) A Framework for Building Customized Adaptation Proxies. *Proc. of the IFIP conference on Intelligence in Communication Systems*.
- RYAN, C. & WESTHORPE, C. (2004) Application Adaptation through Transparent and Portable Object Mobility in Java. IN TARI, Z. & MEERSMAN, R. (Eds.) *International Symposium on Distributed Objects and Applications (DOA 2004)*. Larnaca, Cyprus, Springer-Verlag.
- SUN MICROSYSTEMS (1994-2008a) Dynamic Proxy Classes URL: <http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>.
- SUN MICROSYSTEMS (1994-2008b) Enterprise JavaBeans Technology URL: <http://java.sun.com/products/ejb/>.
- SUN MICROSYSTEMS (1994-2008c) RMI URL: <http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>.
- TATSUBORI, M., SASAKI, T., CHIBA, S. & ITANO, K. (2001) A Bytecode Translator for Distributed Execution of "Legacy" Java Software *15th European Conference on Object-Oriented Programming (ECOOP 2001)*. Budapest, Hungary, Springer.
- TILEVICH, E. & SMARAGDAKIS, Y. (2002) J-Orchestra: Automatic Java Application Partitioning. *Proceedings of the 16th European Conference on Object-Oriented Programming*. Springer-Verlag London, UK.