Aspect-Oriented Programming with Runtime-Generated Subclass Proxies and .NET Dynamic Methods

eva Kühn
eva@complang.tuwien.ac.at

Gerald Fessl
fessl@complang.tuwien.ac.at

Fabian Schmied
fabian@complang.tuwien.ac.at

Institute of Computer Languages, Vienna University of Technology
Argentinierstraße 8
A-1040 Wien, Austria

ABSTRACT

Nine years after its first publication, aspect-oriented programming (AOP) is finding more and more support, but adoption by the industry is still slow. The subclass proxy approach, a new implementation mechansim for .NET-based AOP tools, claims to have the potential of easy adoptability. This paper analyzes subclass proxies as a lightweight infrastructure for AOP, characterizing its properties, advantages, and disadvantages as compared to other implementation techniques. It evaluates technical strengths and weaknesses as well as psychological factors which could influence adoption, and it shows the results of performance benchmarks. In addition, it augments the mechanism with a new way of providing aspects woven at runtime with efficient and safe access to objects' private members.

Keywords
Aspect-oriented programming, code generation, subclass proxies, evaluation.

1. INTRODUCTION

Since 1997, when Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin introduced the term *aspect-oriented programming* (AOP) [KL^{+}97], AOP has found a lot of support in the research community. Successful implementations of the paradigm on the Java platform include AspectJ [LK98], JBoss AOP [Bur04], and Spring AOP [JH^{+}05]. Still, industrial adoption of this new mechanism for software development is naturally slow, and adoption in the field of .NET is hampered by the lack of production-quality AOP tools for this platform.

Recently, projects such as NAspect [Joh05] and XL-AOF [eKS05b] (originally introduced for modelling the concerns of space-based distributed applications in an aspect-oriented fashion [SeK04]) have introduced light-weight implementations of the aspect-oriented programming paradigm based on the .NET platform. Both of them are founded on the same technology, which we call *subclass proxies*. Using subclass proxies as an infrastructure for AOP has some technical advantages over the more classic approaches (i.e. weaving compilers and postcompilers), as well as the important benefit of being easy to adopt by software developers.

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

In this paper, we perform an analysis of the technology backing both NAspect and XL-AOF. We analyze it on a technical level, describing its implementation, and show its potential as a weaving approach, classify it and characterize it and its properties, advantages, and disadvantages, and we evaluate the concept from an adoptability viewpoint. We also introduce a novel way of allowing aspects to efficiently access private fields and methods of their target objects, which was a privilege of code-weaving approaches until now. By providing a performance analysis, we show that the performance of solutions based on subclass proxies is better than it is often assumed of proxy-based approaches.

The rest of this paper is structured as follows: section 2 describes the technical background of the subclass proxy mechanism and its use for aspect-orientation (note that we use notions such as *aspect*, *advice*, *introduction*, and *join point* as defined by AspectJ [LK98] without further explanation). Section 3 augments the mechanism by introducing an approach for accessing a target object's private secrets from within a subclass proxy-based aspect. Section 4 does an extensive analysis of the concept's potential as a base for AOP and section 5 evaluates it from a performance viewpoint. Section 6 concludes the paper.

2. SUBCLASS PROXIES

A *proxy* P is defined to be an object which acts as a placeholder for a target object T [GHJV95]. Wherever

T is expected, the proxy can be used instead, transparently extending the target object's behavior or controlling access to it without client code needing to be adapted. *Runtime proxies* are proxies created dynamically at run time, without the programmer having to prepare a dedicated proxy class for every target class.

The Microsoft .NET Common Language Runtime provides a *transparent proxy* [Low03] mechanism whose uses are limited by its functionality and its impact on application design. Design-wise, it requires the proxied object to be derived from the *System.ContextBoundObject* base class. This will not be an option in some cases, in other scenarios it might require unclean changes to the application design, which is contrary to the goals of AOP [KL$^+$97]. Functionally, it is designed for .NET Remoting (i.e. communication between different application domains, processes, or computers) and it cannot extend behavior of an object which is accessed from its own context (including self calls—methods called on the *this* reference), a drawback for realizing a join point model. Introduction can not be implemented using transparent proxies at all. Positive aspects of the mechanism include that proxies are also *created* transparently because the CLR intercepts the *newobj* instruction (*new* in C$^\#$, *New* in *Visual Basic .NET*) and returns a proxy instead of the target object. In addition, the CLR automatically corrects the *this* reference within T's methods—it refers to P instead of T, which is important if it is to be passed to other objects.

