Transparent
Runtime Evolution
of Components

Micha Trautweiler
Master Thesis
September 15, 2003 – March 14, 2004

Supervising Professor: Prof. Gustavo Alonso
Supervising Assistant: Andreas Frei
Abstract

Nowadays, we rely on computers to transfer money, control air traffic and even process emergency calls. Systems in charge of such important tasks are expected to be flawless and failsafe. However, they are neither. Runtime evolution has emerged as a solution to correct software immediately.

Existing frameworks address this issues only partially, provide limited functionality or modify the underlying software layers (e.g. the virtual machine). In contrast, we approach runtime evolution at language level (in Java) to guarantee platform independence. We use components as entities to evolve. Our goal is to provide maximum genericness and flexibility. To realize this, we use the Jadabs container. It allows to dynamically load any JAR file as a component.

Need for runtime evolution may arise unanticipated for the evolving component itself and its clients. Thus, it should be transparent to the latter and not impose any requirements on the former. We create proxy objects based on the java class loading mechanism. They hide objects from being accessed outside the component they belong to. On method invocations, return values (and arguments, too) are wrapped and unwrapped on the fly if they (arguments: do not) belong to the same component as the callee.

Mission critical system that cannot afford any downtime benefit most of runtime evolution. Safe-points are designed to support evolution of active components. They crosscut control flow at points that are suitable to migrate state. Implemented as dynamic aspects, safe-points may be defined and woven in at runtime. To remain available during migration, strategies to handle service requests may be defined and deployed (e.g. buffering or executing old version code). As implementation of a future component release is unknown to its clients, adaptation from the proxy to the upcoming component version must be done at runtime. We support for any kind of code evolution including interface modifications to let components evolve independently. Adapters specify how to map invocations to the new implementation. In addition, they customize when and how to perform an upgrade, and which state to migrate. To separate migration concerns from business logic, proxy as well as adapter components are introduced. While proxies are generated and remain stable, adapters are provided along with the new component version.

Due to double indirection, method invocations are subject to a slow down in comparison to ordinary execution. This applies for invocations across component boundaries only. Thus, usage is most advantageous for complex operations as provided by mission critical components mentioned above. They may be transparently supplanted by their successors at runtime. Neither state nor availability is lost. No anticipation from the developer is needed do enable runtime evolution of its component. As type safety is maintained and our framework runs on any Java virtual machine, usage is not restricted to special infrastructure either.
# Contents

1 Introduction .................................................. 1
   1.1 Motivation ............................................. 1
   1.2 Goal of this Thesis ..................................... 2

2 State of the art ............................................. 3
   2.1 General Issues in Runtime Evolution .................... 3
   2.2 Existing Approaches for the Java Language ............. 6

3 Transparent Runtime Evolution .............................. 9
   3.1 Concepts used .......................................... 9
   3.2 Hiding External References .............................. 12
   3.3 Evolution Initiation Conditions ........................ 20
   3.4 Code Evolution .......................................... 23
   3.5 Component Migration .................................... 27

4 Implementation Notes ....................................... 35
   4.1 Design .................................................. 35
   4.2 Problems ................................................ 42
   4.3 Samples: First Steps in Runtime Evolution ............... 45

5 Benchmarking ................................................. 49
   5.1 Experimental Setup ...................................... 49
   5.2 Configurations Compared ................................ 49
   5.3 Benchmark Cases ........................................ 50
   5.4 Results .................................................. 51

6 Conclusion and Outlook ..................................... 55
   6.1 Summary ................................................ 55
   6.2 Future Work .............................................. 57
Chapter 1

Introduction

1.1 Motivation

Software development is driven by request for modification. These changes are typically classified into four groups: Corrective, perfective, adaptive and preventive [12]. Corrective changes are modifications to existing functionality, while perfective changes introduce new functionality to the system. Adaptive maintenance provide flexibility to a system to react to changes in the execution environment, while preventive takes actions that will simplify or remove future, additional change requests.

There is a class of computer applications that must run continuously and without interruption, and yet must be changed from time to time to fix bugs and upgrade functionality. Well known examples of such applications are financial transaction processors, middleware architectures or air traffic control systems. A common solution to make such mission-critical systems run non-stop, is to design them in a special way and/or run them on a specially configured, redundant hardware. However, this makes them very expensive and hard to maintain [5].

Our vision to address this issue is to reflect the software evolution process in the running application. A change in the code should be seamlessly integrated in the system at runtime. The advantages of being able to perform corrective changes immediately and without loss of availability are apparent: The risk of functional bugs as well as security leaks is greatly reduced. Preventive as well as perfective changes in the traditional form of an installation package hold the risk of loss of state (e.g. configuration files) by overwriting existing data by accident. A runtime evolution system able to inherit the current state of the running application eliminates even manual exporting and importing of data. Adaptive changes become important in mobile environments where it may not be possible to foresee or to load all the functionality needed from startup. Runtime evolution is well suited to adapt software frequently and in unanticipated ways: The user may not even have to care about.

To illustrate the benefit of runtime evolution, imagine a security leak is detected in your web server. From the time of detection until installation of a patch, the server remains vulnerable. Moreover, it must be shut down and thus
close any connection and be unavailable during installation process. It is an awkward situation to choose between being exposed to attackers that possibly destroy your data on one side and being unavailable thus possibly lose your customers on the other side. In such a situation it would be of great value to have another alternative: Just fix the leak immediately, compile the new version, and let the runtime evolution system take care of a seamless upgrade, leaving your server available all time and become secure before anyone has had a chance to attack you.

1.2 Goal of this Thesis

The realization of a system as outlined above is far from trivial. Several questions arise influencing design decisions:

How may one find and replace obsolete code and state? Or, in terms of object-orientation, how to replace existing references to obsolete objects? In case of a web server, how may vulnerable code be deactivated and exchanged by a secure version? How to be sure that the vulnerable code is not anymore referenced?

How may one keep the state after upgrade? Keeping the web server up while patching it offers the advantage to keep the state beyond the upgrade. Client connections, sessions, caches and other internal state may remain unaffected by the runtime evolution of the server. However, some state may be lost if a thread is stopped or objects get finalized after evolution has completed.

How may one be sure not to corrupt execution while upgrading? In case of a web server, you cannot wait until it is idle before starting the upgrade. But at what point in execution can you be sure not to cancel a transaction or to replace code while it is in use?

How may a piece of software, that has been upgraded on its own, further on cooperate with software not being aware of the upgrade? Maybe an interface has to be modified to prevent malicious code to be inserted in the server. How may clients and other parts of your server communicate with an interface that was changed in an unanticipated way?

This thesis tries to address some of these questions, trade off different approaches to them and implement a runtime evolution framework prototype that imposes minimal restrictions while offering maximum flexibility. The prototype is based on the Jadabs [1] light-weight container developed at the information and communication systems group at ETH Zurich.
Chapter 2

State of the art

2.1 General Issues in Runtime Evolution

We will concentrate on work of runtime software evolution in the context of object-oriented languages running in virtual machines. There arise similar issues and solutions in most frameworks available today. Under what conditions is it secure to replace code? How to keep the state of a running application beyond evolution steps? How to find and replace references to obsolete objects? How to deal with code evolution that violates the type safety? How to evolve existing objects? In this chapter we will discuss how these challenges are met by existing systems.

Initiation conditions Before an evolution step may be initiated, one must be sure the system remains consistent replacing one piece of code by a different one (1). While systems targeting development and testing areas do not have to care about this at all (1d), for systems in a productive environment, the flawless switching is crucial. Techniques such as (1b) and (1c) require knowledge of the semantics of both code versions.

1 Finding execution points where safe evolution is guaranteed
   a) require a quiescent state (idle and no pending transactions)
   b) predefined execution points (at compile time)
   c) execution points specified at runtime
   d) arbitrary, e.g. on active methods

Reference Replacement Having found a point suitable to start evolution, the software replacing the one currently running is launched and its objects are initialized. Options in dealing with objects from different generations of a software (2) are summarized next:

Long term coexistence of different versions (2d), maybe offering a choice in what version to be used does not touch the existing system. On the other extreme, immediate replacement of all objects regardless of the application state can be found in development tools. However, global replacement as (2c)
may also take care about execution and object state. Generational update (2a) lets old objects continue to exist while newly created take new class definition. Thus, termination of the update depends on the life cycle of the objects involved. To replace each object on demand, meaning on dereferencing (2b), distributes reference updating over time. Ensuring that generational or lazy replacement does not result in a inconsistent state is very challenging.

If variables holding obsolete objects should be updated, these variables have to be detected first (3) and should be transparently replaceable. Native heap traversal and reference manipulation (3a) or language level traversal of the object graph (3b) combined with a hiding mechanism like a proxy to exchange references transparently are used to accomplish this task. Another alternative to access the heap would be through a debugger API (3c) provided by the virtual machine.

2 Updating reference policy
   a) generational
   b) on demand
   c) at once
   d) coexistence of different versions

3 Transparent replacement of obsolete objects
   a) heap manipulation in native code
   b) proxy mechanisms
   c) through a debugger interface

State Migration  Reference replacement techniques like (2b) or (2c) require to transfer state from objects to their successor. Migration of state from one object version to the next one (4) includes the question what exactly comprises the state. Migrating resources (4d) as file handle with read/write position, sockets or services outside the container as database handles is transparent if both version share the VM. However, in case the versions reside in different address spaces, state transfer even needs shared memory or other interprocess communication techniques. Migrating threads (4e) becomes much more complicated than simple member copying (4b) or object serialization (4c). As we will see, most systems today ignore the whole issue of state transfer or are restricted to member variable copying.

4 Transferring state between versions
   a) persistent state only
   b) deep copy by means of reflection
   c) object serialization and deserialization techniques
   d) include sessions and local resources
   e) thread migration including call stack, local variables and program counter
2.1. **GENERAL ISSUES IN RUNTIME EVOLUTION**

**Code Evolution** In software development, code is subject to continuous change. Even lacking the guarantee that the interface or API of one release remains compatible with the previous one, it is much more likely that the implementation (a component or class) does not undergo refactoring or structural changes (5) that turn it incompatible with its predecessor.

Thus, software evolution may have serious impact on type safety: The interface (5f) and implementation (5ab) may be affected. Methods can be added, deleted, or modified (5d, signature and body), and fields may be renamed, re-typed, added or deleted. Furthermore, the type itself can change (5e): Adding or removing superclass or interfaces effectively changes the set of types that an instance can be cast or assigned to, with potential effects on any variables bound to such an object.

5 Code evolution support

- a) implementation only (method bodies)
- b) changing internal method signatures and static types of members
- c) binary compatible changes
- d) adding and deleting methods and fields
- e) structural changes in the class hierarchy (superclass, implemented interfaces)
- f) arbitrary changes including interface adaptability

The resulting systems discussed below differ with respect to their special requirements to code (6) subject to runtime evolution or platform of the framework (7), (type) security and efficiency.

6 Requirements to the user code

- a) need for anticipation by the developer
- b) relies on a coding convention
- c) transparent to the user code (works for legacy code)

7 Special requirements to the infrastructure

- a) VM implementation version
- b) platform dependency: operating system and hardware
- c) run in a special execution mode (debug)
- d) conform to a container model
2.2 Existing Approaches for the Java Language

Although some object-oriented languages [15] support dynamic typing at runtime and thus provide maximum flexibility for code evolution, type checking adds a significant overhead to the runtime system. The widespread use of Java, the deployment of its container model for critical applications that would benefit most from runtime evolution, its platform independence and VM based implementation makes it most interesting for investigation runtime evolution techniques. A selection of different approaches using Java is given below. Their characteristics are classified according to the criteria presented and their limitations and benefits are summarized.

