2005

Classpects in Practice: A Test of the Unified Aspect Model

Hridesh Rajan
Iowa State University

Kevin Sullivan
Iowa State University

Classpects in Practice: A Test of the Unified Aspect Model

Abstract

The most successful model of aspect-oriented modularity to date is that embodied in the AspectJ language. We have shown that the AspectJ notions of aspect and class can be unified in a new module construct that we 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. We embodied our unified model in the Eos language design, in which the basic unit of modularity is a *classpect* ; and we realized the model in a concrete and usable form in the Eos compiler. The main contribution of this paper is a fairly demanding experimental evaluation of the Eos component model, language, and compiler in terms of their application to two significant systems: ConcernCov, a tool for concern-based code coverage analysis of test suites (20K LOC), and the Eos compiler, a near-industrial strength *classpect-oriented* extension to the CSharp language (50K LOC). Our assessments of the resulting designs provides evidence for the potential design structuring benefits of the Eos model, the usability of the Eos language, and the practical utility of our language implementation in the Eos compiler. In a nutshell, we contribute a demonstration of the immediate practical value of our conceptual work.

Disciplines

Programming Languages and Compilers

Classpects in Practice: A Test of the Unified Aspect Model

Hridesh Rajan
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50010, USA
hridesh@cs.iastate.edu

Kevin Sullivan
Dept. of Computer Science
University of Virginia
151 Engineer's Way
Charlottesville, VA, 22904, USA
sullivan@cs.virginia.edu

ABSTRACT

The most successful model of aspect-oriented modularity to date is that embodied in the AspectJ language. We have shown that the AspectJ notions of aspect and class can be unified in a new module construct that we 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. We embodied our unified model in the Eos language design, in which the basic unit of modularity is a *classpect*; and we realized the model in a concrete and usable form in the Eos compiler. The main contribution of this paper is a fairly demanding experimental evaluation of the Eos component model, language, and compiler in terms of their application to two significant systems: ConcernCov, a tool for concern-based code coverage analysis of test suites (20K LOC), and the Eos compiler, a near-industrial strength *classpect-oriented* extension to the C# language (50K LOC). Our assessments of the resulting designs provides evidence for the potential design structuring benefits of the Eos model, the usability of the Eos language, and the practical utility of our language implementation in the Eos compiler. In a nutshell, we contribute a demonstration of the immediate practical value of our conceptual work.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-Oriented Design Methods*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.2.10 [**Software Engineering**]: Design—*Methodologies*

General Terms

Experimentation, Languages, Measurement

Keywords

Classpect, Aspect-Oriented, Unified Language Model, Binding, Eos, Case Studies, Concern Coverage Analysis

1. INTRODUCTION

In prior work, we showed that the notions of aspect and class in the most successful model of aspect-oriented modularity to date, namely the AspectJ model [18], can be unified in a new module construct that we called the *classpect* [28]. We also showed that this new model is significantly simpler, more compositional, and able to accommodate a broader set of requirements for modular solutions to complex integration problems [30, 31].

We embodied our unified model in the Eos language design [26], in which the basic unit of modularity is a *classpect*; and we realized the model in a concrete and usable form in the Eos compiler. We demonstrated the benefits of the unified language design in the context of small but representative examples. These demonstrations did provide some basis for further speculations about the language model's potential to solve large-scale problems. The representative examples, however, are insufficient to answer the skeptical software practitioner's questions: Are these benefits observed in significant software systems? What are the other problems (if any) that one may run into?

The main contribution of this paper is a fairly demanding experimental evaluation of the Eos component model, language, and compiler in terms of their application to two significant systems: ConcernCov, a tool for concern-based code coverage analysis of test suites (20K LOC) [27], and the Eos compiler, a near-industrial strength *classpect-oriented* extension to the C# language (50K LOC) [26]. We applied the new language model extensively towards the experimental re-design of the Eos compiler. The compiler presents multiple opportunities where the new language model results in an improved modularization. In the redesign of the Eos compiler, the unified model was applied after–the–fact, whereas *ConcernCov* was designed using Eos to begin with. As a result, the experiences in both cases were different.

These case studies show that, the unified model and the supporting infrastructure is robust enough for and scalable to the real world systems and provides similar benefits as in the representative examples. Applying the unified language design to the real world systems also showed some practical problems with the Eos language infrastructure. At the end of this paper, we describe these problems. Our assessments of the resulting designs provide evidence for the design structuring benefits of the Eos model, the usability of the Eos language, and the practical utility of our language implementation in the Eos compiler. In a nutshell, we contribute a demonstration of the immediate practical value of our conceptual work.

```
1 aspect Tracing {
2   pointcut tracedCall():
3       execution(* *(..)) && !within(Tracing);
4   before(): tracedExecution() {
5       /* Trace the Execution */
6   }
7 }
```

Figure 1: A Simple Example Aspect

The rest of this paper is organized as follows. The next section gives some background on aspect-oriented programming and the unified aspect-language model. Section 3 describes the ConcernCov case study. Section 4 describes some of the scenarios in which unified model was beneficial in the Eos compiler redesign. Section 5 describes some limitations of the approach discovered by the case studies. Section 6 reflects on the results of the case study. Section 7 discusses related work and Section 8 concludes.