Looking for an alternative to transparent proxies, it can be noted that the property of substitutability used previously for defining the term "proxy" is similar to the *Liskov Substitution Principle* (LSP) [LW94], which describes the relationship between subtypes. Like a proxy P, which can be substituted for an object T, the LSP states that an object of a subtype can be substituted for one of a supertype. This similarity can be used to implement proxies using the subtyping mechanisms present in .NET: interfaces and inheritance.

To realize a proxy using interface implementation—we call this an *interface proxy* approach—the target object T must implement a set of interfaces I and all client code must access T via these interfaces only. Then, a proxy object P can be created which also implements I and holds a reference to T for delegation. T in the client code can be transparently replaced by P, which plays the role of a proxy. Within T's method implementations, however, the *this* reference refers to T rather than P, which is problematic if the reference is used to access the object: such access will not be registered by the proxy. In addition, like with transparent proxies, the interface proxy approach does not allow self calls to be extended, it is therefore a suboptimal solution as well.

Realizing proxies using inheritance—the *subclass proxy* approach—is different from the aforementioned approaches. Whereas transparent and interface proxies have an object instance P replacing a target object instance T (and delegating), inheritance allows proxy and target to be one and the same object: a class P is derived from the target class T, overriding its methods and delegating to the original implementation. When P is instantiated, one object instance implements both P's and T's functionality. Since subclasses are subtypes in .NET, the LSP applies and instances of P can be used wherever instances of T are expected. Subclass proxies intercept self-calls correctly, the *this* reference is automatically correct, and introduction is possible via interface implementation (see below). Figure 1 compares the unproxied scenario with a simplified drawing of transparent, interface, and subclass proxies.

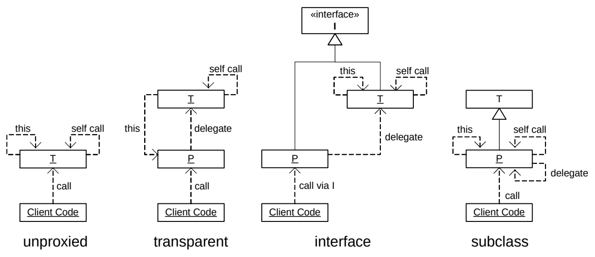

Figure 1. Proxy approach visualization.

In contrast to transparent proxies, both interface and subclass proxy have the disadvantage of needing a class factory [GHJV95] to make object creation transparent to client code. Table 1 summarizes the properties of the different proxy approaches, positive characteristics are shown in boldface.

	Transparent	Interface	Subclass
Parent class	ContextBoundObject	**arbitrary**	**arbitrary**
Creation	**newobj**	factory	factory
Usage	**direct**	interfaces	**direct**
This reference	**P**	T	**P**
Extend self calls	no	no	**yes**
Introduction	no	**yes**	**yes**

Table 1. Properties of proxy approaches.

2.1 Runtime Subclass Proxies

In the simplest form, subclasses do not implement a runtime proxy approach: the programer needs to write dedicated derived classes for each target type, manually overriding the methods that need to be extended. Using code generation, this can however be generically performed at runtime by a tool or framework. The .NET Base Class Library provides two powerful mechanisms allowing for runtime code generation: the *System.Reflection.Emit* namespace contains low-level classes and methods to dynamically generate .NET assemblies and types, *System.CodeDom* provides base classes for higher-level code generation. In this article, we will concentrate on *System.Reflection.Emit*.

Listing 1 shows how to dynamically generate a subclass of an arbitrary type at runtime using *System.Reflection.Emit*: it asks the current application domain to define a dynamic assembly, naming it "proxies", which in turn is used to define a dynamic module named "proxies" as well; by giving the module a DLL file name (and using the *RunAndSave* flag when creating the assembly), it is possible to save the module to disk after generation in addition to using its types. The dynamic module is used as a factory for the type which is to be created. The type's base class is specified to be *baseType*, the parameter passed to the method (which corresponds to the proxied type T), its access attribute is *Public* to make it publicly accessibly from other assemblies, and its name is defined by attaching "___Subclass" to the base type's name. By calling its *CreateType* method, the dynamic type is finished and the corresponding *Type* object (P) is returned and can be instantiated using *System.Activator.CreateInstance*.

```
public Type DefineSubclass(Type baseType) {
 AssemblyBuilder a =
   AppDomain.CurrentDomain.DefineDynamicAssembly(new
    AssemblyName("proxies"),
   AssemblyBuilderAccess.RunAndSave);
 ModuleBuilder m = a.DefineDynamicModule("proxies",
   "proxies.dll");
 TypeBuilder subtype = m.DefineType(baseType.Name +
   "___Subclass", TypeAttributes.Public, baseType);
 return subtype.CreateType();
}
```

Listing 1. Creating a subclass at runtime.

