Preserving Separation of Concerns through Compilation

A Position Paper

Hridesh Rajan Robert Dyer Youssef Hanna Harish Narayanappa
Dept. of Computer Science
Iowa State University
Ames, IA 50010
{hridesh, rdyer, ywhanna, harish}@iastate.edu

ABSTRACT

Today's aspect-oriented programming (AOP) languages provide software engineers with new possibilities for keeping conceptual concerns separate at the source code level. For a number of reasons, current aspect weavers sacrifice this separation in transforming source to object code (and thus the very term *weaving*). In this paper, we argue that sacrificing modularity has significant costs, especially in terms of the speed of incremental compilation and in testing. We argue that design modularity can be preserved through the mapping to object code, and that preserving it can have significant benefits in these dimensions. We present and evaluate a mechanism for preserving design modularity in object code, showing that doing so has potentially significant benefits.

1. INTRODUCTION

Aspect-oriented programming (AOP) techniques [9] separate traditionally non-modular (crosscutting) concerns during the analysis, design and implementation phases. In the compilation phase, however, to conform to existing execution models [1], implementations of the crosscutting concerns are scattered and tangled again with the base code (See Figure 1). This loss of separation of concerns is a fundamental problem that leads to complications in post compilation phases.

At the minimum, this problem makes efficient incremental compilation and unit testing of AO programs challenging. The best AO compilers available today take significantly more time compared to their object-oriented counterparts for incremental compilation. A recent report on the application of AspectJ [8] to the development of a J2EE web application for Video Monitoring Services of America showed that incremental compilation using the AspectJ compiler usually takes at least 2-3 seconds longer than near instant compilation using a pure Java compiler [12]. It also showed that if an aspect is changed the incremental compilation resorts to full compilation.

The report observed that due to the increase in incremental compilation time, *human attention can wander and it can take time to re-contextualize after the compilation. This problem is particularly pronounced for the full builds, which tempt the programmer to switch to another task entirely (e.g. email, Slashdot headlines).*

The significant increase in incremental compilation time is because when there is a change in a crosscutting concern, the effect ripples to the fragmented locations in the compiled program forcing their re-compilation. Note that the system studied by Lesiecki [12] can be classified as a small to medium scale system with just 700 classes and around 70 aspects. In a large-scale system, slowdown in the development process can potentially outweigh the benefits of separation of concerns.

Besides incremental compilation, loss of separation of concerns also makes unit testing of AO programs difficult. The dependence of aspects on other classes and vice versa makes it harder to test them separately. AOSD has shown real benefits in its ability to achieve a separation of some traditionally non-modular concerns. In order to continue receiving these benefits in large-scale systems without impeding the design-build-test cycle common in agile development processes, it is essential to address these issues effectively.

The tyranny of the dominant decomposition [23], based on files, modules, classes, in the execution models, and the conformance requirement imposed on AO compilers, affects the efficiency and the complexity of post-compilation phases. The design decision to produce an output that conforms to the existing execution models was perhaps a genuine effort to attract early adopters by lowering the barrier to entry. Now, the potential shown by the technology as well as trends towards industrial adoption [21] warrants investigating the validity of this design decision and its impacts.

This problem is not unique to AOSD; rather it pertains in general to the mechanisms for separation of concerns (SoC) (See Figure 2). Consider an analogy in the procedural abstraction world. In an instruction set architecture (ISA) that does not support method calls, one could still decompose a program into a set of procedures in the analysis, design and implementation phases. The compiler would then translate these programs into a monolithic set of instructions by inlining the procedure bodies.

For these programs, benefits of procedural abstraction such as modular reasoning, parallel development, etc., are observed in analysis, design and implementation phases. In later phases, however, it is difficult to utilize the benefits because there is no clear separation anymore. For example, changes in the source code of a procedure affect all call sites of the procedure, because it is in-lined. Incrementally compiling these procedures is thus harder and more time

[1]By execution model, we mean infrastructure for which a high-level language compiler generates executable code e.g. in the context of AspectJ [8] it would mean the Java Virtual Machine (JVM) [14] and for Eos [20] it would mean the .NET Framework [15].