Class File Replacement

In the course of the HotSwap research project at SUN microsystems [5], the Hotspot VM has been extended with the ability to substitute modified code in a running application through the debugger API. In its current implementation, it is most useful for interactive debugging purposes. While debugging, the developer identifies a problem, fixes it, and continues to debug with the fixed code, all without having to restart the application. For example, one can edit and recompile a single class and replace the old class instance with the new instance. Future invocations in the current debugging session will execute the new code. If the currently executing method is subject to change, it is re-executed using its new implementation (popping all frames above the lowest that has been redefined).

Concerning the issues presented above, the approach taken by SUN may be categorized as instantaneous replacement (1d) restricted to method bodies (5a). This kind of pure code evolution is transparent to the code (6c). Neither sophisticated reference replacement nor state transfer is needed because of the limited functionality. In addition, the target VM is required to run in debug mode (7c) as SUN implemented the functionality in the Virtual Machine Debugger Interface (JVMDI) layer. The hotswap debugging API is included in the J2SE hotspot VM since version 1.4 (7a) and used by many Java IDEs today. Nevertheless, it cannot be counted as a software evolution framework because it does not yet address object evolution at all (2,3,4).

inxar hotswap

Another approach has been made by the hotswap library from inxar (information exchange architecture) [10]: Evolution is achieved through recompilation, dynamic class reloading, and object state migration throughout the life of an application. Source code modifications lead to automatic recompilation and on each method invocation\(^1\), the need of another evolution step is checked. Just checking alone is very time consuming. Additionally, an evolution step is encapsulated by a transaction and committed in a two phase commit (2PC) protocol.

\(^1\) on explicit calls of \texttt{hotswap()} only in another implementation version
Altogether, this concept has more in common with a developer tool rather than an evolution framework.

The objects subject to hotswapping are never directly referenced by the application program, but hidden behind a proxy (3b). Thus, the proxy as well as the hidden object must implement a common interface that the client application references. Furthermore, the developer has to anticipate (6a) which classes should be able to evolve. Object state migration is done by global (2c) recreation of all new objects and simple field copying from the old ones (4b). There seems to be no support for sophisticated code evolution beyond (5a).

Dynamic Java Classes

S. Malabara et al [14] propose an evolution system that extends the JVM by a dynamic class loader that supports replacement of class definitions. In contrast to user defined loaders, this one requires minor changes inside the VM itself (9a): Namely the Just In Time compiler (JIT) and method in-lining are disabled and some data structures have been extended.

```java
public class DynamicClassLoader extends ClassLoader {
    public Class reloadClass(String newc);
    public final int replaceClass(String oldc, Class newc);
}
```

Listing 2.1: Class definition replacement interface

The authors main concern is to guarantee type safety and remain consistent to the Java linking mechanisms (e.g. byte code verification). On this account, they define a dynamic class change as valid if (A) no class depends on a field or method removed and (B) if a class depends on the hierarchy of another class c (recursive superclass and interfaces), c’s hierarchy is not allowed to be modified. These limitations prohibit that an object gets incompatible after evolution with the type of the variable that already stored it. In practice, this means that just code may be replaced that is not in use (5a) and limited forms up to (5c). It excludes active methods, too.

Not only code but state evolution is supported: During global object update, all other threads are blocked (2c). Finding and updating objects on the heap is implemented in a mark and sweep manner: In many JVM implementations, including JDK 1.2, the heap is divided into a handle pool and an object pool. Java objects are always addressed indirectly through their handles. The use of handles facilitates garbage collection. When an object is moved, only the pointer in its corresponding handle needs to be updated; the handles never move (3a). This model is very useful when handling object update for dynamic classes since one may allocate new space for an object when updating it, without changing the handle used to reference the object (6c). Of course, this manipulation may be done in native code only.

In their approach, the authors do not address the question under what conditions a class definition may be securely replaced. There is no kind of
safe-point mechanism built in (1).

**JDrums**

The Java Distributed Run-time Update Management System [2] allows the introduction of new versions of existing Java classes on the fly while preserving the internal state of objects. Conceptually, the updating is transparent to the user of the Java application (6c). The implementation is based on the Sun JDK 1.2 virtual machine, where the internal representation of classes and objects is enhanced (9a) by a pointer to a conversion class. On dereferencing, the class or object is migrated if this pointer is not null. This leads to a lazy evolution where code and data are updated on demand (4b). Although this approach distributes the costly object state migration over time, it causes serious consistency issues. Further limitations include no state migration of superclass and inner class objects using reflection (4b).

JDrums provides sophisticated code evolution support: Conversion and mapping utilities that rely on heuristics to automate state transfer even in case of renamed and retyped member variables, method migration being planned (5e).

To control evolution initialization, the developer may predefine safe suspend/resume points in the code at compile time (1c). At the time the execution reaches one of these, an online version upgrade may be initiated by replacing the class definition.

**Dynamic Component UPdating**

As a part of their component model SOFA [17], the authors have developed a dynamic component updating extension (DCUP). Similar to CORBA [16], SOFA offers a distributed component environment. Applications are viewed as a hierarchy of nested components that specify their interfaces in a custom interface definition language (IDL). At deployment time, wrappers are generated (by an IDL compiler) that form the permanent and implementation independent part of a component. Components contain application logic only, while connectors (also generated from a specification) implement the necessary interaction semantics and cover deployment dependent detail.

The non-permanent (replaceable) part of a component may be exchanged at runtime. Before an update of a component may be started, it must not have any active threads (1a). The state of the replaceable part running is externalized (4c) and loaded by another component that conforms to the specifications in the IDL (5c). Because externalization of state seems not to be implemented yet (planned as 3b), DCUP may be categorized as a software distribution tool performing upgrades transparently (6c) and automatically. However, a SOFA component (7d) is unavailable during the upgrade, thus limiting transparent and runtime evolution. Nevertheless, from the design point of view, it seems to be one of the most comprehensive solution available today.
Chapter 3

Transparent Runtime Evolution

As we have seen in the previous chapter, runtime software upgrade systems in Java share many issues that originate from lacking functionality of the language or runtime system. This results in trade offs leading to limited functionality, partial solutions and systems targeting special application domains only. Thats because we have decided to develop a runtime component evolution framework that tries to address as much issues as possible found in a productive environment:

Most important, an evolution framework should not impose any requirements on software, and thus enable runtime evolution for legacy code. One can not expect a programmer to rewrite its code just for compliance to the framework, adding another version to be maintained. Second, any requirement to the platform or VM implementation version that is non-standard will turn the framework rather useless. You won’t buy a new car just to have the latest audio system built-in, will you? Third, it s crucial to guarantee a secure transition from one software version to the next one. Today, this issue is not even well understood for offline software upgrades. That’s why it is a risk for companies to upgrade their yet running systems they depend on. Fourth, the evolution framework should support any kind of code evolution found in software development, thus enabling to replace code in unanticipated ways. A system that is built with these concepts in mind is presented in-depth in this chapter.

3.1 Concepts used

Before approaching the key issues, we introduce some techniques which we apply when building our system.

Class Loading in Java

The Java Virtual Machine (JVM) resolves references to a class during runtime using a mechanism called the class loader. A class loader is responsible for locating the definition of a class, which takes the form of a class file, and loading
it into the JVM. Java supports lazy loading of classes, e.g. just before instan-
tiation of the first object. According to the VM Specification ([13], chapter 5.3).

At run time, a class or interface is determined not by its name alone, 
but by a pair: its fully qualified name and its defining class loader. 
Thus, it is allowed to have several classes having the same name in a single 
JVM, assuming each of them has been loaded by a different class loader object. 
Once a class has been loaded, the VM foresees no means to modify or replace 
a class object. Even unloading of classes is an operation that is not supported 
fully by the specification (see chapter 2.17.8):

A class or interface may be unloaded if and only if its class loader 
is unreachable.

However, there exist some systems that perform such operations using byte 
code instrumentation [10] or extended class loaders [14].

Software Container Model

Separation of concerns between business logic and system has emerged models 
like Enterprise Java Beans [19] and the CORBA Component Model [16] that 
provide a runtime infrastructure and often horizontal services like persistence, 
transactions and security. The bean developer does not have to care about how 
the bean is used. Instead, a declarative way is used to add required services in 
the deployment phase before starting.

Jadabs

is a light-weight container for Java applications. Just communication services 
are built in, but other services may be loaded as needed. Jadabs managed 
entities are components. Three kinds of components are supported: Services, 
libraries and aspects. While services are active components having a main class, 
libraries are passive server components. These kind of components encapsulat-
ing business logic will be denoted bean components and their objects and classes 
beans. Aspects are not beans, but they [18] are used to dynamically adapt ex-
isting code to changing environment while adding or replacing components may 
adapt to new functionality requirements. Components have not to be built for 
Jadabs: Any JAR file containing valid Java classes may be loaded as library 
component. However, a service component has to satisfy two main conventions:

1. Implement the IComponent interface for initializing, starting and stopping 
the component.

2. No public constructors, but a factory method has to be available to access 
the components main object: IComponent createComponentMain()

In the Jadabs container, each component is loaded using its own custom class 
loader instance. After having stopped a component, this enables you to unload 
its classes from the memory.
Inter Component Dependencies

Despite its own loader, components should be able to access classes that are defined outside of it (external classes). In Jadabs, this is supported by component dependencies: Each component specifies its dependencies to others in its JAR’s manifest file (Class-Path attribute). On loading, it consults the class loader object of each dependency if the requested class was not found elsewhere. More exactly, the class loader used in Jadabs performs the following steps on a call to `loadClass(String className)`. Step 5 is executed in any case, steps 2 to 4 just on failure of the preceding step.

1. try to load from cache
2. try to load from system classes (e.g. `java.util.Vector`)
3. try to load from own component code base
4. try to load from dependency components
5. Linking (preparation, byte code verification, resolution)

Using the dependency specification, you gain control over where external classes get loaded from. As long as no class definition has been found, the loader iterates over the dependencies in declared order. This implicates that each component is required to declare all of its dependencies in the manifest: An external class won’t be found otherwise (unless it is on the system class path). The declarative way decouples the components and provides a flexible way of dynamic class loading for the container.
3.2 Hiding External References

As pointed out in section 2.1, references to external components should be detectable and editable to be able to exchange them in case of runtime evolution. Because we aim at a framework which is as general as possible, e.g. independent of platform and VM implementation, we avoid manipulation in layers below language level. A common way to hide objects from direct access, is a proxy mechanism combined with a registry. A registry that keeps track of every proxy instance for obsolete reference detection is easy to realize: Just let each proxy perform its registration at instantiation time (e.g. in the body of its constructor(s)).

But how to implement a proxy that suits our needs? Most implementations rely on an interface and thus force the component programmer to declare its variables this way. However, such a coding convention requires extra effort from the developer and conflicts with our intention to support unanticipated evolution for legacy code. Furthermore, to be fully independent of the implementing class, instantiation of these proxies should be encapsulated inside a factory that returns an object of a class implementing a known interface. The dynamic type of the object returned remains unknown to the caller, but the factory guarantees it to conform to the public interface. A proxy that requires this kind of coding convention will be denoted as non-transparent.