2.2 Weaving Based on Subclass Proxies

Aspect-oriented programming is based on two main concepts: join points, i.e. points in the imperative program flow where aspects' advice methods are triggered, and introduction of new members to the aspects' target classes. Both concepts can—to a degree—be implemented with subclass proxies; the method of doing so is described in this section. An analysis on the join point model which is gained from this mechanism is performed later in section 4.

Join Points By overriding the methods of its base class, a proxy class can provide replacement code for them, delegating to the original (base) implementation if necessary, and triggering join points before, after, and instead of (or around) method executions.

With *System.Reflection.Emit*, overriding methods is easily possible by inserting code prior to calling *TypeBuilder.CreateType*. Listing 2 shows how to override all virtual methods of the given base type. It does so by using the .NET *Reflection* mechanism to find all the public and nonpublic instance methods of the base type, checking whether they are virtual, and, if yes, defining a method with the same name and signature. The signature is found by inspecting the parameters of the base method and extracting their types (using an anonymous delegate for brevity); the override's return type is the same as that of the base method.

```
foreach (MethodInfo m in baseType.GetMethods(
   BindingFlags.Public | BindingFlags.NonPublic |
   BindingFlags.Instance)) {
 if (m.IsVirtual) {
   ParameterInfo[] parameters = m.GetParameters();
   Type[] parameterTypes =
      Array.ConvertAll<ParameterInfo,
        Type>(parameters,
      delegate(ParameterInfo parameter)
      { return parameter.ParameterType; });
   MethodBuilder subMethod =
     subtype.DefineMethod(m.Name,
      MethodAttributes.Virtual |
        MethodAttributes.Public,
      m.CallingConvention, m.ReturnType,
        parameterTypes);
   ILGenerator il = subMethod.GetILGenerator();
   il.Emit(OpCodes.Ldarg_0);
   foreach (ParameterInfo parameter in parameters) {
     il.Emit(OpCodes.Ldarg, parameter.Position + 1);
   }
   il.EmitCall(OpCodes.Call, m, null);
   il.Emit(OpCodes.Ret);
 }
}
```

Listing 2. Overriding methods.

The code snippet then defines the override's method body via IL (intermediate language) opcodes. The body loads the object reference (argument 0) and the parameters, calls the base method, and finally returns to the caller. An AOP approach can insert additional code into the body, implementing before, after, and around advice and delegating back to the original method if desired.

Opposed to method join points, construction and creation join points need not be implemented by the proxy itself: they can be triggered by the factory used to create the proxy types and their instances. Property get and set join points are equivalent to method join points, since all properties are backed by respective getter and setter methods. Finalizer join points can be implemented the same way as method join points by overriding the *Finalize* method of the object. Field get and set join points cannot be implemented with subclass proxies. It is up to the AOP implementation of how to bind advice methods to the join points implemented with the subclass proxy mechanism, the most runtime-efficient way being to directly encode calls to advice methods (or even inline these) into the override's method body.

Introduction As opposed to compiler-based AOP approaches, runtime weaving approaches cannot simply introduce new members to a class. While it is easily possible to add these members to a subclass proxy, client code uses the proxy transparently and has no way of accessing the introduced entities with a statically typed programming language. The only form of introduction easily conceivable for runtime approaches is interface introduction: an aspect can add an interface and its implementation to an object, and client code can cast its object reference to the interface type. Since the proxy implements the interface, the cast succeeds.

Interface introduction can be easily implemented with *Reflection.Emit* by having the dynamically created subclass implement the interface. This is similar to listing 2 and therefore not separately demonstrated here.

3. PRIVILEGED ACCESS TO TARGET OBJECTS' SECRETS

One important property of aspects is that they often require more privileged access to their target object's internals than other objects should have, because they implement cross-cutting concerns which can be tightly coupled to the objects they cut. While a subclass proxy naturally has access to all public and protected (family-accessible) fields and methods of its base class, it has no access to private or assembly-visible members.