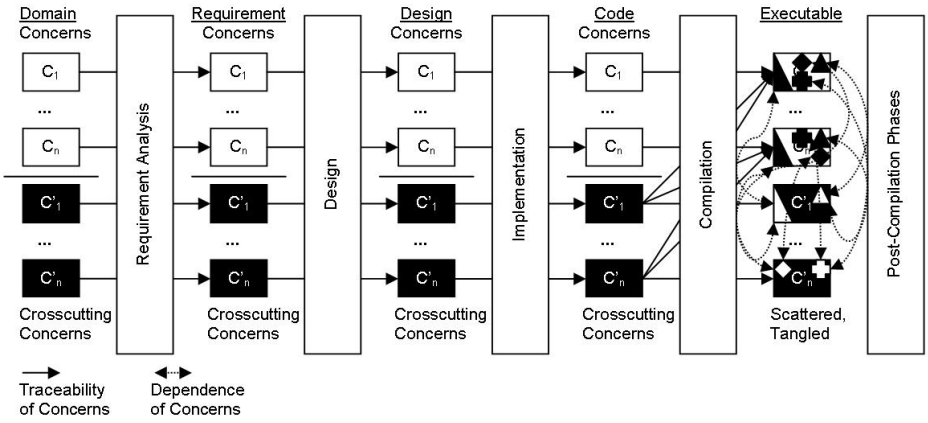

Figure 1: Tracing Concerns through the Life Cycle

Abstraction techniques	Abstraction mechanism at the interface
Procedural Abstraction	Method Call as instruction in ISA
Object-Orientation	Objects and Virtual Method Calls
Aspect-Orientation	An Open Question

Figure 2: Support at the Interface Improved the Benefits of SoC Techniques

consuming. Unit testing such procedures is also a challenge. The support for method call in ISAs, along with the invention and refinement of linking technology has more or less solved these problems for procedural decomposition.

Consider another analogy in the object-oriented world. Like procedural abstraction, object-orientation can be emulated in the analysis, design and implementation phases without the support for objects and dynamic dispatch by translating the OO program into a procedural program that uses methods and structures; however, loosing the traceability of concerns during compilation does affect post-compilation phases. The problem is solved by providing support for *class* and *virtual method calls* abstractions at the interface.

For both SoC techniques, emergence of an abstraction mechanism at the interface between the language compilers and execution models extended the benefits of SoC techniques to post compilation phases (See Figure 2). These mechanisms pushed the decoupling between concerns further down the execution model, abstracting it behind the interface. Encouraged by the history of the past two major SoC techniques in this work we present a new interface between aspect-oriented compilers and execution models. Our interface provides new primitives to represent crosscutting concerns. This interface governs the code that a high-level language compiler generates and the semantics of the primitives that the execution model provides. It abstracts the details of realization of crosscutting mechanisms in the execution model from the language compiler implementation.

In the next section, we provide motivation for our approach. The Section 3 discusses some related work. Section 4 describes our approach in detail. We also present a preliminary evaluation of our approach. Some limitations of our approach and future research directions are discussed in Section 5. Section 6 concludes.

2. MOTIVATION

To motivate our approach, we first demonstrate common static weaving techniques through a simple example application (See Figure 3). We have implemented this application using Eos [7, 18]. Eos is an aspect-oriented extension of C# for Microsoft .NET Framework [15] that implements the unified-aspect model proposed by Rajan and Sullivan [20]. Rajan and Sullivan showed that the AspectJ notions of aspect and class can be unified in a new module construct that they called the classpect, and that this new model is significantly simpler and able to accommodate a broader set of requirements for modular solutions to complex integration problems [22].

The binding construct in this model allows modularization of crosscutting concerns. A binding is a mechanism to select a subset of join points in the execution of the program and associate a method to execute at those points. The subset of join points selected by the binding are called subjects of the join point. The method that is associated by the binding to execute at these join points is called the handler of the binding.

Our application has two classpects: *HelloWorld* (shown inside the white box) and *Trace* (shown inside the grey box). The classpect *Hello* declares a method *Main* that prints the string *Hello* on the screen and exits. The classpect *Trace* declares a pointcut *traceMethods* to select all method execution join points in the program and a static binding. The effect of declaring the binding is that the handler method *trace* is invoked at all the subject join points selected by the pointcut *traceMethods* and prints the string *trace() called*. As a result, after the execution of the method *Main* the string *trace() called* is printed.