2. BACKGROUND

In this section, we briefly review the AspectJ and unified language model embodied by Eos. The focus is on their key differences. The AspectJ language model is described in detail by Kiczales et al. [18]. A complete language manual and compiler is available from the AspectJ web site [1]. The unified language model is described in detail by Rajan and Sullivan [28] and the Eos compiler is available from the Eos web site [6].

2.1 The AspectJ Language Model

In this subsection, we will review basic concepts in the dominant aspect-oriented model, namely the AspectJ model. AspectJ [18] is an extension to Java [10]. The rest of this paper starts with this model. Other languages in this class include AspectC++ [29], AspectR [4], AspectWerkz [2], AspectS [16], Caesar [22], etc. While Eos [26] is not AspectJ-like, it is in the broader class of Pointcut-Advice-based AO languages [20]. The central goal of AspectJ-like languages is to enable the modular representation of crosscutting concerns.

AspectJ-like languages organize programs into a two-layered hierarchy. The concerns that can be modularized using the traditional OO modularization techniques, classes, are in the first layer. The first layer is also often called the base layer [19]. The modularized representation of the crosscutting concerns, aspects, are in the second layer. These aspects affect the behavior of the classes in the base layer.

These languages add five key constructs to the object-oriented model: join points, pointcuts, advice, inter-type declarations, and aspects. (For the purpose of this paper, inter-type declarations are not relevant as they remain unchanged in the unified model.) A simple example is shown in Figure 1 to make the points concrete.

An aspect (lines 1-7), modifies the behavior of a program at certain selected execution events exposed to such modification by the semantics of the programming language. These events are called join points. The execution of a method in the program in which the Tracing aspect appears is an example of a join point. A pointcut (lines 2-3) is a predicate that selects a subset of join points for such modification – here, execution of any method outside the Tracing aspect. An advice (see lines 4-6) is a special *method-like* constructs that effect such a modification at each join point selected by a pointcut. An aspect (lines 1-7) is a class–like module that uses these constructs to modify behaviors defined by the classes of a software system.

```
1 class Tracing {
2   pointcut tracedExecution():
3       execution(* *(..))&& !within(Trace);
4   static before tracedExecution(): Trace();
5   public void Trace() {
6       /* Trace the call */
7   }
8 }
```

Figure 2: A Simple Example Classpect

Like classes, aspects also support the data abstraction and inheritance, but they do differ from classes. First, aspects can use pointcuts, advice, and inter-type declarations. In this sense, they are strictly more expressive than classes. Second, instantiation of aspects and binding of advice to join points are wholly controlled by the Aspect language runtime. There is no *new* for aspects. Aspect instances are thus not first-class, and, in this dimension, classes are strictly more expressive than aspects. Third, although aspects can advise methods with fine selectivity, they can select advice bodies to advise only in coarse-grained ways.

2.2 The Unified Language Model

Rajan and Sullivan addressed the limits of aspects with respect to instantiation and join point binding under program control [26], but they left aspects and classes separate and incomparable, and the resulting compositionality problems unresolved. They tackled this problem in the following work [28], leading to a unified language design in which advising emerged as a general alternative to quantified overriding or method invocation.

The new language model unifies aspects and objects in three ways. First, it unifies aspects and classes as *classpects*. A *classpect* has all the capabilities of classes, all of essential capabilities of aspects in AspectJ–like languages, and the extensions to aspects needed to make them first class objects. Second, the unified model eliminates advice in favor of using methods only, with a separate and explicit join-point-method binding construct. Third, it supports a generalized advising model. To the usual object-oriented mechanisms of explicit or implicit method call and overriding based on inheritance the unified model adds implicit invocation using before and after advice, and overriding using around advice, respectively.

To make these points concrete we revisit the example presented in the previous section in Figure 2. A classpect (lines 1-8), similar to the aspect in the previous section, declares a pointcut (lines 2-3) to select the execution of any method and then composes it with the *within(Trace)* pointcut expression to exclude the methods in itself to avoid recursion. A static binding (line 4) binds the method Trace (lines 5-7) to execute before all join points selected by the pointcut tracedExecution. Note that by binding statically join points in all instances are affected. A non-static binding would bind to instances selectively. The key difference in this implementation is that all concerns are modularized as classpects and methods. The crosscutting concerns, however, uses bindings to bind the method containing the implementation of the crosscutting concerns to join points.