.NET provides a Reflection mechanism to work around this: given the necessary rights, every object can reflect over another object's private fields and methods in order to inspect and change the fields' values or invoke the methods. However, Reflection is not optimized for performance: our tests have shown that accessing a field via reflection is around 200 to 700 times slower than direct access, and still 180 times slower than invoking an accessor method would be. Since field access is such a basic operation, this might conceivably slow down an aspect-oriented application, depending on the degree of coupling between aspects and object state.

It would be desirable, therefore, to at least have accessor methods for those private fields required by an aspect. Unfortunately, such a method cannot be added to a subclass, which has no access to private members. As a solution, with .NET 2.0 there is a new mechanism called *Lightweight Code Generation* (LCG), or *Dynamic Methods* [Mic06]. It allows methods to be generated at runtime which can be attached to any existing type, allowing access to all its private data. Access to the method is provided via a delegate, allowing flexible invocation which is still 15 to 20 times faster than reflection-based field access. The diagram in figure 2 shows a performance comparison of the different ways of accessing fields (measured on an Athlon XP1800+ with 512 MB RAM and .NET framework v2.0.50727).

Field Access Framework For an AOP approach based on subclass proxies, we suggest a field access infrastructure which consists of a set of generic delegate types *Setter* and *Getter* as strongly typed wrappers for the accessor methods, an accessor method generator *MethodGenerator* which generates the accessor methods using LCG, and a wrapper structure for fields, which simply initiates the accessor method generation

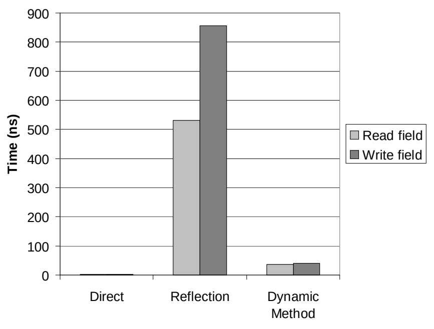

Figure 2. Performance of access methods.

when being constructed and provides a *Value* property delegating to the accessor methods for convenient use.

Listing 3 shows the source code for these infrastructure entities. The method body constructed by *CreateSetter* simply loads the given target object (argument 0) followed by the value (argument 1), which is then stored in the given field before returning. The method body constructed by *CreateGetter* first loads the target, then loads the field value, and then returns, leaving the field value on the evaluation stack, returning it to the caller. Because the created dynamic methods are associated with the target type (ClassType), they can safely access even private fields using the *ldfld* and *stfld* opcodes.

```
delegate FieldType Getter<ClassType, FieldType>(
   ClassType target);
delegate void Setter<ClassType, FieldType>(
   ClassType target, FieldType value);

class MethodGenerator {
 public static Setter<ClassType, FieldType>
   CreateSetter
    <ClassType, FieldType>(FieldInfo fieldInfo) {
  DynamicMethod newMethod = new DynamicMethod(
     fieldInfo.Name + "___GeneratedSetter",
       typeof(void),
     new Type[] { typeof(ClassType),
       typeof(FieldType) },
     typeof(ClassType));
  ILGenerator ilGenerator =
    newMethod.GetILGenerator();
  ilGenerator.Emit(OpCodes.Ldarg_0);
  ilGenerator.Emit(OpCodes.Ldarg_1);
  ilGenerator.Emit(OpCodes.Stfld, fieldInfo);
  ilGenerator.Emit(OpCodes.Ret);
  return (Setter<ClassType, FieldType>)
     newMethod.CreateDelegate(
     typeof(Setter<ClassType, FieldType>));
 }

 public static Getter<ClassType, FieldType>
   CreateGetter
    <ClassType, FieldType>(FieldInfo fieldInfo) {
  DynamicMethod newMethod = new DynamicMethod(
     fieldInfo.Name + "___GeneratedGetter",
     typeof(FieldType), new Type[] {
       typeof(ClassType) },
     typeof(ClassType));
  ILGenerator ilGenerator =
    newMethod.GetILGenerator();
  ilGenerator.Emit(OpCodes.Ldarg_0);
  ilGenerator.Emit(OpCodes.Ldfld, fieldInfo);
  ilGenerator.Emit(OpCodes.Ret);
  return (Getter<ClassType, FieldType>)
     newMethod.CreateDelegate(
     typeof(Getter<ClassType, FieldType>));
 }
}

struct Field<ClassType, FieldType> {
 public readonly FieldInfo FieldInfo;
 public readonly ClassType Target;
 public readonly Getter<ClassType, FieldType>
   Getter;
 public readonly Setter<ClassType, FieldType>
   Setter;

 public Field(ClassType target, FieldInfo
   fieldInfo) {
  this.FieldInfo = fieldInfo;
  this.Target = target;
  this.Getter = MethodGenerator.CreateGetter
     <ClassType, FieldType>(fieldInfo);
  this.Setter = MethodGenerator.CreateSetter
     <ClassType, FieldType>(fieldInfo);
 }

 public FieldType Value {
  get { return Getter(Target); }
  set { Setter(Target, value); }
 }
}
```

Listing 3. Infrastructure for efficient field access.