We compiled this simple application using the Eos compiler [19]. We disassembled the assembly [2] using *ildasm*, the disassembler for .NET Framework. Figure 4 shows the dis-

[2]Assembly is a .NET Framework term for an executable.

```
public class Hello{
  static void Main(string[] arguments){
    System.Console.WriteLine("Hello");
  }
}
```

```
public class Trace{
 pointcut traceMethods(): execution(any any(..));

 static after traceMethods(): trace();

 void trace(){
      System.Console.WriteLine("trace() called");
 }
}
```

Figure 3: A Simple Aspect-Oriented Application

assembly. We have used the common intermediate language (CIL) notations to represent the disassembly. Please note that the weaving techniques for static bindings is similar to that of AspectJ-like languages [8], so the intermediate code shown in Figure 4 is a representative of the current weaving techniques.

The Figure shows the disassembly of *Hello* in the white box and the disassembly of the *Trace* in the grey box. As can be observed, the intermediate code to invoke *Trace* at join points is inserted into the class *Hello* in the method *Main*. As a result, the concern modularized by the classpect *Trace* ends up being scattered and tangled with the *Hello* concern. This scattering and tangling directly affects incremental compilation. For example, let us assume that the source code for the classpect *Trace* changes, so that it now selects all execution join points where the method name begins with a *Set* for example SetX, SetY, etc. The method *Main* in the *Hello* class is no longer selected by this pointcut for advising.

This change will trigger the incremental compilation of *Trace*. In addition, it will also trigger the compilation of *Hello* to reflect the changes in the pointcut. The full recompilation of this simple system is not a huge burden on the program; however, in nontrivial systems the overhead of compilation can be significant enough to disrupt the build-test-debug cycle common in current agile software development processes. The re-compilation time is affected by two factors. First, increase in the number of crosscutting concerns in a large-scale system. Second, increase in the number of modular concerns that these crosscutting concerns are scattered and tangled with. For a change in a crosscutting concern such as tracing or logging, recompilation of the entire system will be necessary.

We attribute this increase in incremental compilation time to the scattering and tangling of the concerns in the intermediate code. If the separation of concerns would have been preserved in the intermediate code, it would have been sufficient to just recompile the changed concerns in the system. The nature of the concerns modularized by aspect-oriented techniques dictates that they execute at scattered and tangled points in the execution of the program. The loss of separation at runtime thus seems unavoidable; however, we argue and show through preliminary results that we can do better.

```
.method private static void Main(string[] args)
   cil managed {
.entrypoint  .maxstack 1
   stloc.0
  .try{
    ldstr "Hello"
    call void System.Console::WriteLine(string)
    leave.s IL_0032
  }finally{
    call class Trace Trace::GetStatic()
    ldloc.0
    callvirt instance void Trace::traceMethods()
    }
  ret
}
```

Crosscutting code mixed with the intermediate code of the class Hello

```
.method private hidebysig specialname rtspecialname
  static void .cctor() cil managed {
   … /* Construct a static instance */
}
.method public static class Trace GetStatic()
  cil managed {
   … /* Return the static instance */
}
.method public instance void trace ()
  cil managed{
  .maxstack  1
  ldstr "Trace::trace() called"
  call void System.Console::WriteLine(string)
  ret
}
```

Figure 4: Disassembled HelloWorld

3. RELATED WORK

Three closely related and complimentary research ideas are run-time weaving, load-time weaving and virtual machine support for aspect-oriented programming. We will discuss these ideas in detail below.

There are several approaches for run-time weaving such as PROSE [17], Handi-Wrap [4], Eos [19], etc. A typical approach to runtime weaving is to attach hooks at all join points in the program at compile-time. The aspects can then use these hooks to attach and detach at run-time. An alternative approach is to attach hooks only at potentially interesting join points. In the former case, aspects can use all possible join points, excluding those that are created dynamically so the system will be more flexible. The disadvantage is the high overhead of unnecessary hooks. In the later case, only those aspects that utilize existing hooks can be deployed at run-time, but the overhead will be minimal for a runtime approach.

Eos uses the first model, i.e. only instrument the join points that may potentially be needed. Handi-Wrap uses the second model, making all join points available through wrappers. PROSE indirectly uses the second model, exposing all join points through the debugger interface. PROSE allows aspects to be loaded dynamically without restarting the system. An additional advantage of indirectly exposing join points through debugger interface is that new join points (created by reflection) are registered automatically.As observed by Popovici et al. [17] and Ortin et al. [16], however, performance in both cases is a problem.

A load-time weaving approach delays weaving of crosscutting concerns until the class loader loads the class file and defines it to the virtual machine [13]. Load-time weaving approaches typically provide weaving information in the form of XML directives or annotations. The aspect weaver then

revises the assemblies or classes according to weaving directives at load-time. Often a custom class loader is needed.