The AspectJ dichotomy between object-oriented and aspect modules, with the latter able to advise the former but not vice-versa promotes a two-level, asymmetric design style. AspectJ does provide limited mechanisms for aspects to advise other aspects, so two-level architecture is not strictly enforced, but it is awkward to write aspects that advise aspects because advice is not easily named. We view advising as a general module composition mechanism and believe that there is value in supporting it as a first-class mechanism. The next few sections demonstrate that the unification shows benefits in canonical examples and real systems.

3. CASE STUDY 1: CONCERNCOV

In our earlier work, we discuss a new approach for code coverage analysis, namely concern coverage, and a supporting tool, *ConcernCov* [27]. *ConcernCov* was build by integrating back-end implementation of the concern coverage analysis logic with the *NUnit* GUI implementation [11]. One of the goals of this project was to keep the back-end implementation name-independent from GUI implementation and vice-versa so that no modification in the original code of *NUnit* is required to fit the new tool. The unified model of Eos was able to achieve the objectives successfully. In the rest of this section, we discuss the challenges and the solution presented by Eos.

3.1 Adding GUI Elements Transparently

ConcernCov uses the GUI interface of *NUnit* as the front-end. As shown in Figure 3, it was necessary to add two GUI elements, a new content tab and a button, to the *NUnit* interface. To allow simultaneous testing of more than one assembly *ConcernCov* creates one instance of the *NUnit* front-end for each assembly. [1] If the coverage information is required for an assembly, new GUI elements are added to the *NUnit* instance that is being used to test the assembly. This scenario requires selective advising of object instances. In the absence of selective instance advising as a language feature one could use workarounds, however, as we have shown previously these workarounds are unnecessarily complex and often incur needless performance overheads [26]. The unified language model, on the other hand, supports selective advising of object instances.

We added these GUI elements transparently to the *NUnit* interface using inter-type declarations as shown in the classpect in Figure 4. The classpect *GUIMixin* (Lines 1-66) adds three fields, a button (Line 5), a tab page (Line 3) and a text box (Line 4), to the main GUI form of *NUnit* using inter-type declarations. It also adds two methods, *CInit* (Lines 7-12) and *CInitComponents* (Lines 13-47), to perform additional initializations required by these new controls. These methods also add the new GUI elements to the form container to display them at runtime.

A .NET Framework form initializes its GUI components in a standard function called *InitializeComponent*. For the *NUnitForm* instances corresponding to the assemblies that

[1] An assembly is a Microsoft .Net [23] equivalent of an executable

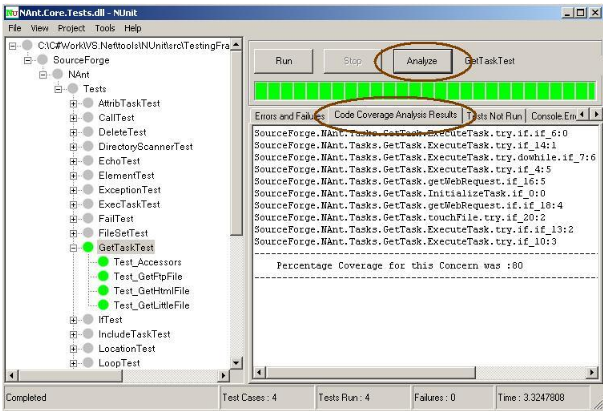

Figure 3: A Screen-shot of *ConcernCov* in Action

require coverage information additional initializations performed by *CInit* and *CInitComponent* are also necessary. The *GUIMixin* classpect uses two bindings (Lines 53-54, and 58-59) to make sure that the method *CInit* executes after the constructor of the form, and the method *CInitComponents* executes after the execution of *InitializeComponent*.

These bindings are non-static bindings. If we recall, a non-static binding in Eos binds the handler method to object instances selectively. The selective binding to instances is performed by calling the implicit method *addObject*. The effect of calling the *addObject* method with an argument object instance is to register the handler method with that instance. Here, the method *AddGUIElements* (Line 63-65) takes an instance, *form*, of the *NUnit* front end, *NUnitForm*, as argument and calls the implicit method *addObject* with *form* as an argument (Line 64). The net effect is to initialize and display the new GUI components for a *NUnitForm* instance selectively.

3.2 Collecting Coverage Information

The assembly version 2.1.4 of *NUnit* uses static member *outWriter* and *errWriter* of the class *NUnit.UiKit.AppUI* throughout the execution as standard output and standard error respectively. The *AppUI.Init* function initializes these members. The *NUnit* form calls these functions to redirect the output and error from test cases to specific content tabs designed in the form. In the initial versions of *ConcernCov*, the tool instrumented the executable under test such that it would output information for coverage analysis at each join point on the standard output. The coverage tool then filters the standard output for coverage information, collects it, and passes the rest to the *NUnit* form. The classpect *IntroduceFilter* shown in Figure 5 does this filtering without modifying the *NUnit* code. Note that this filtering is required only for *NUnit* form instances that are being used to test assemblies for which coverage information is required, necessitating selective advising of form instances.