Existing Proxy Mechanisms

The Java Dynamic Proxy API

Since version 1.3 of its J2SE [20] implementation, Sun microsystems provides an application programming interface (API) [21] that lets you generate a proxy class given a list of interfaces as shown in listing 3.1. The proxy class generated extends java.lang.reflect.Proxy and implements all methods in the given interfaces. At runtime, it delegates invocations of these methods to the InvocationHandler object associated to the proxy (e.g. the hidden object itself). In the user code, one may declare instances of this dynamic proxy as any interface type implemented by the proxy class (in fact, one has no choice since the name of the proxy class is unspecified).

```
InvocationHandler hiddenObj = new MyInvocationHandler();
3 IamAnIfc dynProxy = (IamAnIfc) Proxy.newProxyInstance(
4    classloader,
5    new Class [] {IamAnIfc, MeToo},
6    hiddenObj
7 );
8
dynProxy.anyMethodOfAnyInterface();
```

Listing 3.1: Dynamic proxy creation

Being suitable in most cases, the Java dynamic proxy API does not overcome the limitations mentioned above. Extending the Proxy class, a dynamic
3.2. HIDING EXTERNAL REFERENCES

proxy can neither be assignable from nor be compatible with the object it hides and is thus non-transparent. Its usage is restricted to variables or parameters typed as an interface it implements. This is not much a problem as long as you use it in your own code. However, in system-defined classes, you cannot replace signatures of methods such as `renameTo(File)` or constructors as `new FileInputStream(File)` or change the static type of members or local variables (e.g. `private File f;`). In general, you are not even allowed to change code that does not belong to your component, including its dependencies. Thus, you need to cast your proxy object to the corresponding static type of the given parameter (e.g. `File`) or field. Although the hidden external class may extend this class (class `JavaFile extends File`), its interface will not. Say the external class is:

```
package externalCop;
public class JavaFile extends File
    implements InvocationHandler {
    public void renameTo (IFile f) { /* ... */ }
    public URL toURL () { /* ... */ }
    public Object invoke( Object callee,
        Method invoked,
        Object[] args ) { /* ... */ }
}
```

Listing 3.2: Extending a library class

You must supply an interface that declares all public methods that you want to offer to clients and the method inherited from its superclass `File`:

```
package externalCop;
interface IFile {
    public void renameTo (IFile);
    public URL toURL ();
    // public ...
}
```

Listing 3.3: Declaring a superinterface

then you generate the dynamic proxy class (and an instance of it) and perform operations defined in `IFile`:

```
IFile dynProxy = (IFile) Proxy.newProxyInstance(
    System.getSystemClassLoader(),
    new Class [] {IFile, Comparable, Serializable},
    new JavaFile()
);
((Comparable)dynProxy).compareTo(new File('.'));
dynProxy.toURL(); // works fine
```

Listing 3.4: Creating a dynamic proxy

The last argument to `newProxyInstance` is an `InvocationHandler` instance, in this example the hidden object itself. While you may call methods defined in any interface implemented by the proxy, trying to pass a proxy object
as an argument to a method expecting an instance of the hidden object’s type, JavaFile in this case, fails at compile time. Trying to cast the dynamic proxy to File compiles, but will lead to a ClassCastException at runtime.

```
1 File f = new File(‘.’);
2 // following lines cause errors
3 f.renameTo(dynProxy);
4 dynProxy.renameTo(f);
5 FileInputStream in = new FileInputStream(dynProxy);
6 File obj = (File) Proxy.newProxyInstance( /* ... */);
```

Listing 3.5: Limitations of the dynamic proxy API

Even though File is assignable from JavaFile, it’s not from IFile. Declaring the proxy instance as File does not work either\(^1\). Thus, Java dynamic proxies are not our first choice because they are non-transparent and do not allow interface adaptability.

**Aspect Oriented Programming**

Another alternative proxy implementation is found in the context of Aspect oriented programming (AOP) \([6]\): You may intercept the control flow of an application and perform additional or different operations at these points. In terms of AOP, an aspect is woven into the code that executes its advice code at joinpoints matching the crosscut definition. An implementation of such an aspect that serves as a proxy for any File object is given in listing 3.6. The code is written using the language AspectJ \([24]\) that is an AOP extension for Java.

```
1 public aspect proxy4Files {
2     Object around (): call (public * File.* (..)) {
3         return proceed ();
4     }
5 }
```

Listing 3.6: A wrapping aspect

The keyword aspect declares the following code as to be woven into the ordinary object-oriented code at compile time. The aspect proxy4Files declares one method, the so-called advice. It is executed whenever the conditions stated in line 3 are satisfied: That is, the method invoked may have arbitrary name and parameter types, but must be declared by the class File. An aspect may declare different kinds of advices: The around advice shown here will lead to a replacement of methods matching the conditions declared by the code in the body of the advice. The special keyword proceed means that the replaced method is executed.

A disadvantage of this elegant approach is the need to extend the Java language and to be limited to load-time weaving of the aspect into the original

\(^1\) Note that changing the signature of the method IFile.renameTo(IFile) to renameTo(File) would be inconsistent, but line 4 in listing 3.5 compiles that way.
code. The latter may easily result in too many proxies since it is undecidable if a (public) object will ever be used outside the component its class belongs to.

PROSE seems to be a very promising approach that overcomes both limitations: Based on the AOP concept, it goes beyond AspectJ: Aspects are ordinary Java classes, PROSE [18] runs on top of any standard JVM and allows weaving aspects in (and out) at runtime. Unfortunately, the implementation of around crosscuts in PROSE had not been completed at the time of writing, but the aspect should look similar to the AspectJ code shown above.

**Proxy by Inheritance**

Our first approach based on pure Java was based on the idea to extend the class to be hidden and overwrite all of its public methods.

To achieve transparency, each constructor invocation of an external class `extClass` in the code of a class `clientClass` should be replaced by a factory method returning an instance of the corresponding proxy class `proxyClass`. Using the Javassist library[4], this could be done easily (see listing 3.7).

```java
1 CodeConverter conv = new CodeConverter();
2 conv.replaceNew(extClass, proxyClass,"createExternal");
3 clientClass.instrument(conv);
```

**Listing 3.7:** Replacing constructor call by a factory with Javassist

Thus, any method would be called transparently on the proxy before passing it to its hidden parent object using the `super` keyword. However, this approach has several drawbacks: If fails on classes and method declared `final`. Moreover, it requires the new version class to be a superclass of the proxy, too. This clearly does not work in a single inheritance structured language like Java.

**A Transparent Proxy**

Because no proxy implementation known to the author at the time of writing is transparent and does not have any special requirements, the decision to develop such a proxy has been made. At this point, it is helpful to recall the concepts of class loading and dependency declaration as used in the Jadabs container.
Hiding Components from Clients

A benefit that results from the specification of dependencies by each component is the ability to manipulate this specification. Since a class is allowed to be loaded just once (see above), this replacement is limited at runtime: If an external class of a dependency component has not yet been loaded, it may be substituted by a different class that was unknown at compile time having the same fully qualified name (a proxy). To illustrate this, suppose you want to access an object from a component jxmeudp from within your code in eventsystem. First, you have to make sure to declare jxmeudp as a dependency of your component eventsystem. That is, add jxmeudp.jar to the Class-Path attribute of the manifest file of the eventsystem component. Now, you are ready to run both components in the Jadabs container.

```java
package org.es.client; // component "eventsystem.jar"
import com.jxmeudp.JxmeService; // external class

class EventService {
    foo() {
        JxmeService jxmeudp = new JxmeService();
        jxmeudp.bar(1);
    }
}
```

Listing 3.8: Client component eventsystem.jar

```java
package com.jxmeudp; // component "jxmeudp.jar"

public class JxmeService {
    public JxmeService() { /* ... */ }

    bar(int i) {
        // do something
    }
}
```

Listing 3.9: Dependency component jxmeudp.jar
3.2. HIDING EXTERNAL REFERENCES

```java
package com.jxmeudp; // component "proxy4_jxmeudp"

public class JxmeService {
    public JxmeService() { /* register proxy */ }

    public void bar(int i) {
        Class thisClass = this.getClass();
        Class[] params = new Class[] {Integer.TYPE};
        Method bar = thisClass.getMethod("bar", params);
        this.handler.invoke(this, bar, new Integer(i));
    }
}
```

Listing 3.10: Proxy component for jxmeudp.jar

```text
Content-Type: jadabs-cop
codebase: eventsystem.jar
Main-Class: org.es.client.EventService
Class-Path: jxmeudp.jar // dependency
```

Listing 3.11: Manifest file of eventsystem.jar

Imagine we replace the dependency jxmeudp.jar of the component eventsystem by proxy4_jxmeudp before foo() executes for the first time. The class loader of jxmeudp, after having failed to load com.jxme.JxmeService from its cache, the system classes and its own Jar file, delegates loading to its dependency components. This way, not the class named com.jxme.JxmeService in component jxmeudp that EventService has been compiled with gets loaded, but the one defined in the new dependency component proxy4_jxmeudp. This class acts as a transparent proxy.

Usage Preconditions

Loading from a different class file as described above will succeed if following conditions are satisfied:

a) The proxy component (e.g. proxy4_jxmeudp) is specified as dependency in the manifest file at load time of the using component (eventsystem)

b) The Jadabs custom class loader is used

c) The proxy class is indistinguishable from the compiled one for the loader

d) The proxy class successfully passes the verifier

e) The byte code loaded does not lead to additional linking errors (e.g. NoSuchMethodError) at runtime than the class it replaces.

Having assumed conditions a) and b), c) and d) require that

c1) the class loaded has the same name as the class requested and

c2) is defined in the same package (e.g. com.jxmeudp)
d1) no class satisfying c1) and c2) has already been loaded by the class loader object associated to the using component (eventsystem)

d2) the binary contains a valid Java class.

To guarantee e), the substituting (or proxy) class is required to

e1) inherit from the same superclass (or a superclass that is equivalent according to the criteria a) to e) )

e2) implement the same interfaces and

e3) declare the same (public) methods.

It is not a must that the replaced class (the hidden class in case of a proxy) is assignable from the proxy class as it will never be accessible to the client component (eventsystem).

Figure 3.2: Dynamic vs. transparent proxy design

Note that neither a variable intended to hold an instance of the hidden class has to be declared as an interface by the client programmer (eventsystem) nor the constructor has to be replaced by a factory method. In fact, there are no conditions or conventions the client code must meet in order to support this proxy. The client code remains untouched and the proxy is transparent to the client. Figure 3.2 illustrates the difference in design of the transparent proxy: All public methods may be invoked on the transparent proxy, not limited to methods declared in an interface (as the java dynamic proxy is). Thus, the
client programmer does not even have to anticipate that a proxy may be used in future. The proxy class may be generated at load time of a component that references an instance of an external class and is instantiated transparently at the time the client code tries to create or access the hidden object for the first time.
3.3 Evolution Initiation Conditions

Before thinking on how runtime evolution is performed, one has to be aware that it is only useful if one knows in what state the running component will be then. Kind of a barrier synchronization has to be implemented to get a consistent state to evolve from. As discussed in section 2.1, existing approaches require a quiescent state or just stop all remaining threads immediately. Both concepts are not suited for real-world application scenarios. In general, you cannot wait until a web server becomes idle - or, if you can, there is no need for runtime upgrade. Nor can you afford to pause any thread in your virtual machine as this leads to unavailability, too. Even worse, execution may continue in obsolete code after an evolution step.

The key issue to solve is finding execution points where safe evolution is guaranteed. This includes state consistency by moving from a valid state of the old version component to a valid state of the new version component. Similarly, execution flow has to be maintained by choosing the correct transition from the states of both components. To be able to identify such points in execution flow, semantics of the code has to be known. To compare the approaches outlined in 2.1, we discuss them next (the letters in braces reference to the items in the referenced enumeration):

To wait for the component to enter a quiescent state (a), a dynamic aspect that monitors activity on each proxy object could be woven into the component code at the time a component gets upgradeable (see section 4.1). An example of a PROSE aspect declaring a crosscut at every method entry and exit is shown in listing 3.12. Note that a PROSE aspect may be woven in at runtime. This is essential because it would be nearly impossible to foresee at compile time (b) where safe-points must be installed to switch to a successor component that may not yet exist.

```java
public class activityMonitorAspect extends DefaultAspect {

    private IEvolutionManager mgr;

    private Class actor; // the proxy class being monitored

    private int pendingActivitiesCount = 0;

    Crosscut beforeInvocation = new MethodCut() {
        // method entry pointcut
        protected PointCutter pointcut() {
            return Methods.before()
                      .AND ( Within.type(actor) )
                      );
        }
    }

    Crosscut afterReturn = new MethodCut() {
        // advice
        synchronized(this) {
            pendingActivitiesCount += 1;
        }
    };

    Crosscut beforeInvocation = new MethodCut() {
        // method entry pointcut
        protected PointCutter pointcut() {
            return Methods.before()
                      .AND ( Within.type(actor) )
                      );
        }
    }

    Crosscut afterReturn = new MethodCut() {
        // advice
        synchronized(this) {
            pendingActivitiesCount += 1;
        }
    };
```
Because components just get upgradeable if they are not referenced by external code, they start in a quiescent state. At the time every proxy object of a component becomes quiet again after an evolution step is requested, state may be migrated securely.