4. CONCEPTUAL ANALYSIS

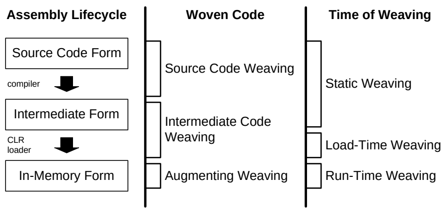

Figure 3. Classification of weaving mechanisms.

Figure 3 shows two orthogonal classifications of weaving mechanisms with regard to an assembly's lifecycle. On the one hand, weaving is classified based on the kind of woven code: *source code weaving* manipulates an assembly's source code to inject aspect or glue code[1]; it can be performed either by a code transformation tool or by a dedicated compiler (plugin). *Intermediate code weaving* manipulates intermediate code (IL and metadata [ECM05]) to inject aspect or glue code; this is usually done by a post-compiler or custom class loader. *Augmenting weaving* does not manipulate existing code, but instead augments it with new glue code, connecting it to aspect code; this can be performed by frameworks rather than tools. Subclass proxy-based approaches are augmenting mechanisms, while traditional approaches like *AspectJ* are source code or intermediate code weaving mechanisms or mixes thereof.

On the other hand, weaving can be classified based on when it is performed: *AspectJ*, for example, has traditionally been a static weaver (including load-time weaving in recent versions) [tAT05]. Contrarily, subclass proxy approaches are real runtime weaving approaches, with the main weaving done at object instantiation time, when the proxy type is created.

[1] *Glue code* is code which "glues" an aspect to its target objects.

With regard to the kind of code being woven, we characterize based on the following properties, displayed in table 2 with advantageous properties in boldface:

Invasiveness is a measure for the degree of manipulation the weaving approach performs on user-written code. Source code weaving approaches compiling a dedicated aspect language have low invasiveness, augmenting approaches only extend and also have low invasiveness. Other approaches change the structure of user-written code and are thus highly invasive.

Debuggability denotes how much effort is needed to make the woven program debuggable with standard mechanisms (e.g. Microsoft Visual Studio). This is easy with proxy-based approaches, because the original debug information remains valid after weaving. Source code weaving also results in correct debug information. With intermediate code weaving, debuggability involves manipulating a debugger-specific file format. This is not portable and often hard: for example, the undocumented *Program Database* file format used by Visual Studio cannot be easily manipulated.

Join point model denotes the join point kinds an approach can provide. Source code weaving makes no restrictions whatsoever to the join point model. With intermediate code weaving, the only restrictions are those posed by IL and metadata (e.g. there are no "for" loops available in IL; to a certain extent this can be overcome by pattern matching as it is also done by decompilers). Augmenting weaving relies on the manipulation mechanisms provided by the CLR, i.e. OOP techniques such as interfaces implementation and method overriding.

Design prerequisites describes prerequisites needed from the perspective of the application designer. With subclass weaving, it is necessary to use a factory to instantiate objects. With source code and intermediate code weaving, there is no such restriction.

Tool prerequisites describes the tools needed for the approach. Source code weaving needs a precompiler or real compiler, intermediate code weaving needs a post-compiler or class loader, augmenting weaving can be done by a framework or library.

Implementation effort is a measure for the effort needed to create a tool based on the approach and keep it up to date with platform changes. Source code weaving requires the most effort by an implementer: it needs at least a source code parser and source code emitter. If the tool is a compiler, complexity is even worse. With intermediate weaving, an IL and metadata parser and emitter are needed, although IL is typically simpler to weave than source code. Augmenting weaving only requires a very simple framework.

Compatibility is a measure for the compatibility of the approach with third-party compilers, frameworks, or metaprogramming tools. With compiler-based source code weaving tools, third-party compilers cannot be used. Intermediate code and augmenting weaving tools pose no compatibility problems and can usually easily be combined with any compiler, framework, or tool.