The classpect *IntroduceFilter* (Lines 2-16) binds the method *Replace* (Lines 10-15) around the execution of *AppUI.Init* function. The method *Replace*, constructs a coverage information collector using the text writer supplied as argument to the *AppUI.Init* function, and passes the coverage information collector as the argument to the original function, thereby introducing a filter between the *AppUI* text writers and the form text writers to collect coverage

```
public class GUIMixin{
  introduce in NUnit.Gui.NUnitForm {
    public TabPage coverage;
    public RichTextBox coverageTab;
    public Button resultButton;
    public void CInit(){
       this.coverage.SuspendLayout();
       coverageTab.Enabled = true;
       resultButton.Enabled = false;
       this.resultTabs.Controls.Add(this.coverage);
       this.groupBox1.Controls.Add(this.resultButton);
     }
    public void CInitComponents(){
      /* Construct and initialize the tab page,
            the tab text box, and the coverage button*/
     }
    private void resultButton_Click(object sender,
                                 System.EventArgs e){
       coverageTab.Clear();
     }
 }
 after objectinitialization( public NUnitForm(..))
       && this(form):CallCInit(NUnitForm form);
 public void CallCInit(NUnitForm form){
          form.CInit();
          }
 after execution(private NUnitForm.InitializeComponent())
    && this(form):CallCInitComponents (NUnitForm form);
 public void CallCInitComponents(NUnitForm form){
          form.CInitComponents();
          }
 public void AddGUIElements(NUnitForm form){
          this.addObject(form);
          }
}
```

Figure 4: Adding GUI Elements Transparently

```
using Eos.Runtime;
 public class IntroduceFilter{
  public static CoverageInfoCollector coverageCollector;
  void around execution(public static void
     NUnit.UiKit.Init(Form, TextWriter, TextWriter) &&
     args(Form, tW, TextWriter)) && aroundptr(p)
     && joinpoint(jp):
     Replace(TextWriter tW, IJoinPoint jp, AroundADP p);

  void Replace(TextWriter tW,
     AroundADP p, IJoinPoint jp) {
     coverageCollector = new CoverageInfoCollector(tW);
     jp.Args[1] = coverageCollector;
     p.InnerInvoke(); // Invoke the underlying join point
     }
  }
```

Figure 5: Collecting Coverage Information

```
public class FormCovColMediator{
   public FormCovColMediator(
        NUnit.Gui.NUnitForm form,
        CoverageInfoCollector collector){
     this.form = form;
     this.collector = collector;
     addObject(form);
     }
   NUnit.Gui.NUnitForm form;
   CoverageInfoCollector collector;
   after execution(private void
       NUnit.Gui.NUnitForm.resultButton_Click(...)):
          Display();
   public void Display(){
       collector.Display(
             new TextBoxWriter(form.coverageTab ));
       }
 }
```

Figure 6: Displaying Coverage Information

information.

3.3 Displaying Coverage Information

The next integration requirement was to display the coverage information when the result button is pushed. The components involved in this case were the coverage tab, the coverage collector, and the result button. The coverage tab already implements the text writer interface, so we implemented coverage collector to take a text writer interface as input to emit coverage information. If the coverage collector was to be integrated after–the–fact transforming the coverage output form to the display form might have been necessary.

Another classpect FormCovColMediator shown in Figure 6 fulfils this requirement. This classpect provides a constructor that takes a form reference and a coverage collector reference as argument. It stores the references and registers itself with the form using the implicit method *addObject*. It also provides a binding that selects the execution of the method *resultButton_Click*. When a used clicks the result button, the .NET environment run-time automatically calls the method *resultButton_Click*. The binding binds a method *Display* to the method *resultButton_Click*. The method *Display* calls the method *Display* on coverage collector instance, passing it the text box writer constructed from the coverage tab.

Finally, to put everything together when *NUnit* is invoked, the control and supplied arguments must be passed to the coverage tool to allow it to filter out its argument, compile and instrument the source code according to the *ConcernCov* directives. This integration was also done using a classpect. This classpect, binds a method, *InitCodeCoverage*, before the execution of the entry method (*Main*) of *NUnit*. This method constructs an instance of the coverage tool, and passes the arguments to the method *Main* to the tool. After returning from *InitCodeCoverage*, the method *Main* of NUnit proceeds normally.

The assessment discussed in this section showed that the unified model, the Eos language, and the Eos compiler improved the modularization of challenge concerns in the *ConcernCov* tool. Each challenge concern that we discussed is a representative of a different type of design structure observed in software systems. For example, selective addition of GUI elements is an example of a mixin [3], coverage information collection is an example of a pipe-filter based architecture [9], and coverage display is an example of an integration concern [31].