In fact, AOP is not required to implement this functionality. It could be hard coded in each proxy class generated.

A more advanced approach is to let the developer specify safe points at runtime (c) that guarantee a semantically correct transition. Execution points that are predestined to take such a role include thread exits, return from methods and loop condition violations. A dynamic aspect may be woven in that crosscuts some of these points. On writing such an aspect, one has to be aware that the more possible points of execution may be hit (joinpoints), the more likely it becomes that one of these joinpoints is reached, thus shortening the delay until evolution may begin. On the other hand, one may choose a few suitable joinpoints that impose a minimal effort in state transfer. A safe-point aspect looks similar to the one shown to monitor quiescence in listing 3.12. However, a safe-point advice does not have to maintain an internal state as a counter, but notifies the evolution manager every time it is executed. It suspends execution, saves the current state (see section 3.5) an throws an exception to pop all stack frames that become obsolete on evolution.

Allowing evolution initiation at arbitrary execution points, (d) is realized by simply start an evolution step upon detection. In fact this is a special case of (c) where no safe-points are available at all.

In all cases (a) to (d), the component loader registers a runtime evolution step at the component evolution manager as soon as it detects a newer component version. The time to start evolution is then determined by the approach chosen:

(a) After the currently executing component version becomes quiescent, the manager is notified.
CHAPTER 3. TRANSPARENT RUNTIME EVOLUTION

(c) The component evolution manager is responsible to install the safe points. On each hit of a safe point, the manager checks for outstanding hits and starts runtime evolution if none is found.

Further steps are common for all approaches. Before touching the running component, the strategy to handle incoming service requests during evolution\(^2\) is applied. Default behavior includes blocking, rejecting and buffering method invocations until evolution has completed.

\(^2\)component behavior during evolution as well as safe-points are specified by the adapter component associated with the evolution step, see section 4.1
3.4 Code Evolution

Evolution of software during development is a steady process. A release is just a snapshot of development delivering a set of functionality. As development continues, code is subject to redesign, refactoring and re-implementation aiming at more scalable, more performant code and systems easier to maintain. However, to communicate with other components, some kind of contract must be established between components, e.g. an API.

Inter Component Communication

In a transparent framework, this kind of contract between the service provider and its customers should hold beyond runtime evolution of both components.

In Java, interfaces are used to achieve implementation independence and to let components cooperate. The most intuitive way to fulfill this contract is to disable code evolution for interfaces. An interface that is loaded into the system remains stable throughout the lifetime of the component declaring it. In fact, interface contracts build the base of runtime evolution frameworks with limited functionality like 2.2. Considering non-runtime software evolution, e.g. installation packages, keeping interfaces stable sounds reasonable with respect to current life cycles around one year.

However, even APIs change from one release to the next. To maintain backward compatibility to the customers, obsolete interfaces are often marked deprecated but are still supported for another release. In spite of being useful for clients, backward compatibility often turns code hard to maintain and slow. Moreover, after backward compatibility is given up, customers need to be recompiled to fit the new interface introduced.

Decoupling Components

Decoupling clients from specific versions of their service providing components would be a gain on both sides: Clients do not need to be upgraded each time one of their service component evolves and services do not have to care about backward compatibility to each of its clients. Some approaches to decouple cooperating components includes service discovery like Jini [22] and WSDL [23], connectors [11] and work flow engines.

Service discovery promises to dynamically lookup a service as need occurs. But in reality, the service being provided must be known pretty well already at compile time: Semantics and interface signature are required, while just service location and implementation are transparent for service consumers.

Work flow engines make components cooperate that may be unaware of each other. Thus, it comes down to batch processing by output redirection to the next work flow step input. This limits collaboration since it restricts execution and data flow. For example, it is hard to build a notification service between two isolated components in a work flow system.

The concept of connectors overcomes most limitations: Consumer arguments are matched to conform the service interface and services may commu-
CHAPTER 3. TRANSPARENT RUNTIME EVOLUTION

Interface Adaptability

To allow interface modifications at runtime, the evolution framework freezes the interface of the current implementation by moving it to the proxy component. An interface in the proxy component is denoted *stable*. However, this does not implicate that any upcoming version of the component has to declare and implement that proxy interface. In contrast, the bean itself is not bound to any contract and may evolve independently. This provides maximum flexibility of code evolution, since all code of the bean becomes exchangeable. After evolution, a component may declare different packages, interfaces and classes, have a different inheritance hierarchy, declare additional methods, lacking other ones and redefining methods with a different signature. The client components using it are not even aware of the evolution.

Unstable Proxies

Allowing interface modifications at runtime has impact on the proxy mechanism. As shown in figure 3.4, for each interface declared by the bean component, a corresponding one has to be available in its proxy component. External objects may implement the stable proxy interface, not the hidden one.

A method expecting an object that implements an interface as one of its arguments, as `subscribe(IEventListener listener)` is designed to be able to accept external listener objects of types that cannot be known at compile time of the class declaring the method. The interface guarantees to the callee that any object passed as an argument implements it.

However, with interface hiding turned on (see section 4.1), listener objects implement stable proxy interfaces, not the interface declared in the hidden component. Proxy classes also take these stable interfaces as parameters to their methods: A listener object may be passed to a transparent proxy without troubles. But variables and methods of the hidden callee object expect an argument that implements the hidden interface declared by its own bean component. This situation is illustrated in figure 3.4. Thus, forwarding the invocation to the hidden object would result in a `ClassCastException`.

![Figure 3.3: Supporting interface adaptability with double indirection](image)
Therefore, a proxy implementing the current bean interface must be created to hold the external argument implementing the stable interface. Such a proxy is called *unstable*, since it depends on the version of the bean component declaring the interface hidden. On runtime evolution, unstable proxies become incompatible with the new version interface. This implicates that new unstable proxies have to be created for existing external implementers.

### Supporting Code Replacement

As described in the previous section 3.2, a proxy component is the single point of entry to its hidden bean component. The proxy guarantees interoperability of components beyond evolution of an individual component. Proxy functionality typically involves redirection of incoming invocations to the hidden object and pass back the results. To be able to do this, the proxy needs to know what method it has to call on the object it hides. This is not an issue in general as this is usually known at compile time of the proxy class.

In the context of runtime evolution, however, this is not the case: How could it be possible that a proxy (that is, of course, compiled before its first usage) know the interface or even semantics of a future version of an object it will hide? Thus, it seems reasonable to separate the reference hiding mechanism from the one that implements the message routing to the hidden object. In literature [8], the latter pattern is often referenced as an *adapter* and the proxy as a *bridge*. The proxy becomes version independent whereas the adapter remains version
CHAPTER 3. TRANSPARENT RUNTIME EVOLUTION

specific. An overview of the resulting design may be seen in figure 3.4.

Linking the Proxy with the Implementation

The adapter is the mechanism that matches the stable proxy to its current implementation bean object. Adapters are neither part of the proxy nor of the bean itself, but build their own adapter component. This way, the compatibility issue is extracted from the functionality as a separation of concerns.

```java
package ch.ethz.iks.eventsystem;

class Adapter4_EventServiceImpl implements IAdapter {

    public Object invoke(Object o, Method invoked, //...
    
    // fast: pass name of method only
    public Object invoke(Object o, String methodName, //...
    
    // even faster: pass hashCode of method only
    public Object invoke(Object callee,
            int methodCode,
            Object [] args
    ) throws Throwable {
        TestComponentMain tcm = (TestComponentMain) callee;
        switch (methodCode) {
            case METHOD_PUBLISH_EVT:  
                tcm.publish((IEvent)args[0]);
                return null;
            case: METHOD_TO_STRING:  
                return callee.toString();
            // case: ...
            default: throw new NoSuchMethodException("...");
        }
    }
}
```

Listing 3.13: A custom adapter using hash codes to identify methods

The adapter component is granted access to the hidden bean component to be able to load its classes. Adapters are tight to the version of their bean component and must be provided along. Note that this is not an additional workload for the developer - compatibility must be maintained in both cases. The difference is just where the code is inserted. An adapter is responsible for translating invocations from a proxy to the current implementation version of a hidden object. It should provide maximum flexibility on routing incoming messages to methods having a different name, different parameter types, or even to a different object of the same or of a different type. On creation of the bridge (a stable transparent proxy), its associated adapter is trivial as it just needs to forward any invocation to the implementation (see listing 3.13). Adapters to subsequent component versions may have to perform additional operations to match the implementation or interface of the hidden object if it has been subject to code evolution meanwhile.
3.5 Component Migration

In the following section, we care about how to transfer state of execution (and data) between components in runtime evolution. To be able to answer this question, one must agree upon a definition of state.

**Classification of State**

Migration of state is also a hot topic in the context of mobile agents. In addition to object states, execution migration includes program counter (pc) and threads, data migration member, call stack and resource migration are included in the definition of state (see figure 3.5) given in [7]. Although the problem of saving and restoring execution state are similar, some significant differences exist:

- Mobile agents do not have to deal with code evolution: One-to-one mapping of pc and call stack is feasible.
- Runtime evolution does not have to deal with host migration, thus simplifying local resource migration.
- Migration is *self-initiated* in mobile code whereas it is *forced* on loading of new component version in an approach as ours.

In contrast to certain agent systems, this approach does not deal with operand stack migration because execution cannot be crosscut by the safe-point mechanism asynchronously, but before or after only.

![Figure 3.5: Classification of state given in [7]](image-url)
CHAPTER 3. TRANSPARENT RUNTIME EVOLUTION

Challenges

To achieve a transparent state transfer from one component version to the next, techniques from agent migration [7] are adopted. Especially, the Java Platform Debugger Architecture (JPDA) APIs and byte code instrumentation is used. As mentioned earlier, agent migration is self-initiated and the agent is aware of migration. Code taking care of saving and restoring state is hard coded or inserted at compile time of the agent. In contrast, runtime evolution supports unanticipated forced migration. Because points in execution where evolution is allowed to begin (safe-points) may be defined as late as the new version component becomes available, code dealing with state transfer cannot be known before runtime of the previous version component.

Another difficult task is to ensure no statement gets executed twice, causing undesirable side effects in the worst case. In combination with code evolution, it grows in complexity. Source code preprocessing is used in agent implementations as [9] to guard regions of code that has already been executed while guiding execution to follow the same control flow decision that have already been taken before migration. Prior to taking over such an approach, one has to be aware that code may change in runtime evolution and thus neither call stack nor program counter may be just restored in a straightforward manner. Instead, a mapping to the corresponding call stack and location to the new version code has to be made first.

The next difference to an agent scenario becomes clear when considering threads that originate from outside the evolving component, e.g. invocations on services provided by that component. This situation does not occur on agents since they operate autonomously and all threads are started anew after migration. Approaches to these challenges are discussed in detail now.

Object State Migration

Transferring members from one version to another seems to be the easiest part of all: The built-in Java serialization mechanism performs this task.