There are load-time weaving approaches for both Java and .NET framework. For example, AspectJ [8] recently added load-time weaving support. Weave.NET [11] uses a similar approach for .NET framework. The JMangler framework can also be used for load-time weaving [10]. It provides mechanisms to plug-in class-loaders into the JVM.

A benefit of the load- and run-time weaving approaches is that they delay weaving of aspect-oriented programs. It may be possible to improve incremental compilation using these approaches, although we do not currently have any evidence to confirm or to deny. A contribution of our approach might also be perceived as delaying weaving, however, we view the interface and corresponding contracts between the language designs and execution model designs as the main contribution of our work. The load-time weaving approaches do not provide these benefits.

Steamloom [5] extends the Jikes Research Virtual Machine (RVM), an open source Java virtual machine [1]. Traditional approaches for supporting dynamic crosscutting involve weaving aspects into the program at compilation. Steamloom moves weaving into the Virtual Machine (VM), which allows preserving the original structure of the code after compilation and shows performance improvements of 2.4 to 4 times when compared to AspectJ. It accomplishes this by modifying the Type Information Block to point methods to a stub that modifies the existing bytecode to weave in the advice.

Our approach and Steamloom are in some sense complimentary. Similar to Steamloom, our approach also advocates support for crosscutting in the execution models. Steamloom investigates techniques to improve the performance of these crosscutting mechanisms provided by the execution model, whereas, our approach focuses on separating the compiler implementations and execution model implementations by defining an interface between the two. Our focus is on providing the basic mechanisms at the interface that can be used as primitives by compiler implementations. Our approach thus potentially allows multiple language models to use the same VM and/or multiple VMs. Each of these VMs may have their own method of weaving.

Steamloom, however, restricts the type hierarchy of aspects. An aspect must inherit from a special class. In languages like Java, this restriction burns the only available inheritance link. Our approach does not impose any restrictions on programming language constructs, leaving those design decisions to programming language designers and compiler implementers.

In the next section, we will discuss our approach to preserve separation of concerns through the compilation process. We will then discuss the implementation of the HelloWorld application using our approach.

4. OUR APPROACH AND PRELIMINARY EVALUATION

We have developed an improved approach to aspect-oriented compilation. The basis of our approach is a new interface to represent crosscutting abstractions between the HLL compiler implementation and the runtime environment. This interface abstracts the realization of the crosscutting behavior at run-time from the HLL compiler implementations. The interface governs the intermediate code that the HLL compiler can generate and the semantics of the crosscutting primitives that it can expect.

bind	1. Pops top two values from the stack: pattern and delegate 2. Semantics: After this atomic instruction is complete, for every join point matching the pattern, the delegate is invoked.
remove	1. Pops top two values from the stack: pattern and delegate 2. Semantics: After this atomic instruction is complete, if there was a binding between the delegate and the pattern, it is removed.

Figure 5: Extensions to the combined intermediate language (ECIL)

Our interface provides two primitives, *bind* and *remove*, in the form of extended CIL instructions (See Figure 5). We call this new intermediate language *ECIL* for extended CIL. The *bind* and *remove* instructions abstract association and disassociation of advice with join points respectively. The *bind* instruction expects the delegate to a method and the pattern to select join points as the top two items on the stack. The pattern is equivalent to a pointcut expression. The effect then is to associate the delegate to execute at all join points selected by the pattern. Similarly, the *remove* instruction takes the delegate and the pattern and puts an end to the association between the join points selected by the pattern and the delegate.

To illustrate, let us revisit our *HelloWorld* example. The intermediate code for the application in ECIL is shown in Figure 6. Like before, the disassembly of classpect *Hello* is shown in the white box and the disassembly of classpect *Trace* is shown in the grey box.

Instead of explicit callbacks in the intermediate code for *Hello*, a set of association instructions are generated in the intermediate code for *Trace*. Note that we are translating a static binding that affects all instances of *Hello*. To model the semantics of static binding, a set of association instructions are inserted in the static constructor of *Trace*. The constructor pushes a delegate to the method *trace* on to the stack followed by the string *after execution (any.any(..))*. The *bind* instruction follows these two push instructions. As a result, when the type *Trace* is initialized, the handler *trace* is associated to execute at the selected subject join points.