Language support denotes the number of supported programming languages. While intermediate code and augmenting weaving strategies can handle all programming languages targeting .NET, a source code weaving tool can only target a single programming language. Since one of .NET's goals is to be a multilanguage environment [ECM05], this is an important restriction which might exclude a high number of potential users (much more important than on the Java platform).

Performance is a measure for the runtime efficiency of the approach. With source code weaving, performance is optimal, all compiler optimizations and JIT optimizations can be performed. With IL code weaving, compiler optimizations should be disabled in order to retain a powerful join point model (e.g. target methods must not be inlined by the compiler), but JIT optimizations can be performed without any restriction. With augmenting weaving, some optimizations are disabled by the use of certain OOP features (like virtual method calls), but most JIT optimizations are available.

	Source	**Intermediate**	**Augmenting**
Invasiveness	**low** to high	high	**low**
Debuggability	**no-effort**	hard	**no-effort**
Join point model	**arbitrary**	**IL and metadata**	OOP
Design prerequ.	**none**	**none**	factory
Tool prerequ.	compiler	postcompiler	framework
Impl. effort	very high	high	**low**
Compatibility	low	**high**	**high**
Language support	one	**all**	**all**
Performance	**optimal**	**good**	medium

Table 2. Weaving approaches by code form.

With regard to the time of weaving, we characterize the approaches as follows, summarized in table 3:

Changeability denotes how much effort is needed to add or remove an aspect to or from the application. With static weaving, recompilation or reinstrumentation of the assembly is needed, the application has to be restarted and redeployed. With load-time weaving, the application domain needs to be reloaded, often requiring a restart. With runtime weaving, changes can be applied immediately to objects created after the change.

Deactivating aspects is equally possible in all three weaving variants and requires some sort of join point manager which is asked before a join point is triggered.

Error detection refers to the point of time when weaving configuration errors are detected. With static weaving, this is before application deployment, whereas it is after deployment with the other two approaches.

Testability is inversely proportional to the effort needed to test an object in scenarios with different (or no) aspects attached to it. This follows directly from the changeability: static and load-time weaving require much effort, whereas runtime weaving does not.

	Static	**Load-Time**	**Runtime**
Changeability	recompilation	reload	**immediately**
Deactivating aspects	**immediately**	**immediately**	**immediately**
Error detection	**before depl.**	after depl.	after depl.
Testability	low	low	**high**

Table 3. Weaving approaches by time of weaving.

4.1 Join Point Model

From an AOP perspective, a number of join points can be implemented using subclass proxies, whereas others can't. Table 4 characterizes the join point model realizable with the approach. Using a source code weaving tool, all the join points shown could be realized.

Join Point Type	**Before**	**Instead of**	**After**
Object creation	**yes**	**yes**	**yes**
Constructor execution	**yes**	no	**yes**
Class construction	no	no	no
Object finalization	**yes**	**yes**	**yes**
Method execution	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Method call	no	no	no
Property get	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Property set	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Field get	no	no	no
Field set	no	no	no
Exception thrown	no	no	no
Exception caught	no	no	no
Exception escaping	**yes** (virtual)	**yes** (virtual)	-
Construct (for, if, . . .)	no	no	no

Table 4. Join point model with subclass proxies.

While this join point model is definitely restricted when compared to that of a source code weaving tool, we believe that this is not a problem in most AOP scenarios. When an application is designed from scratch in an aspect-oriented way, all join points are known in advance, before any of the classes or aspects is to be implemented. With a subclass proxy approach, the design would naturally evolve around the join point kinds being available, ignoring those which can't be used. In most cases, however, small design changes can work around the missing join point types.

For example, because field access join points cannot be realized using subclass proxies, a design guideline could be created to access fields via accessor methods (or properties) only, which is a common guideline with OOP already. Those methods whose execution is needed as a join point would be defined to be virtual. The only join points which can't be worked around are: instead-of constructor execution, class construction, exceptions thrown and caught in the same method, and join points at a statement-level granularity. In addition, subclass proxies cannot advise non-virtual methods or distinguish between method call and execution.