Java Serialization

To add support for serialization of Java objects, its declaring class has to implement the Serializable interface. This may be done at load time of the class using byte code instrumentation: Because this interface is empty, it must be added to the class header only (... implements Serializable). Using Javassist to do this results in the three lines of code shown in listing 3.14:

```java
String dumpIfcName = Serializable.class.getName();
CtClass notSerialClass = ClassPool.get(nameOfClassToLoad);
notSerialClass.getClassFile().addInterface(dumpIfcName);
```

Listing 3.14: Implementing the Serializable interface by byte code instrumentation
3.5. COMPONENT MIGRATION

Note that this disqualifies components that were loaded before the evolution manager component for evolution, including legacy code. Upon hitting a safe-point as defined in section 3.3, execution is suspended. Then serialization of the object graph is initiated by passing the root object(s) of the component to the `ObjectOutputStream.writeObject(Serializable)` method. However, external objects (those not belonging to the evolving component) being referenced from evolving ones do not need to be serialized, but their references assigned to the new version objects: How to detect and avoid deserialization of these if you do not have control over serialization at all? Further, how to avoid serialization of evolving objects that are not relevant for state transfer?

Java offers custom migration by specification which fields have to be dumped using the the `ObjectStreamField[] serialPersistentFields` member variable. As this variable is not declared in classes that have not been built for serialization, one can not rely on this mechanism.

Another problem with Java serialization arises when code evolution comes into play: Although bijective mapping of old to new classes is supported in Java using class identifiers `serialVersionUID`, it lacks more sophisticated code evolution support, e.g. map one old to two new classes.

**Object Graph Traversal**

Because we need to keep control over which objects get serialized while keep legacy code support, serialization has to be customizable from outside the component. On this account, we reuse the concept of adapter components mentioned in section 3.4. This time, however, an adapter component is not associated to one component, but to one evolution step. By default, no primitive value is migrated without explicit specification. If a field is relevant for state transfer according to the adapter, it is serialized if it belongs to the evolving component. Otherwise, just its reference is included in serialization (toString()) and stored in a map linking to its stable object. Starting from the entry points of the old version component, the object graph is traversed recursively, including super-objects.

On deserialization, the object graph of the new version component is constructed by instantiation of objects that correspond to serialized ones. This correspondence is defined by the adapter state mapping function. After assignment of a new version object as value of a member of another one, this state mapping is cached. Note that it is not feasible to take over the old objects because of code evolution. Instead, values of primitives are deserialized and assigned to member variables of newly created objects using reflection. Serialized references indicating external objects are replaced by this stable object. An object graph equivalent to the one of the old component is built up.

**Proxy Migration**

Special care must be taken on migration of proxy objects. Proxy objects are migrated during member migration, but proxies require some additional operations if they either hide an evolving object or are stored in a variable of an
evolving object. In particular, three cases must be considered (see figure 3.4 to remember the different kinds).

All stable proxies and unstable proxies that hide objects belonging to the component being evolved are valid beyond evolution: Just the type they hide evolves. The latter case may occur if an object is stored in a member of a dependency component (typically declared as an interface type). The type of the variable holding the unstable proxy does not evolve since it does not belong to the evolving component, but to its dependencies.

Both kinds of proxies may be treated as external references during member migration. After state migration has finished, each object of the old version corresponds to (an) object(s) of the new component version (neither injective nor subjective relation). This mapping is used to determine the successor object of the obsolete object the existing proxy hides.

In the third case, unstable proxies of a component that depends on the component evolving (e.g. a client of it) are considered. The interface this unstable proxies implement are declared in the evolving component and thus may be subject to code evolution. This implicates that these proxies get incompatible and must be migrated. A new unstable proxy is created that implements the new version interface as declared by the dependency component after evolution. The hidden object of the unstable proxy does not change, but is assigned to the new proxy: It does not change since it belongs to the client component.

Call Stack Migration

To allow transparent migration, execution has to be resumed in the new component version at the point it was suspended in the old version.

After execution is suspended, the current call stack may be retrieved through a debugger API\(^3\).

For each call frame on the stack, program counter (Location object in the JVMDI) and current values of local variables are stored. This continues until the lowest frame of a method in the evolving component (be careful in case of callbacks and recursion) is passed. The first stack frame not serialized should then be a frame of a proxy method (entry point of an external invocation) or the stack should be empty (threads started by the evolving component itself).

Having saved all relevant information, execution is forced to return immediately through all frames (e.g. by throwing an error). It is resumed at the lowest proxy frame (that catches the error), and then suspended again. In case of internal threads, the error is not caught at all and thus terminates the thread. To make sure the error thrown (ObsoleteStackFrameError) is not caught elsewhere, a dynamic aspect crosscutting catch clauses of super types of this error and re-throws the error if already handled above the proxy frame is woven in after member migration.

Having constructed the object graph of the new component, the call stack has to be restored, too. The concept is to transparently re-execute external

---

\(^3\)PROSE connects to the debugee using the JVMDI API. Unfortunately, it does not yet make accessible the method to retrieve call stack information.
invocation starting in the catch clause in the proxy object, but on the new version code. However, one must avoid executing statements which corresponding ones already have been executed in the old version code (e.g. writing files or opening connections). Thus, before re-execution begins, dynamic aspects that prevent duplicate execution on stack frames are woven in. For all frames on the call stack saved, corresponding new invocations are figured out. Again, a mapping defined by the adapter component performs this task. Note that there may be frames of external methods on top of the lowest proxy frame, e.g. in case of callbacks. These methods are mapped as identities because their code does not evolve.

Dynamic around aspects are created that is executed instead of all methods that should be on the call stack. These aspects have to declare all local variables (initialized to default) needed after return from the upper frame and take the corresponding control flow decisions (if, while, for) to reach a state corresponding to the stack saved. Also, the aspects have to restore values of local variables before passing execution to the upper stack frame. Statements after return remain the same as in the new version implementation of the methods bypassed. These aspects (or their specification) have to be provided by the adapter component to guarantee separation of concerns.

On resumption of execution, one frame at a time is executed and suspended again (in the advice body). Then the aspect weaves itself out and the one for the upcoming frame is woven in. This is required as the same method may be called multiple times, (e.g. in recursion), but may need different aspects (e.g. the topmost frame).

After the external invocation leaves the evolving component, it should have performed the operations and return the value expected by the client. In case the functionality is not available in the new version, an exception indicating a failure should be thrown.

Program Counter Migration

Because the JPDA lacks functionality to set the program counter (pc) and stack frame, it is not possible to build up a call stack at the language level without having to execute these method calls. In the Java Debugger interface (JDI), the Location object of each thread and stack frame corresponds to the pc. Mapping the pc to the location in control flow of the new version component is done implicitly by implementation of around aspects mentioned above. Remember that because code may evolve, this cannot be done using

\[ \text{StackFrame.location().codeIndex();} \]

Execution of statements stays inside the around aspects until it leaves the evolved component by returning from the lowest proxy frame. Thus, execution in the aspects must be equal to the method bypassed after having reached the topmost frame again where execution was halted and member migration was initiated. Resume is done inside the catch block of the proxy in case of externally originated threads and leads to a (hard coded) second try of execution, this time on the new version hidden object, or - more exactly - on the around
aspects imitating new version code\(^4\).

Internal threads of the obsolete component version are exited. Their corresponding threads in the new version component are started and execute the around advices leading to the same stack as the old version stack on suspension (except that aspects are on the stack instead of the methods crosscut). These around aspects may be based on the original method body enhanced by operations to construct the stack. Control flow must be guided to the reach the call stack desired, but has to remain unaffected afterwards. See listing 3.15 for an example of an advice body featuring this.

```java
int local_int = 0;
Object obj = null;
IEvolutionManager mgr = // helper variable
EvolutionManager.getManager(this);
if (evolutionMgr.isUpgrading() ) {
  local_int = mgr.loadValue(this,"local_int");
  obj = mgr.loadValue(this,"obj");
} else {
  // original execution in case flag == false
  obj = foo(local_int); // upper frame
  if (obj != null) {
    // execution after evolution
    foo(obj.hashCode());
    // ...
  }
}
```

Listing 3.15: Around advice to build up the call stack

. Instances of internal threads that are started after evolution has completed are executed on original new version code. Methods called after evolution, too (e.g. `foo(obj.hashCode())`). Eventually, all aspects are unwoven except the ones hiding infinite threads as endless loops in `main`. Because the aspects allow both normal execution and stack construction without interference, this is not a problem.

Note that migration of call stack and pc has to be done only if the component is not quiescent at the time of evolution initiation. External invocations entering the component after migration is finished as well as internal threads started afterwards execute methods implemented by the new version component.

Resource Migration

Local resources like file handles or connections are easy to deal with if a resource is handled by an object that does not belong to the evolving component, e.g. a runtime class as `File` or `Socket`: Just pass the reference to the new version object to keep state. The object will remain valid because it was loaded by the

\(^4\)In case the old component is specified to stop before the new one launches, this is not feasible.
system class loader. Thus, just its reference is serialized. However, if resources are managed by objects of the evolving component like

    class SpecialFileInputStream extends Object

that use native methods and these objects do not explicitly keep track of the resource’s state (e.g. the position in a file while reading), it becomes impossible to keep state while evolving the handle object. In Java, such a situation is unlikely to occur thanks to predefined runtime classes dealing with resources.
CHAPTER 3. TRANSPARENT RUNTIME EVOLUTION
Chapter 4

Implementation Notes

4.1 Design

An overview of the design of the framework is shown in figure 4.1:

A double indirection is used to connect clients (e.g. Client in component copWithDeps.jar) instantiating a transparent proxy (e.g. proxy4.TestComponentMain) via a custom adapter adapter2.TestComponentMain to the hidden object (TestComponentMain). Each of these three object kinds are bundled in a component: The proxy in the proxy component proxy4.testcop, the adapter2.testcop in the adapter component and the bean object in the original component testcop.jar. Note that the proxy objects are named as the original one (e.g. TestComponentMain on the left) to allow transparent replacement.

Loaded into the Jadabs container, each component is associated with two main control objects each: A resource object providing meta information on the component such as its code base (usually a JAR file), its version and a list of components it relies on (its dependencies). Second, a custom class loader that is allowed to load from its own code base or delegate loading to its dependencies, as described in section 3.1.

Evolving Jadabs

Upon activation of the central evolution manager component (that is an ordinary bean component itself), it has to evolve the Jadabs core itself to prepare it for runtime evolution of the components it manages. A loader triggering runtime evolution replaces the existing LocalComponentLoader.

The framework then checks which of the components running would be replaceable in future. This excludes namely components that are in use already by other components: A reference that has been passed across component boundaries may never become replaceable again\(^1\) because it is outside the domain of the declaring component.

Next, every upgradeable bean component’s resource (class UpgradeableComponentResource) and class loader (class UpgradeableClassLoader) ob-

\(^1\)From a user point of view. This excludes heap based techniques
Figure 4.1: Partial class diagram of the evolution framework
ject gets evolved. The class loader inherits the classes already loaded and the state of the resource object is migrated to its successor. As long as the evolution manager component is available, these bean components as well as components loaded later on gain the capability of runtime evolution.

Kinds of Components

In addition to the bean components encapsulating business logic, the evolution framework adds another two kinds of components:

A **ProxyComponentResource** contains stable transparent proxy objects to prevent access to the beans from outside their own component. These proxy components also use a different class loader than ordinary components. The proxy’s class files are not packaged in a JAR but lie inside its *proxy folder*. Because another class may be added to the proxy component anytime, it would be a waste of time packaging the JAR all new. Each proxy component loads its proxy classes from a folder named as the component it hides (without the .jar suffix) prefixed by `proxy4_`

The second kind of component added is the **AdapterComponentResource**. Binaries of adapter components are not stored in the persistent component repository (pcoprep) but in the *adapter folder* prefixed by `adapter2_` ...

An adapter component is associated with a pair of consecutive component versions representing a runtime evolution step: They allow customization and specification of the runtime upgrade to be performed.

Definitions of safe-points and state to be migrated may be provided by these components. Strategies to handle incoming invocations during upgrade and state transfer may also be included. Adapter components also contain adapter classes to link from the stable proxy to the implementation of the new component version, as explained in section 3.4.