Our example demonstrates that the separation of concerns is preserved for modules represented in ECIL. The code for *Hello* is free of the callbacks to the *trace* method in the *Trace* classpect. As a result a change in *Hello*, which is not a crosscutting concern, will only affect the intermediate code representation of the *Hello* module. Similarly, a change in *Trace*, which is a crosscutting concern, will only affect the intermediate code representation of the trace module. The changes are thus traceable to a limited number of modules at the intermediate code level, resulting in an improved incremental compilation time compared to existing approaches.

The example we presented above demonstrates static weaving. The bind/remove primitives can also be used for runtime weaving (See Figure 7). The figure shows a variation of our *HelloWorld* application. Now we want to enable and disable tracing at runtime. To do that, the modified classpect *Trace* provides two methods, *On* and *Off*. The

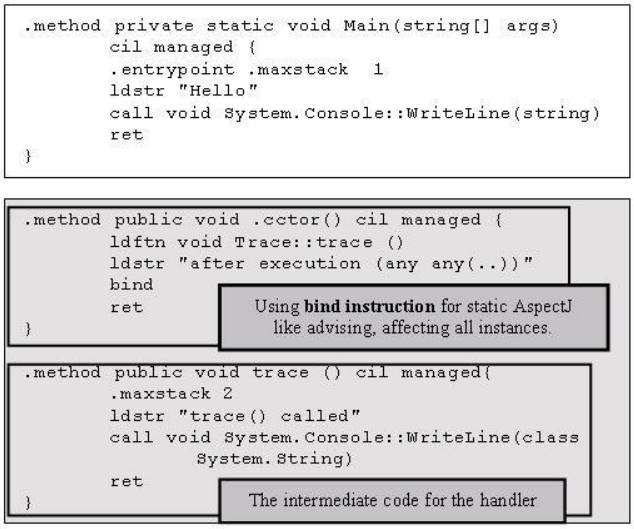

Figure 6: ECIL version of HelloWorld

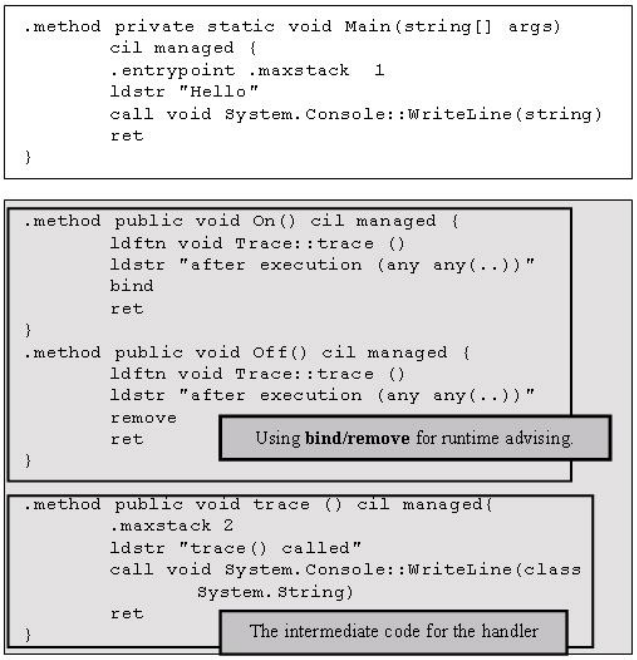

Figure 7: Implementing Runtime Advising in ECIL

ECIL representation of method *On* consists of instructions to push the delegate to the handler and the pattern on the stack followed by the *bind* instruction (similar to the static constructor implementation in the static tracing example). The method *Off* also consists of instructions to push the delegate and the pattern followed by the *remove* instruction. The semantics of *bind* and *remove* ensures that calling *On* activates the tracing and calling *Off* deactivates it.

We argue that providing abstractions to represent crosscutting concerns at this level has two potential benefits. First, it preserves the separation of concerns at the intermediate code level. Second, it allows for separate development of aspect-oriented compilers and runtime environments using and supporting the new interface. We also showed that both static and runtime advising are supported.

Please note that at this time we have explicitly decided not to support static crosscutting mechanisms such as inter-type declarations. There are two reasons behind this design decision. First, in most inter-type declarations there is a one-to-one explicit mapping between the classes and the aspects. Therefore, the impact of changes in the aspect can be statically traced to a very small number of classes. As a result, the increase in compilation time is not significant. Second, inter-type declarations can be emulated using partial classes in C# version 2.0 [6].