We were able add the required GUI elements to the *NUnit* interface transparently, without having to modify the *NUnit* source code. The classpect *GUIMixin* provides an interface to selectively modify a *NUnitForm* instance to display GUI elements related to concern coverage analysis when necessary. We were able to selectively introduce a filter between the NUnit form instance and the executing assembly. The non-static around binding in the classpect *IntroduceFilter* replaces original initialization method with a new method that in turn replaces the original text writer with a new text writer. We were also able to introduce a mediator between the result button and display method of the code coverage collector. The non-static after binding in the classpect *FormCovColMediator* was used to receive notification of result button click event and to call an appropriate method to display the coverage result. Eos thus offered simple and elegant designs without the need for unnecessary overheads in each case.

4. CASE STUDY 2: EOS

To facilitate the discussions of the redesign efforts of the Eos compiler, it will be worthwhile to give a brief overview of the organization of the Eos compiler's source code, components and architecture. The next section briefly describes the implementation of the Eos compiler. The description below reflects the implementation of Eos version 0.3.

4.1 Eos Compiler Implementation

The current implementation of Eos has seven components: *Abstract Syntax Tree (AST)*, *Code Compiler*, *Code Weaver*, *ConsoleUI*, *Parser*, *Runtime*, and *Utilities*. These components are shown in Figure 7. All components create or manipulate programs stored as an abstract syntax tree. The *AST* component handles abstract syntax tree building and provides a representation of the abstract syntax tree nodes in the compiler. As the name reflects the components *Parser*, *CodeWeaver*, *CodeGenerator*, and *CodeCompiler* parse, weave, generate and compile source code. The component *Utils* provides string manipulation, argument construction and similar functions. The component *ConsoleUI* is the main driver of the compiler and it invokes other components. Total size of the compiler is around *50K* line of code and it is organized as shown in Table 1) among components.

Figure 7: Components of Eos

As shown in table 1, the component *AST* has 224 classes in separate files. Out of 224 classes, 9 are used internally (*Eos.Utils* and *Eos.AST* namespace). The *CodeWeaver* uses approximately 120 classes to access and transform the representations of join points, pointcuts, bindings and classpects and to perform pointcut matching. The component *Parser* and the component *CodeGenerator* use all classes declared in the component *AST* except the 9 internal classes to construct and generate abstract syntax tree nodes. The components *Parser*, *Code Weaver* and *Code Generator* use several classes declared in the component *AST* to create, manipulate and generate abstract syntax tree nodes therefore they are highly coupled with it; however, they are mutually name–independent.

The architecture of the compiler is shown in Figure 8. It is essentially a blackboard architecture [15]. The component *AST* is the blackboard and other components in the

Table 1: Line of Code Distribution for Eos

Component	Namespaces	Number of Files	Line of Code
Eos.AST	Eos.AST.	1	82
	Eos.AST.Base.Attributes.	6	434
	Eos.AST.Base.Expressions.	46	2486
	Eos.AST.Base.Statements.	58	3017
	Eos.AST.Base.TypeMembers.	32	3399
	Eos.AST.Crosscuts.JoinPoints.	18	2551
	Eos.AST.Crosscuts.Matching.	6	218
	Eos.AST.Crosscuts.Patterns.	5	508
	Eos.AST.Crosscuts.Pointcut.	24	1172
	Eos.AST.Crosscuts.TypeMembers.	20	2134
	Eos.AST.Utils.	8	4507
	Total	**224**	**20508**
Eos.Utils	Eos.Utils.	17	4293
Eos.Parser	Eos.Parser.	11	16032
	Eos.Parser.Common.	8	413
	Eos.Parser.Crosscuts.	2	136
	Eos.Parser.CSharp.	3	2550
	Total	**24**	**19131**
Eos.CodeGenerator	Eos.CodeGenerator.	3	3499
Eos.CodeCompiler	Eos.CodeCompiler.	2	220
Eos.CodeWeaver	Eos.CodeWeaver.	12	1172
Eos.ConsoleUI	Eos.ConsoleUI.	3	275
Eos.Runtime	Eos.Runtime.	10	1494
	Eos.Runtime.Signature.	16	471
	Total	**26**	**1965**
	Grand Total	**311**	**51063**

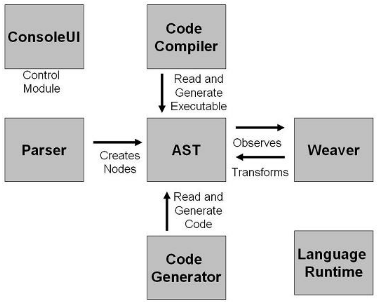

Figure 8: The Architecture of the Eos System

system read and write to it. The component *Parser* creates nodes on the blackboard. The component *CodeGenerator* and the component *CodeCompiler* are readers that read the component *AST* and generate another representation. The component *Weaver* observes changes to the *AST*.