Adapter components to initial versions of a bean component represent a special case: They must be available upon activation of either the evolution manager component or the associated bean component, whatever occurs later. They contain adapters that simply forward invocations on proxy objects to their hidden beans. These trivial adapters are planned to be auto-generated, but must be provided manually until then (in case of optimizing adapters as specified by the command line argument `-adapt hash` or `-adapt name`). As an alternative, a non optimized adapter may be used to avoid implementing these trivial adapters (`-adapt reflect`).

File System Representation

Binaries of the components shown in figure 4.1 above are mapped directly to a hierarchy on the file system. Note that the package prefixes are omitted in table 4.1 for better readability only. Running bean components are placed in the pcoprep folder in Jadabs. Jar files added get loaded and started from this repository automatically. If a Jar file has the same name as one that is already executing, the latter component is replaced by the new one. Having
the evolution manager component \texttt{evolution.jar} in the repository, a runtime evolution step is performed.

<table>
<thead>
<tr>
<th>lib</th>
<th>jadabs.jar</th>
<th>(core library)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext</td>
<td>testcop.jar</td>
<td>(components not yet loaded)</td>
</tr>
<tr>
<td></td>
<td>testcop.jar</td>
<td>(version 2)</td>
</tr>
<tr>
<td>pcoprep</td>
<td>testcop.jar</td>
<td>(Bean component repository)</td>
</tr>
<tr>
<td></td>
<td>pcopWithDeps.jar</td>
<td>(Class-Path: testcop.jar)</td>
</tr>
<tr>
<td>proxies</td>
<td>proxy4_testcop</td>
<td>(proxy components)</td>
</tr>
<tr>
<td></td>
<td>TestComponentMain.class</td>
<td>(stable proxy)</td>
</tr>
<tr>
<td>adapters</td>
<td>adapter2_testcop</td>
<td>(initial adapter for testcop)</td>
</tr>
<tr>
<td></td>
<td>Adapter2_TestComponentMain.class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>adapter2_copWithDeps</td>
<td>(adapted class)</td>
</tr>
<tr>
<td></td>
<td>Adapter2_Client.class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>adapter2_v2_testcop</td>
<td>(migration customizing adapter)</td>
</tr>
<tr>
<td></td>
<td>TestcopVersion2UpgradeStrategy.class</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TestcopVersion2Evolution.class</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Organization of component binaries in the file system

Object Wrapping

An object will be to be hidden anytime it is be passed across a component boundary. There are three main cases to differentiate: (a) A new object is created externally, (b) an object is returned from a method called on an external object and (c) an object is passed as an argument to an external object’s method. To handle these cases, the proxy factory class offers the methods shown in listing 4.1. Their signature is held in the style of the Java dynamic proxy API [21]:

```java
1 class ProxyFactory {
2     3     public static IProxy
4     5     newProxyInstance(ClassLoader loader,
6     7     Class toHide,
8     9     Object [] initArgs
10    11    ); // case (a): external instantiation
12    13     public static IProxy
14    15    newProxyInstance(ClassLoader loader,
16    17    Object toHide
18    19    ); // case (b): method return value
20    21     public static IProxy
22    23    newProxyInstance(ClassLoader loader,
24    25    Object toHide,
26    27    Object [] initArgs
28    29    ); // case (c): method argument
```


4.1. DESIGN

Listing 4.1: Transparent proxy creation API

In the first case (a), the class loader of the component ensures that a transparent proxy class is loaded from the proxy component instead of the hidden class as discussed in section 3.2. A proxy object gets instantiated and registers itself. Note that the hidden object does not yet exist. It is created upon registration using the arguments of the proxy constructor and kept within its hidden bean component.

Case (b) may occur on a method invocation on a proxy created as in case (a). If this method returns an object that is external with respect to the calling context (e.g. belongs to the same component as the callee), it needs to be wrapped behind a proxy before it is returned from the adapter. To guarantee that no external object is returned, the return value of each invocation has to be checked.

Note that in contrast to case (a), in this case the hidden object already exists before creation of the proxy.

Wrapping method arguments as in case (c) is necessary, for example, if an external object (extObj in listing 4.2) offers a method (subscribe) to pass an object implementing an interface (ListenerInterface) declared by the external object’s component.

Listing 4.2: Statements that need on the fly wrapping of objects

On the other hand, a proxy object may be passed as an argument back to its own declaring component. Thus, arguments are checked for proxies, too and unwrapped by the default adapter as needed.

Evolution using Dynamic Proxies

In contrast to the transparent proxy configuration that provides maximum flexibility, a light-weight configuration with limited functionality has been implemented. It is based on the Java dynamic proxy API discussed in section 3.2 and thus relies on the client programmer to be compliant with the coding conventions to operate correctly.

In particular, a client component must not import any class of its dependency component other than interfaces. Thus it works on interfaces only and can just invoke methods declared in these. To obtain a first reference of the dependency component, it is must use the `getExtObject` method defined in the
components resource object. Upgradeable components do not return the main object itself, but a dynamic proxy that hides it.

The dynamic proxy implementation may use the same adapters as the transparent proxy (but restricted to the reflect signature. In contrast to the transparent proxy configuration, interfaces are visible beyond component boundaries and must thus remain stable throughout the lifetime of a component.

**Command Line Arguments**

Two arguments may be passed to the evolution manager component (evolution.jar) at startup.

```
[-proxy ifc] [-adapt (reflect | name | hash)]
```

If the first argument is set (as is per default), not only Java classes but interfaces are hidden too. This enables interface adaptability but requires the transparent proxy implementation. Otherwise, interfaces of bean components remain visible to their clients. Thus, code evolution is limited but dynamic proxy usage is enabled (see previous subsection 4.1).

The second command line argument influences the way a proxy calls its adapter and may take three values:

- **reflect** using `InvocationHandler.invoke(Object callee, Method m, Object[] args)`. This way is *very* slow as it makes heavy use of reflection. However, no custom adapters need to be implemented. Invocations on the proxy are mapped by method name and argument runtime types to a method declared by the current object hidden.

- **name** using `IAdapter.invoke(Object callee, String methodName, String declaringClass, Object[] args)`. Optimized implementation that requires custom adapters. Arbitrary methods on any object may be executed to provide the functionality declared by the proxy method. This option is not applicable for dynamic proxies as the `declaringClass` argument containing the proxy class name is unknown\(^2\).

- **hash** using `IAdapter.invoke(Object callee, int methodHashCode, String declaringClass, Object[] args)`. Fastest implementation available for the transparent proxy only. The hash value computed must be unique for each method signature. This is the default option in case no argument is provided. Custom adapters have to compute these hash values once to be able to compare them with the `methodHashCode` argument.

The following chapter contains a detailed discussion of the impact on performance of these options.

\(^2\)The name of the dynamic proxy class depends on the sequence of creation of dynamic proxy classes in the current Sun JVM version 1.4: It starts with $proxy$ followed by a sequence number.
Safe-points

Currently, no safe-points are used. Evolution starts as soon as

```java
IEvolutionManager.startOnlineUpgrade()
```

is executed. The method is designed to insert all safe-points declared and wait until the last one is hit. Thus, evolution is initiated immediately if no safe-point at all is available.

State Transfer

Currently, component state migration is implemented by member copying using reflective traversal of the object graph. Proxy migration is implemented as presented in section 3.5. According to the closed world assumption, nothing is serialized without explicit specification from the adapter component. There are two ways of how to perform evolution: Stop the old version component before starting the new one and short term coexistence for state migration. Serialization in the former case `doLaunchNewVersion() == false` is not yet implemented (dump to a file). In case of coexistence, serialization is replaced by a data structure in memory.

Building up an object graph for the new version component, the adapter may specify what objects get instantiated to replace the old version objects by implementing `createNew(oldObject)`. This state mapping function supports bijective correspondence between objects of different component versions only. However, this may easily be extended to support more complex kinds of mappings. Values of the state mapping function are stored in a hash table to allow proxies to find their new version hidden object.
4.2 Problems

Transparency

A simple call to a constructor of an external object as `new ExternalObject();` is not supported in most proxy implementations, including the java dynamic proxy API, CORBA and EJB. Thus, the developer has to anticipate the usage of its component in a proxy based environment. Because the Jadabs container is spontaneous, components need not to be aware of their deployment. Consequently, component evolution should neither require any anticipation. The class loading mechanism seems to be the way to avoid this problem. Aspect oriented techniques using byte code manipulation may be another approach. However, lacking functionality of constructor crosscuts as well as return value manipulation in PROSE have made it impossible to realize such a solution until now. On this account, implementation of transparent state evolution and safe-points based on around aspects is not yet feasible.

Identity Crisis

Using different class loaders, classes loaded from the same binary file may get incompatible. This is exactly what happens on a runtime evolution even if a class is not changed at all.

```java
1 oldCop.getCodebase().equals(newCop.getCodebase()); // true
2
3 Class oldMain = oldCop.getExtClass();
4 Class newMain = newCop.getExtClass();
5 oldMain.getName.equals(newMain.getName()); // true
6
7 oldMain.equals(newMain); // false if
8 // following condition holds;
9 oldMain.getClassLoader() != newMain.getClassLoader()
```

Listing 4.3: Identity crisis on using different class loaders

Consider listing 4.3. `oldCop` and `newCop` use their own class loader object. Even if the class definition does not change, this leads to two different class objects for the same class file. This identity crisis implicates that objects bound to a class definition in the loader of the old component cannot be taken over on runtime evolution. Instead, they have to be created using the definition of the class object stored in the loader of the new version component.

Reference Equality

In Java\(^3\), reference equality semantics is broken by the introduction of any kind of proxy. `aProxy == anotherProxy` does not compare the objects hidden by the proxies, but the proxy objects themselves. However, the client, being unaware of a proxy, wrote the code intended to compare the two unwrapped objects. One solution to this issue is to replace any occurrence of

\(^3\)Compared to C++ where it is possible to overload operators in a similar way as method overriding
4.2. PROBLEMS

equality comparisons that involve external objects by invocations of the proxies equals(Object) method. This may be accomplished at load time using byte code manipulation, but is not yet implemented in our framework.

Common Methods

There are methods that are defined for proxies as well as for the objects they hide. This is the case for methods declared by the java.lang.Object class, e.g. hashCode(), equals(Object) and toString(). These methods are heavily used in collection to compare objects lacking a specialized interface (e.g. java.lang.Comparable). Forwarding these methods to the hidden object turn them inaccessible on the proxy object\(^4\). In case of dynamic proxies, these methods are overridden in the DefaultAdapter:

```java
if (method == Object.class.getMethod("hashCode", null)) {
    return System.identityHashCode(dynamicProxy);
}
```

For transparent proxies, a cleaner approach has been taken to allow to determine on which object the method should be executed. Because just the evolution framework is aware of proxies, all methods are forwarded to the hidden object. To be able to compare transparent proxy objects themselves, additional methods are declared by the IProxy interface implemented by every proxy class: same and hash. These methods are guaranteed not to be invoked on the hidden objects and for internal usage only.

Proxy Superclass Constructor

Any class may inherit from another one. Similarly, a transparent proxy extends its super proxy class. A superclass of a transparent proxy must also be wrapped behind a proxy class if it is external with respect to the context in which the external reference is used. E.g. a proxy class must be created for a superclass that is declared inside the same component as its extending class that needs a proxy. Adding the extends keyword to the proxy class definition lets the proxy class automatically inherit from its super proxy class if available, because both belong to the same proxy component. However, the proxy subclass has to call its superclass’ constructor in its own. If the superclass does not declare a default constructor, the arguments passed to the super constructor may only be guessed by matching the runtime types of the arguments to the constructor signatures declared by the superclass.