4.2 Psychological Factors

Adoption of AOP is hindered by many factors, which are remedied to a great extent by the use of an approach based on subclass proxies:

AOP as an invasive mechanism: AOP is often regarded with distrust, because its tools weave code together as a black box. Developers can't see what happens when aspects and objects are mangled together, concerns about reliability and debuggability (if an error occurs in mangled code, will it be retraceable to the original source code?) as well as the question of unpredictable execution paths in woven code arise. In contrast, subclass proxies are built on established object-oriented concepts such as method overriding and interface implementation. These are well-known, don't introduce reliability or debuggability problems, and developers can comprehend what happens at runtime.

Adaption to new tools: Aspect-oriented tools often replace the tools (e.g. compilers) developers are used to instead of augmenting them. With all approaches, developers need to adapt to new tools with new error messages, longer or different update cycles, and sometimes incompatibilities with the original tools. Since subclass proxies can be implemented as a framework or class library, there is no need to switch tools with such an approach—developers can continue using their familiar environment and still obtain the benefits of AOP.

Unfinished tools: AOP tools usually need a lot of work, this is the cause of the lack of production quality .NET-based AOP tools. However, since subclass proxies are much simpler to implement than code weaving tools, the probability of reaching production status is much higher with this approach.

AOP based on subclass proxies has high adoption potential. With the described prerequisites, users should be easily convincable of the new technology.

5. PERFORMANCE EVALUATION

With proxy-based approaches, aspect code is not directly inserted into the target code; object-oriented mechanisms are used instead. This is often regarded as a performance disadvantage of such approaches. On the .NET platform, however, most optimizations are not done by a language compiler inlining code, but by the JIT compiler's optimizer at runtime. There are some restrictions to JIT optimization with subclass proxies, because virtual method calls to the target object are always performed through the proxy and can't be replaced by ordinary calls, but these apply just as well when the application makes use of the object-oriented mechanisms itself. Most JIT optimizations should not be affected adversely by the use of subclass proxies.

In this section, we will take a look at two implementations of the subclass proxy mechanism—NAspect and DynamicProxy [Ver04] (which XL-AOF is based on)—and analyze object construction time and method call time, since these represent the main points during program flow where a proxy-based mechanism performs differently from a mechanism based on code weaving.

5.1 Object Creation

The first time an object is created from a target type, the proxy-creating factory must construct the new proxy subclass. This is a lengthy operation, our measurings have shown this to take up to 37ms (DynamicProxy) and 12ms (NAspect), as opposed to the few nanoseconds an ordinary new operation (usually) needs. Seen as isolated numbers, this is a tremendous slowdown.

However, analysis of cross-cutting concerns in space-based computing [eKS05a] reveals that common scenarios only have few types being aspectized at the same time, with a higher number of instances created from those. In such scenarios, the generated proxy subclasses can and should be cached, making an instantiation consist of one hashtable lookup plus one call to the type's constructor (either via Reflection or, optimized, via a delegate), which takes a few hundred microseconds at most in our measurements. In the use cases we studied, this makes instantiation time of proxied objects not a problem. On the other hand, if it is vital that proxied objects of many different types are created with rigid performance requirements (a few nanoseconds per instantiation), pure proxying might not be the mechanism of choice, although pooling and flyweight techniques [GHJV95] can improve on that.

Regarding memory usage, an AOP tool based on subclass proxies should use as few dynamic assemblies and modules as possible. Our tests have shown this to scale much better than having one assembly per proxied type. Caching of the generated proxy types will also improve memory footprint. A user should be aware that the only way to remove the generated proxy types from memory is by unloading their application domain (of course, their *instances* are garbage collected as usual), although again this will not be an issue in scenarios with a reasonable number of aspectized types.

5.2 Method Invocation

Method join point performance is more important than object creation performance because the frequency of method calls as compared to object instantiations is typically very high. With subclass proxies, method join points are implemented via method overrides. A method join point of an optimal proxy is therefore no different from a virtual method call (a few nanoseconds) plus one non-virtual base call if delegation to the original code is needed (a few nanoseconds as well). This optimal approach however requires injection of advice code into the subclass proxy, which is not trivial to implement. Current implementations therefore

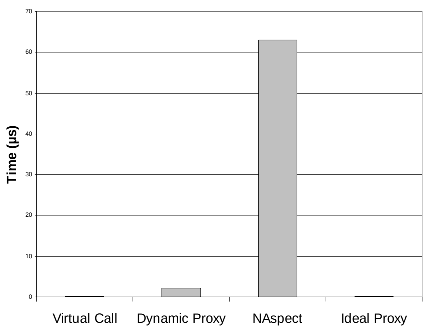

Figure 4. Method call benchmarks.

choose not to directly invoke the base method from within the override. Instead, they encapsulate the base call and hand it to an interceptor provided by the aspect, which may then choose to invoke the method or not. For this encapsulation, DynamicProxy constructs a delegate, whereas NAspect relies on Reflection. Both approaches are not ideal what regards method interception performance, although delegates are an order of magnitude faster than Reflection.