5. LIMITATIONS AND FUTURE WORK

The current version of our interface has at least two known limitations. First, we have not yet developed a statically verifiable and type-safe mechanism to support various different types of join points and pointcut expressions. Currently, the pattern is specified as a string. For example, a string *execution(any any.Set(..))* will select all method execution join points for which the method name is *Set*. The correctness of this string cannot be checked statically at the intermediate code level. However, it doesn't prevent the HLL compilers from providing an AspectJ-like pointcut sublanguage, statically verifying the pointcut expression supplied by the user and then generating the verified pointcut as a string in the intermediate code. Second, current version of our interface does not support instance-level weaving [19]. Future extensions will address these limitation.

In the future, we plan to complete three tasks. First, we will address the limitations, mentioned above and those pointed out by the workshop participants, in the design of our interface. Second, we plan to design and implement an extension to the Shared Source Common Language Infrastructure (SSCLI) [3] to support ECIL (for Extended CIL) to provide an example interface in Microsoft .NET context. In particular, the just in time (JIT) compiler will be extended to interpret ECIL. We plan to use the Phoenix infrastructure [2] to ease development of this extended JIT compiler. The Eos compiler [7] will also be extended to generate ECIL.

6. CONCLUSION

In this work, we argued that the loss of separation of concerns during compilation makes post-compilation phases challenging. We proposed a solution in the form of an interface between the high-level language compiler implementations and the execution models. Our interface specifies the intermediate code that the compiler can generate and the semantics of crosscutting primitives it can expect. We further demonstrated through a simple example that both static and runtime weaving can be expressed in terms of these primitives. We also observed that separation of crosscutting concerns is preserved in the intermediate modules that use these primitives. We speculate, but have not systematically tested yet, that preserving the separation of concerns will significantly improve the incremental compilation time.

Contingent upon successfully addressing the remaining challenges, we hypothesize that this research might make two fundamental advances in the theory of AOSD. First, it may demonstrate that preserving separation of concerns, as

far out in the software development process as possible, has tangible benefits, and second, it may provide an improved compilation and execution model for AOP. We claim that it might have at least three tangible benefits. First, the improved aspect compilation and execution model will considerably decrease the incremental compilation time. Second, it will ease unit testing of aspect programs. Third, it will significantly reduce the complexity of aspect-oriented compilers making new language implementations easier.

The decoupling between language compilers and the execution model achieved by our interface also has the potential to enable independent research in these areas. Simpler aspect language designs and compiler implementations might be realized without spending significant time on the optimization of the underlying AO execution models. Novel optimization mechanisms for the underlying execution models can be developed independent of the language design as long as it conforms to the interface.

7. REFERENCES

[1] The Jikes research virtual machine (RVM). http://jikesrvm.sourceforge.net/.

[2] Phoenix. http://research.microsoft.com/phoenix/.

[3] The shared source common language infrastructure (SSCLI). http://research.microsoft.com/sscli/.

[4] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 86–95, New York, NY, USA, 2002. ACM Press.

[5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[6] ECMA. *Standard-334: C# Language Specification*, 2002.

[7] Eos web site. http://www.cs.iastate.edu/ẽos.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[9] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–90, Boston, Massachusetts, 17–23 May 1997. IEEE.

[10] G. Kniesel, P. Costanza, and M. Austermann. Jmangler-a framework for load-time transformation of java class files. In *1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), 10 November 2001, Florence, Italy*, pages 100–110. IEEE Computer Society, 2001.

[11] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[12] N. Lesiecki. Applying AspectJ to J2EE application development. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, New York, NY, USA, 2005. ACM Press.

[13] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–44, New York, NY, USA, 1998. ACM Press.

[14] T. Lindholm and F. Yellin. Addison-Wesley, Reading, MA, USA, 1997.

[15] Microsoft Corporations. *Microsoft .NET*, 2001. URL: http://www.microsoft.com/net.

[16] F. Ortin and J. M. Cueva. Dynamic adaptation of application aspects. *Journal of Systems and Software*, 71(3):229–243, 2004.

[17] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[18] H. Rajan. *Unifying Aspect- and Object-Oriented Program Design*. PhD thesis, The University of Virginia, Charlottesville, Virginia, Aug. 2005.

[19] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[20] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[21] D. Sabbah. Aspects: from promise to reality. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 1–2, New York, NY, USA, 2004. ACM Press.

[22] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–68, July 1992.

[23] P. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimentional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.