Proxy Class Compilation

In its current implementation version, the transparent proxy class factory lacks minor functionality to be able to create a transparent proxy for every valid Java class input. Support for multidimensional array types in method parameters

---

\(^4\)These methods may be used internally to keep track of all proxy objects in a collection. Forwarding them to the hidden object may result in endless recursion
and return types is partly implemented only. However, one dimensional array
types such as Class[] or int[] work fine. After having written the source code
of the proxy class, the factory compiles the proxy using the \texttt{com.sun.javadoc}
compiler interface. This API of the tools library provided by SUN in their SDK
is unstable. The proxy factory currently uses the 1.3 version API and thus the
tools.jar of this SDK must be available on the class path. Classes compiled this
way are \textit{not} valid for previous releases of Java. If you want to take advantage
of features in a later Java version than 1.3, you have to include this versions
tool.jar. In addition, you have to change the call to the compiler to fit the API
of the library. SUN ensures that Java 1.5 will provide a specification of the
compiler API to remove such inconvenience.

\textbf{Class Name Clash}

In the adapter components which guide an evolution step, one has access to
both old and new version objects of the evolving component. As they belong
to different components, classes of corresponding objects may have the same
fully qualified name. This implicates that the adapter component may just load \textit{one}
of these classes as they are undistinguishable for it. It is not possible
to declare an object as type \texttt{EventService} version 1 and another one as
version 3. The adapter component does not know which version of the class
definition is loaded. To avoid these problems, it is recommended not to import
any classes of neither version of the component subject to evolution. Object
may be statically typed as \texttt{java.lang.Object} or other classes defined outside
their component (e.g. \texttt{ch.ethz.iks.jadabs.IComponent}, \texttt{java.util.Vector}
or \texttt{jxme.IMessageListener} in case of \texttt{EventService} objects).
4.3 Samples: First Steps in Runtime Evolution

Light-weight Evolution using Dynamic Proxies

As a sample, a runtime evolution step of testcop component from version 1 to 2 is implemented using the dynamic proxy hiding mechanism (see section 4.1). To start the sample, make sure you have placed the testcop.jar in the pcoprep folder of Jadabs. Be sure no component that specifies testcop.jar as one of its dependencies must be in the pcoprep folder yet. Next, startup Jadabs by typing

perl jadabs.pl dynamic

The testcop component is loaded and starts producing output like:

```
ch.ethz.iks.test.TestComponentMain - 1: teststring
```

Now, you may insert the evolution manager component evolution.jar to the pcoprep folder (alternatively, it may be there at startup of Jadabs already). The component tries to evolve the Jadabs core library as well as the resource and class loader control objects of each component loaded. It will succeed to to evolve testcop if no other component depends on it. From now on, the evolution manager will take care to hide references passed across component boundaries. To verify that this happens, insert the tcEvolutionTest.jar component into the pcoprep folder. Because testcop is a dependency of tcEvolutionTest, each class file loaded is searched for external references. Upon detection of a reference that is declared in the testcop component, the dependency testcop.jar is replaced by the proxy component proxy4.testcop. Having not set the -proxy ifc argument, dynamic proxy classes are created, loaded and instantiated in the tcEvolutionTest component. Note that the tcEvolutionTest component has to comply to the coding conventions mentioned in section 4.1 to ensure correct operation. The test case performs some operations and ensures that only proxies are passed across component boundaries. The prompt

```
[ HELP ]  cp bin/ext/V2_testcop.jar bin/pcoprep/testcop.jar
```

will appear if everything works fine. Make sure you create a component V2_testcop.jar having version 2 specified in its manifest, then insert it into the pcoprep as testcop.jar.

After a short delay, the enhanced component loader will initiate a runtime evolution step and perform again some operations on the new version component. In particular, it is ensured that (a) the correct class file is loaded, (b) the main object reference remains the same proxy object (is a singleton!) and (c) that state is transferred to the new version component objects. Note that the output changes to

```
ch.ethz.iks.test.TestComponentMain - 2: teststring
```

In case of success, you will get the following output:

```
[ DONE ]  Successfully evolved testcop to version 2
```
Fully-fledged Runtime Evolution using Transparent Proxies

To run the sample evolution of the event system, proceed as in case of the dynamic proxies sample. In contrast to the previous sample, provide start Jadabs with the command line argument

```perl
perl jadabs.pl transparent
```

This enables interface evolution and method invocation based on the methods hash code (`-proxy ifc -adapt hash`).

The component to be evolved is packaged in `eventsystem.jar` and the test case in `esEvolutionTest.jar`. Because the `eventsystem` component has dependencies itself, those must be available. Thus, `jxmeudp.jar` and its dependency `commons.jar` must be put in the pcooprep folder before `eventsystem.jar`. The evolution manager also needs to add support for evolution to the dependencies of the component to evolve. This implicates that `evolution.jar` has to be loaded before `eventsystem` as well. After having completed testing on version 1 of `eventsystem`, insert version 3 when the following message is shown on the screen:

```
[INPUT] Please load version 3 of eventsystem.jar
```

In addition to the conditions (a), (b) and (c) that are also checked by the light-weight test, this test further verifies that (d) unstable proxies are created when passing an external reference as argument to a method. In contrast, unstable proxies of non-evolving components (e.g. `esEvolutionTest`) are kept in the evolving component (e) while unstable proxies of the evolving component (passed to one of its dependencies, e.g. `jxmeudp`) are created anew (f).

These newly created proxies have to point to the new version object and take the place of the obsolete unstable proxy (g), thus preserving state of the dependency. You will get a

```
[ DONE ] Successfully evolved eventsystem to version 3
```

message on successful completion of the evolution step. You may consult the `proxydump.dat` file to compare the content of the proxy registry before and after evolution.

An overview of this evolution step including both old (v1, top) an new version (v3, bottom) component of `eventsystem` (in the middle), their dependency `jxmeudp` (right) and client `esEvolutionTest` components (left) are illustrated in figure 4.3. Proxy objects are marked by ”p4”, adapter by ”a2” and version dependent objects labeled with their version number (e.g. ”v3”) for clearness. Each bean component (Client, ES v1/v3, Jxme) is wrapped behind a proxy component and an adapter component. While the later gets replaced along with its bean version 1, the former is kept beyond evolution of the `eventsystem` to version 3. Thus, the Client keeps its reference on the transparent proxy object p4_ES. A new version adapter a2_ES_v3 replaces the a2_ES_v1, forwarding method invocations to its version 3 bean ES_v3.
Figure 4.2: Sample evolution scenario of the eventsystem component
Similarly, the reference to Client passed to $p4\_ES\_v1$ gets hidden behind an unstable proxy $p4\_Client$ before being stored in the listeners variable. This is necessary as the bean object Client implements $p4\_IEL$ and not $IEL\_v1$ (as $p4\_Client$ does): It would not be assignable to $IEL\_v1$ otherwise. In contrast to the stable proxy $p4\_ES$, $p4\_Client\_v1$ must be replaced on evolution of the eventsystem component as the interface $IEL\_v1$ may be redefined in version 3. It is defined in the adapter component $adapter2\_eventsystem\_v1$ therefore. $p4\_Client\_v3$ is thus created and assigned to listeners. Via the adapter $a2\_Client$, it points to the same bean object Client as before. Although the esEvolutionTest component does not evolve itself, its objects which are stored in the evolving component must be updated (more exactly: their proxies evolve). If an object of the evolving component eventsystem is stored in a dependency (e.g. Jxme.subscriptions), it is wrapped, too ($p4\_ES\_v1$). However, this unstable proxy remains valid after evolution as the interface it implements IML does not change. Just the adapter must be replaced by $a2\_ES\_v3$ because the old one does not exist anymore.

After the evolution step has been finished, objects with a "v1" label do not exist anymore (except the $p4\_ES\_v1$). Thus, on receipt of a message, Jxme delivers it\(^5\) to its subscriptions, including $p4\_ES\_v1$. The proxy then forwards it to the adapter $a2\_ES\_v3$ that maps the call to the $ES\_v3$ object. Processing the message, the bean may generate events that it publishes to its listeners. The new version unstable proxy $p4\_Client\_v3$ has taken the place of the version 1 proxy $p4\_Client\_v1$. This way, the Client gets notified finally.

\(^5\)The message object itself has to be wrapped if it is defined in the Jxme component
Chapter 5

Benchmarking

To evaluate the efficiency of the transparent proxy, we have measured the overhead introduced by the proxy mechanisms described in section 3.2 against the Java dynamic proxy and the original Jadabs container without any proxy.

5.1 Experimental Setup

All benchmarks were run inside the Jadabs container version 0.4.3 on a machine with the following properties:

- RedHat Linux 7.3
- Intel Xeon 2.4 GHz CPU
- 1024 MB RAM
- SUN Hotspot VM Implementation 1.3.1

For each measurement, the difference in system time was taken to approximate CPU time consumption. The first run was excluded from the measurement to avoid that initialization and lazy creation effects (e.g. of the proxy class and objects) as well as just in time compiling falsify the result. Each benchmark case typically executes its target method(s) 1’000’000 times to obtain significant results. In addition, the arithmetic average of several runs was computed for each benchmark. Output is written to the log4j [3] "ProxyBenchmark" file appender currently configured to the path `bmarks/proxy_bm.html`.

5.2 Configurations Compared

The benchmarks were run in three configurations each: Without a proxy, using the Java dynamic proxy and using the transparent proxy. To be able to run case 3, the target methods have to be declared in a different Jadabs component than the benchmark itself.

1. **Without proxy**: Corresponds to a simple invocation of the target methods of the benchmark cases. This original Jadabs configuration is encapsulated in the component `bmWithoutProxy.jar`.
2. **Java dynamic proxy**: The callee object is replaced by a dynamic proxy created through the Java API. It is wrapped inside a custom Invocation-Handler object to be assignable as handler of the proxy (see listing 5.1). The target method `invoked` is thus called on the dynamic proxy and forwarded to the original callee `hidden`. Note that ITestCop declaring `subscribe(ITestListener)` is implemented by TestComponentMain. Neither arguments nor return values are wrapped by this configuration. It is stored in the component `bmDynamicProxy.jar`.

```java
1 public void prepareCase(IBenchmarkCase bmark) {
2     bmark.prepare();
3     ITestCop tc = TestComponentMain.createComponentMain();
4     InvocationHandler tcIH = new InvocationHandler() {
5         private final ITestCop hidden = tc;
6         public Object invoke(Object callee, Method invoked, Object[] args) {
7             if ("subscribe".equals(invoked.getName())) {
8                 return this.hidden.subscr();
9             } // else if ...
10         }
11     };
12     Proxy p4tc = (IEventService) Proxy.newProxyInstance(
13         this.es.getClass().getClassLoader(),
14         new Class[] {ITestCop.class},
15         tcIH
16     );
17     bmark.setCallee( p4tc );
18 }
```

Listing 5.1: Dynamic proxy benchmarking configuration

3. **Transparent proxy**: Note that the benchmarking component should be loaded not until the initial evolution of its dependency components `event-system.jar` and `testcop.jar` has been completed. At load time of the benchmarking component `bmTransparentProxy.jar`, this lead to the creation and loading of a proxy class in the associated proxy component (e.g. `proxy4_testcop`).

These three configurations are available as Jadabs component in a Jar file each. Upon insertion in the container, the components execute the following benchmark cases under its configuration.

### 5.3 Benchmark Cases

We have focused on runtime performance in four selected cases but also done some memory usage benchmarks.