Figure 4 shows a method call benchmark done with an AMD XP1800+ system. The values for ordinary virtual call, DynamicProxy, and NAspect are measured, the value for the ideal proxy is calculated—an implementation can achieve this performance if call times are of much importance. We measured the call and return time of empty methods (with the proxies delegating to the original empty methods); in real scenarios, these values have to be seen in relation to concrete method execution time. For example, our tests have shown that with an average method whose body needs several microseconds for execution, the measured call times are not that significant.

To summarize, while current implementations show medium to significant method call slowdowns, an ideal subclass proxy approach can lead to call times in the range of nanoseconds, not much higher than ordinary method calls. Even the call times of current implementations are less significant if the called methods have nontrivial bodies.

6. CONCLUSION

In this paper, we have motivated, described, and analyzed the subclass proxy mechanism as an implementation infrastructure for aspect-oriented programming. We compared the different proxy mechanisms available on the .NET platform, identifying the subclass proxy mechanism as the most powerful of these. Classifying the weaving approach implementable with subclass proxies, we have shown the disadvantages of the model, such as a more constrained join point model and design restrictions, but have also identified technical advantages over classical implementation mechanisms, such as easy debuggability and runtime weaving capabilities.

Performance benchmarks have shown current subclass proxy implementations to be of medium performance; however the proxy concept could be improved in this regard if necessary in order to achieve call times not much different from ordinary virtual calls.

Analyzing the psychological properties of subclass proxies, we have identified a high potential of the non-invasive mechanism which requires no dedicated compiler tools—we ourselves have successfully used the approach for developing space-based distributed applications [SeK04]. Such light-weight implementations could finally lead to industrial acceptance of aspect-oriented programming.

REFERENCES

[Bur04] Bill Burke. *JBoss Aspect-Oriented Programming (AOP)*, 2004. Available from: `http://www.jboss.org/products/aop`.

[ECM05] ECMA International. Standard ECMA-335 – common language infrastructure (CLI), 3rd edition. Technical report, Ecma International, 2005. Available from: `http://www.ecma-international.org/publications/standards/ecma-335.htm`.

[eKS05a] eva Kühn and Fabian Schmied. Attributes &Co – collaborative applications with declarative shared objects. In *Proceedings WWW/Internet 2005 (Lisbon, Portugal, October 19–22 2005)*, pages 427–434, 2005.

[eKS05b] eva Kühn and Fabian Schmied. XL-AOF – lightweight aspects for space-based computing. In *Proceedings Workshop on Aspect-Oriented Middleware Develoipment (AOMD, Grenoble, France, November 28 2005), Article No. 2*, 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, 1995.

[JH$^+$05] Rod Johnson, Juergen Hoeller, et al. *Spring - Java/J2EE Application Framework*, 2005. Available from: `http://static.springframework.org/spring/docs/1.2.x/spring-reference.pdf`.

[Joh05] Roger Johansson. *NAspect AOP engine*, 2005. Available from: `http://blogs.wdevs.com/phirephly/archive/2005/11/23/11308.aspx`.

[KL$^+$97] Gregor Kiczales, John Lamping, et al. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP 98)*, 1998.

[Low03] Juval Lowy. Decouple components by injecting custom services into your object's interception chain. *MSDN Magazine*, March 2003, 2003.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[Mic06] Microsoft Corporation. *DynamicMethod Class*, 2006. Available from: `http://msdn2.microsoft.com/en-us/library/system.reflection.emit.dynamicmethod.aspx`.

[SeK04] Fabian Schmied and eva Kühn. Distributed peer-to-peer application development with declarative and aspect-oriented techniques. In *Conference Proceedings of International Symposium on Leveraging Applications of Formal Methods (ISoLA, Paphos, Cyprus, October 30 – November 2 2004)*, pages 150–157, 2004.

[tAT05] the AspectJ Team. *The AspectJ 5 Development Kit Developer's Notebook*, 2005. Available from: `http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html`.

[Ver04] Hamilton Verissimo. *Castle's DynamicProxy for .NET*, 2004. Available from: `http://www.codeproject.com/csharp/hamiltondynamicproxy.asp`.