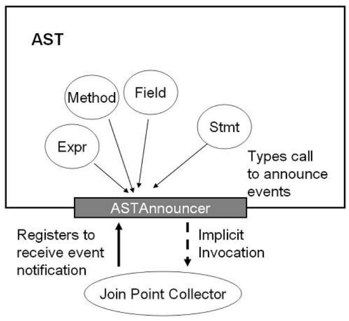

Figure 9: AST Event Announcer

To decouple observers and announcers, the design of *AST* uses implicit invocation mechanisms. As shown in Figure 9, *ASTAnnouncer*, a sub–component of *AST* declares events for the rest of the world. An event of interest related to *AST* nodes are announced through ASTAnnouncer. An event of a given type can be announced by multiple subjects.

4.2 Compilation Passes in Eos

To construct the abstract syntax tree the component

Parser takes the C# source code as input, calls it's subcomponent *Lexer* to extract tokens out of it, and on matching a node calls the constructor of that node type in the component *AST*. The component *Parser* requires only one pass on the complete syntax tree to complete this process.

The component *Code Weaver* performs the following ten different transformations on the abstract syntax tree listed in the order in which they are performed. Each transformation requires traversing the abstract syntax tree, called a compilation pass. In some cases, traversing only up to the type declaration level is necessary. A complete syntax tree traversal is called a *Full Pass* and a traversal until only the type declaration level is called *Short Pass*. Note that saving a full pass can significantly reduce compilation time on large systems.

1. Introduction Collection: IntroductionCollector pass collects all the inter–type declarations in the program. The inter–type declarations introduce crosscutting members into other types. As a result definitions of other types might change, therefore this pass has to be done in the very beginning. This is a short pass.

2. Declare Collection: DeclareCollector pass collects all the declare statements in the program. The declare statements are of three types: declare parents, declare errors, and declare warnings. The declare parents statement changes the inheritance hierarchy of a type and declare error and warning instruct the compiler to report warnings or errors on encountering specific patterns in the code. These patterns are specified using pointcut expressions. The declare parents statement can also potentially affect the definition of types in the program on which other passes depend. This is a short pass.

3. Introduction Planner: The introduction planner pass processes all the inter–type declaration requests collected in the introduction collection pass. This pass also accesses the declare parents statements collected by the declare collector pass to ensure correct behavior. This is a short pass.

4. Well-formedness Checker: This pass verifies that the resulting syntax tree satisfies the some very basic properties after introduction of inter–type declarations. This is a full pass.

5. Join Point Shadow Collector: This pass collects all join point shadows in the program. A join point shadow is a static location in the program, where a dynamic join point might occur. It is essential to perform this pass after the introduction planner pass to collect join points in inter–type declarations as well. This is a full pass. In particular, to collect method call join point, field set and get join point, property set and get join point, and exception throw join points visiting up to the expression nodes is necessary.

6. Binding Collector: This pass collects all method bindings in the program. It is essential to perform it after the introduction collector and planner pass as an inter–type declaration is allowed to introduce a binding into another type.

7. Declare Processor: This pass processes all declare statements collected previously. This is a full pass.

8. Action Collector: This pass collects all the directives for generalized concern processing. This is a short pass.

9. Crosscutting Infrastructure Builder: This pass prepares the necessary crosscutting infrastructure for weaving. For example, it creates the necessary instantiation and binding mechanisms in classpects. This is a short pass.

10. Join Point Weaving Pass: This pass performs the necessary instrumentation at join points to introduce invocation mechanisms to execute methods that are bound to that join point. This is a full pass.

Some of these transformations are performed in parallel. As a result the abstract syntax tree is visited five times by the weaver out of which only two passes are full passes. In total including the code generation and parsing passes, the current compiler implementation performs four full passes and three short passes in the following order:

1. *Full Pass*: Parsing

2. *Short Pass*: Introduction Collector and Declare Collector

3. *Short* Pass: Introduction Planner

4. *Full* Pass: Specification Checker, Join Point Collector, Binding Collector, Declare Processor, and Action Collector

5. *Short* Pass: Crosscutting Infrastructure Builder

6. *Full* Pass: Join Point Weaving

7. *Full* Pass: Code Generation

4.3 Modularizing Integration Concerns in Eos

Performing more of these passes in parallel would result in a further reduced compilation time. For example, the pass in which introductions and declare statements are collected could be merged with the source code parsing pass so that while parsing the parser also collects the introductions and declare statements. An inter-type declaration is also called an introduction. This merge, however, would either require that the introduction collection concern and declare statement collection concern be scattered inside the parsing concern or the component *Parser* allows introduction collector and declare collector to observe it. These components would register with the parser to get event notifications. The parser would invoke these concerns implicitly whenever it completes parsing an introduction and a declare statement. The first design alternative is clearly non-modular. The second design alternative decouples the parser from introduction and declare collector. However, it leaves introduction and declare statement collector name-coupled with parser. As a result, the component Code Weaver cannot be independently compiled and tested. The next subsection describes an alternative approach that is better than either of these design alternatives.