Application classes that benefit most from runtime evolution capabilities are characterized by a demand for high reliability and (often) throughput. Thus, it is interesting to measure the performance of a single method invocation across
5.4. RESULTS

components using the transparent proxy in between. We have done this running the four benchmark cases listed below on each configuration:

a) `ch.ethz.iks.eventsystem.EventServiceImpl.toString()` as inherited from the class `java.lang.Object` represents a typical operation provided by the Java runtime library. The method is not overridden. Configuration (3) will check if its return value has to be wrapped before returning it as is.

b) `IEvent.toXMLString()` is nearly twice as complex as `Object.toString()`. In addition to case (a), a cast to the runtime type `StringEvent` is needed in the adapter to allow the operation on the callee object.

c) An almost empty method changing the internal state to prevent method in-lining (see `nop()` in listing 5.2) is used to estimate the costs of the double indirection of invocations and administrative tasks associated with a language level based proxy. Neither the method body nor its return type (void) add significant extra costs.

```
1    public void nop() {
2        this.running = true;  // bmark (b)
3    }
4
5    public TestComponentMain getHiddenObj() {
6        return singleton;  // bmark (d)
7    }
8
9    public void subscribe(ITestListener l) {
10       this.listener = l;  // bmark (e)
11    }
```

Listing 5.2: Source of methods benchmarked

d) `wrap return`: A getter method revealing a hidden object (`getHiddenObj()` in listing 5.2) illustrates one worst case scenario for the transparent proxy configuration. Instead of passing the return value back, it has to be hidden behind a proxy that needs to be created on the fly and returned on behalf to ensure component version independence.

e) `wrap argument`: A method taking an interface as argument (`subscribe(ITestListener)` in listing 5.2) illustrates the second worst case scenario for configuration (3). The argument implementing the `ITestListener` interface declared in the proxy component of `proxy4_testcop` has to be wrapped behind an unstable proxy that implements the current version of the interface as declared by `testcop.jar`. Again, nothing else happens in this method.

5.4 Results

Starting at an overhead factor 78 using reflection and one generic adapter, we tuned the transparent proxy implementation in five steps shown in table 5.4.
As a first optimization (column optimize step I) we are not anymore using reflection to access the enclosing method object for calling

```java
invoke(Object callee, Method called, Object [] args)
```

. Instead, we extended the `InvocationHandler` interface to accept the name of the method instead (`IAdapter`). This speeds up step I by nearly 20% with respect to the original transparent proxy implementation in the first benchmark case (a). The usage of custom adapters in optimize step II for different proxy classes instead of the generic `DefaultAdapter` makes two time consuming reflection statements obsolete:

```java
Method hiddenMethod = hiddenClass.getMethod(methodName, params);
hiddenMethod.invoke(hiddenObject, args);
```

This improves performance by another 50% based on optimize step I in the `toString` invocation measurement.

Most effect had a simple cache mapping class names to their defining component binaries: The return value of a hidden method needs to be hidden inside a proxy if its runtime type is defined in the same component as the methods declaring type. Optimize step III thus avoids repeated looping through all Jar file contents e.g. in search for the type `java.lang.String`. A huge performance boost of 2000% resulted from step III in reference to step II. In comparison with the original configuration (1), the transparent proxy is about 50% slower in the typical case (a).

Case (c) reveals another performance bottleneck: Further extension of the `IAdapter` interface replaced the `methodName` parameter and the implicated String comparison by an efficient integer based hash code in the custom adapters used (see listing 3.13 in section 3.4). It is important that the method hash codes have to be computed in a way that they are equal for the proxy method and the hidden method but they must not be equal in case of different parameter types. This fourth optimization step caused the benchmark case `nop` to improve over 75%, leaving an overhead of factor 18 in case of an empty method body with respect to the original configuration (1). Even with optimization step V applied, case (d) slows down execution by about 100%.

<table>
<thead>
<tr>
<th>optimization step</th>
<th>toString case (a)</th>
<th>empty method case (c)</th>
<th>wrap return case (d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: reflection</td>
<td>330'000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I: method name</td>
<td>273'000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>II: custom adapter</td>
<td>182'040</td>
<td></td>
<td></td>
</tr>
<tr>
<td>III: cache class2cop</td>
<td>6'813</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>IV: hash codes</td>
<td>4'425</td>
<td>146</td>
<td>3368</td>
</tr>
<tr>
<td>V: cache hidden2proxy</td>
<td>4'557</td>
<td>145</td>
<td>1621</td>
</tr>
</tbody>
</table>

**Table 5.1:** Optimization steps, one million invocation each (milliseconds)
5.4. RESULTS

<table>
<thead>
<tr>
<th>benchmark case</th>
<th>without proxy (ms)</th>
<th>overhead (2) Java proxy</th>
<th>overhead (3) transparent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) toString</td>
<td>4291.5</td>
<td>15.5%</td>
<td>6.2%</td>
</tr>
<tr>
<td>(b) toXMLString</td>
<td>7482.1</td>
<td>3.7%</td>
<td>7.2%</td>
</tr>
<tr>
<td>(c) empty method</td>
<td>7.80</td>
<td>690.6%</td>
<td>1'859.1%</td>
</tr>
<tr>
<td>(d) wrap return</td>
<td>6.73</td>
<td>n/a.</td>
<td>9'148.5%</td>
</tr>
<tr>
<td>(e) wrap argument</td>
<td>7.87</td>
<td>n/a.</td>
<td>20'600.0%</td>
</tr>
<tr>
<td>(f) = (b) + (d)</td>
<td>≈7500.</td>
<td>≈4.8%</td>
<td>≈8.2%</td>
</tr>
<tr>
<td>(g) = (b) + (e)</td>
<td>≈7500.</td>
<td>≈4.6%</td>
<td>≈21.5%</td>
</tr>
</tbody>
</table>

Table 5.2: Invocation performance results

As seen from table 5.4, the Java dynamic proxy (column slowdown Java proxy) imposes a significant slowdown with respect to the original configuration without a proxy in all benchmarks measured. The transparent proxy (column slowdown transparent) is much mightier but also slower due to language level based implementation: On-the-fly wrapping of return values and arguments is performed by the transparent proxy only.

However, remember that cases (b), (d) and (e) were constructed to show the overhead introduced by configurations (2) and (3). Each method call is subject to a slowdown of another 18 calls. An external argument adds about 190 times and a return value across components 80 times of a Java method invocation. This only applies to methods having a minimal body: For methods that does something useful like `Object.toString()`, the overhead is reduced in proportion to its execution time to about 7% each.

The method invocation performance worst case scenario is about 200 times slower than without a proxy: Verifying that the argument value needs to be wrapped, proxy object creation and registration as well as custom adapter creation and initialization adds up a huge overhead on returning an object. However, this feature is not available in the Java dynamic proxy API at all.
Chapter 6

Conclusion and Outlook

6.1 Summary
In this thesis, we have discussed the issues in runtime evolution of components and presented solutions based on the Jadabs container written in Java. Four main concern have been approached: Reference replacement, code replacement, initiation of component migration and state transfer. Our goal is to realize these tasks in a transparent manner. The key concepts to achieve this are summarized next.

**Transparent Proxy** In contrast to any proxy known to the author at the time of writing, the transparent proxy does supports unanticipated use of legacy code: There are no conditions to the objects subject to hiding must satisfy. Custom class loader and dependency replacement is provided transparently to the component. Also, no limitations in usage has been found except the *this problem* (see section 4.2) common to all kinds of proxies. *Stable* proxies exist beyond evolution of the object they hide.

**Safe-points** Finding points in execution of the running component where state may be migrated and execution be resumed from afterwards is essential. However, it is not always possible to wait until a component gets idle: The concept of safe-points is introduced to allow execution to be suspended at thread exits or method returns for example. Safe-points may easily be implemented using dynamic AOP.

**Adapters** To support arbitrary modifications in code including interface modification, adapter objects are used. An adapter maps invocations on the proxy to its current implementation object and method. It has to be customized for a specific component version and is placed outside the component it maps to. Adapters encapsulate functionality needed for runtime evolution exclusively. This results in a clear separation to application logic. They are also used to implement component behavior during evolution to ensure availability and to specify state to be transferred.
This concept provides independent evolution of components. Users of an evolving component remain compatible with the component after evolution: Updating of clients is not necessary. Type safety is preserved as adapters are realized at the language level.

**State Evolution** Performing state migration transparently is very hard to realize. The more powerful the mechanism to define points of migration, the more complex becomes migration itself (e.g. definition of safe-points at runtime). In the context of runtime evolution, migration of call stack and program counter has to account for code evolution.

Thus, our approach goes beyond existing agent migration techniques. Unanticipated and customizable object graph migration is supported. Mapping of object state, call stack and program counter between component versions is foreseen to allow for code evolution. Call stack and program counter information is retrieved through JPDA. Execution is escaped and resumed at the components entry points (proxies) after the proxy has updated its hidden object. Around aspects for each frame on the mapped stack are used to avoid duplicate execution on building up the call stack. Statements after the mapped program counter on the topmost frame have to be copied to the advices to guarantee seamless returning from the call.

**Performance** The transparent proxy is far more than a simple proxy: Transparent and on-the-fly wrapping of objects during method entry (arguments) or exit (return value) as well as support for unanticipated code evolution (e.g. interface adaptability) being just the main features.

These features do not come for free: Method invocations across component boundaries are subject to a performance penalty\(^1\).

Apparently, the transparent proxy is too slow for simple operations. However, such operations may be performed without the need of external software components. For operations at least as time consuming as `Object.toString()` or even more complex operations that satisfy the usage of a specialized library component, the overhead becomes mostly acceptable.

---

\(^1\)You may specify components to deactivate evolution if performance really matters. Add Upgradeable: NONE to the component’s manifest file. As a consequence, you will lose the capability to perform runtime evolution on these components.
6.2 Future Work

As mentioned earlier, advanced features like sophisticated state transfer and safe-point support are not yet implemented. These are the two most important todo’s to complete functionality. But there are also other tasks waiting to be done:

**Migration of Library Components** We have concentrated on Jadabs service components in our implementation. Library components, however, are as important as service components with respect to runtime evolution support. Because library components do not expose a main class, it becomes difficult to transfer state by reflection. The root objects of the object graph to traverse are unknown at the language level. However, any object being referenced from the outside of the library is wrapped by a proxy. Thus all the objects hidden by a proxy form the root set of the object graph of the library component. Object state migration of library components may be implemented this way.

**Declarative Specification of Safe-points** To unburden the developer to write PROSE aspects to catch suitable execution points for runtime evolution initialization, it would be a great benefit to be able to find safe-points automatically or to create them from a specification file.

**A Dynamic AOP Proxy** As an alternative solution to the transparent proxy, it would be interesting to have one built by dynamic aspects as discussed in section 3.2. This has not been done because lacking functionality in the PROSE system. Another approach that is feasible would be to leave the method bodies of the transparent proxies empty. Instead of an adapter, a dynamic aspect crosscutting method entries may be used to map to the current implementation version. As weaving these aspects is supported at runtime in PROSE, code evolution does not have to be anticipated by the developer.

**Support Added Methods in the Proxy** Code modifications may include addition of methods to a class definition. After evolution, these methods are available on the new version objects. However, clients may not call them since they are hidden behind a proxy. Because the proxy was generated before evolution, methods added later are not accessible through the proxy. For clients that are compiled with the old version component, this is no problem since they do not know these methods and cannot call them.

In contrast, another client which was compiled with the new component version may invoke methods added to new class definitions. How to prevent a NoSuchMethodError on the proxy? At the time of proxy generation, one does not know how methods added in a later version will look like. An approach to solve this problem is to extend the IProxy interface by a generic method that takes a callee object, a method (name) and an array of arguments similar to the ones presented in section 4.1. On loading the client code, one instruments its byte code: Any occurrence of an added method is replaced by a call to
the generic method. This is the same concept the proxy uses to forward its
invocations to its adapter. However, note that this does not turn the proxy
obsolete as the adapter is version dependent. If clients had direct access to the
adapter, the adapter class would be loaded by them and thus no adapter to
future version could be installed anymore.
The jadabs project - java adhoc application bootstrap.


Apache Software Foundation. Log4j.


[24] Xerox Parc Inc. AspectJ.