4.4 Modularizing Inter-Type and Declare Statement Collection

The alternative design is to use the mediator–based design structure to integrate the parser with the introduction collector and declare collector. Using this design alternative, the parser will declare and announce the event *CodeMemberIntroductionParsed* and *CodeMemberDeclareParsed*. Separate mediators will register with the *CodeMemberIntroductionParsed* and *CodeMemberDeclareParsed* events announced by the parser instance and on invocation will invoke the Collect method on the introduction collector and declare collector instance respectively. This design completely decouples the components. However, it leaves the event declaration and announcement concern scattered in the parser component. Using aspect-oriented programming to modularize this integration concern requires instantiation and instance-level advising emulation [26].

```
1 public class ParIntMediator {
2  public ParIntMediator(ASTParser parser,
3             IntroductionCollector IntC){
4   this.parser = parser;
5   this.IntC = IntC;
6   addObject(parser);
7   }
8  ASTParser parser; IntroductionCollector IntC;
9  after execution(public virtual CodeMemberIntroduction
10       ASTParser.CreateIntroduction(..) && return(o):
11                   IntParsed(CodeMemberIntroduction o);
12 public void IntParsed(CodeMemberIntroduction o){
13  IntC.Collect(o);
14  }
15 }
```

Figure 10: The Parser Introduction Collector Mediator

```
1 public class ParDeclMediator{
2  public ParDeclMediator(ASTParser parser,
3                   DeclareCollector DeclC){
4    this.parser = parser;
5    this.DeclC = DeclC;
6    addObject(parser);
7    }
8  ASTParser parser; DeclareCollector DeclC;
9  after execution(public virtual CodeMemberDeclare
10       ASTParser.CreateDeclare* (..) && return(o):
11          AfterDeclareParsed(CodeMemberDeclare o);
12 public void AfterDeclareParsed(CodeMemberDeclare o){
13   DeclC.Collect(o);
14   }
15 }
```

Figure 11: The Parser Declare Collector Mediator

The unified model, however, achieves an improved modularization of these concerns as classpects. These concerns are modularized as *ParIntMediator* and *ParDeclMediator* classpects as shown in in Figure 10 and 11. These two mediators when introduced in the system eliminate the second pass of the compiler by merging inter type declaration and declare statement collection in between the parsing pass. This is done without introducing any name dependencies between the component *Parser* and the component *CodeWeaver*.

The *ParIntMediator* classpect (lines 1–15 in Figure 10) declares a constructor (Lines 2-7) to integrate a *ASTParser* and an *IntroductionCollector* instance. The constructor stores references to both instances and binds itself to the parser instance using the implicit method *addObject*. Recall that the implicit method *addObject* registers the method handlers of the binding to the parser instance. The binding (Lines 9–11) in the mediator selects the execution of the *ASTParser.CreateIntroduction* method and binds method *IntParsed* (Lines 12–14) to execute after it. It also binds the parameter *o* of the IntParsed method with the return value of the *ASTParser.CreateIntroduction* method. The *IntParsed* method calls the *Collect* method (Line 13) on the stored *IntroductionCollector* instance passing it the introduction to collect.

Similarly, the classpect *ParDeclMediator* (Lines 1-15 in Figure 11) declares a constructor (Lines 2-7) to integrate an *ASTParser* and a *DeclareCollector* instance. The constructor stores references to both instances and binds itself to the parser instance using the implicit method *addObject*. The binding (Lines 9-11) in the mediator selects the execution of all methods in the *ASTParser* whose names begin with the pattern *CreateDeclare**. The pattern *** is used to collect creation of declare statement concern that is spread over three different methods that create declare parents, declare error, and declare warning constructs. The binding binds the method *AfterDeclareParsed* (Lines 11-13) to execute after the execution of these methods. It also binds the parameter *o* of the method *AfterDeclareParsed* with the return value of these methods. The method *AfterDeclareParsed* calls the method *Collect* (Line 12) on the stored *DeclareCollector* instance passing it the declare statement to collect.

This redesign effort thus led to the saving a short pass resulting in shortened compile time. For the rest of the discussion, the numbering of passes will remain the same to facilitate ease of reference, so even though the introduction collection and declare collection passes are merged with the first pass and there is no second compilation pass now, we will continue calling introduction planner and successive passes third pass and so on.

4.5 Reordering Third and Fourth Full Pass

Encouraged by elimination of the second pass by merging it with the first parsing pass in a modular way, we explored the possibilities of merging the fourth pass with the first pass. Note that the third and fourth pass cannot be reordered without accounting for their interaction due to the data dependence between these passes; however, benefits of saving a full pass of the compiler were enticing.

To understand and identify the data dependence between the third pass and the fourth pass, understanding the nature of an introduction is necessary. An introduction identifies type in the programs by a pattern, and introduces a set of type members such as methods, fields, bindings, actions, etc. into them at compile-time. An implicit effect of introducing new type members into a type is to also introduce new join point shadows in the type, that in turn need to be collected by the join point shadow collector. The join point collector in the fourth pass thus depends upon the output of the introduction planner in the third pass.

The optimization strategy used was to first collect the existing join point shadows and then in the introduction planner pass additively insert the join point shadows as new type members are added into type declarations. Two new classpects were introduced to implement this strategy. The first classpect, ParJpCMediator acts as mediator between the Parser and the Join Point Shadow collector module. Like the mediator between the Parser and the Introduction Collector, this classpect keeps references to the Parser and the Join Point Shadow Collector instances, registers a method with the join points in the Parser class to be invoked when Parser finishes parsing join point shadows and calls the Join Point Shadow Collector instance to collect shadows.

The second classpect, JpCIpMediator acts as a mediator between the introduction planner and join point shadow collector. This classpect invokes the join point collector to collect join point shadows whenever the introduction planner introduces a new type member into a type declaration. We introduced other classpects as mediating components between binding collector and parser, action collector and parser, specification checker and parser and action collector and parser to complete the merge of the fourth full pass with the first parsing pass.

The assessment discussed in this section showed that the unified model, the Eos language, and the Eos compiler improved the modularization of challenge concerns in the Eos compiler. The challenge concerns that we discussed are examples of component integration concern [31]. We were able to eliminate one short pass and one full pass resulting in a significant saving in compilation time. Eos thus offered simple and elegant designs without incurring unnecessary overheads of instance emulation and instance-level advising emulation in this case as well.

5. LIMITATIONS

Using Eos infrastructure to handle significant systems demonstrated two key limitations: the need for source code to perform source level weaving and the lack of tool support. In the rest of this section, we will discuss each of these issues in detail and identify how it affected our case study.

5.1 Source Level Weaving: A Constraint

Currently Eos performs source level aspect weaving. It requires source code of a component or class to advise it. This requirement severely affects separate compilation. The production versions of Eos, essentially the bootstrapped version built using plain C#, are organized into a directory hierarchy corresponding to different components in the system. The component structure is shown in Figure 7. These components are compiled, tested, and debugged separately.

The new version of Eos built for this evaluation, retained the directory structure of the bootstrapped versions, however, keeping the separation at source code level between component boundaries was difficult when they were integrated using classpects. To implement a connector, access to the source code of all involved components was required.

To address this concern, weaving approaches at the .NET's intermediate language (IL) level will be explored in the future. The intermediate language called MSIL is interpreted by the Common Language Runtime (CLR) [12], a language and platform–neutral infrastructure. The CLI supports multiple programming languages and is controlled by a vendor neutral Common Language Infrastructure (CLI) [5] specification. Enabling weaving support at the CLR level thus also opens up opportunities for cross–language aspect–oriented programming.

5.2 Tool Support

Building large and complex systems without significant tool support is very difficult. Currently the Eos project does not provide any tool support, such as integrated development environments for developing and debugging programs in Eos. The lack of debugging support is a significant problem. Aspect–oriented programming features are not widely understood i.e. chances of error are even greater. Lack of support for detecting precisely where the error is may make it very difficult to write programs. Writing debugging support for Eos is difficult due to the proprietary nature of the PDB format of Microsoft.

There is however, hope stemming from the community involvement. A student from University of Szeged in Hungary, Somkutas Pter, recently developed an add-in for Eos [25]. This add-in allows Visual Studio to compile Eos programs using the compiler developed at the University of Virginia. The add-in is still in preliminary stage, however, it shows signs of the impact of the Eos project internationally and the hopes of supporting tool development for the language model.

6. DISCUSSION

The case studies described in this work show that the unified language model described in our previous work is beneficial in real world systems. An improved separation of concerns was achieved in all challenge problems considered in the context of Eos compiler redesign and ConcernCov tool design. The evaluation provided in this work, however, does not provide direct evidence that the unified model is beneficial beyond the challenge problems considered in the case studies. The concerns that we have applied the unified model are largely co-ordination concerns, in that they co-ordinate the behavior of two or more components. Coordination concerns are extremely important class of concerns; however, they are not the only crosscutting concerns aspect-oriented programming addresses. Aspects are used for modularizations of other concerns like transaction management, thread pooling, security policy enforcement, etc. From the evaluation presented in this work, it is not evident whether the unified model is beneficial in these cases.

We speculate, but have not yet systematically tested, that the benefits of the unified model will be perceived in the modularization of other concerns. For example, consider a system where a variety of security policies exist and a policy is applied to an object instance depending upon how it interacts with the outside world, e.g. local security policy for local object instances, domain security policy for object instances that only interact with the other entities on the local area network (LAN) and global security policy for the rest of the object instances. In such system, implementing security policy as an aspect with no support for instance-level advising will at the minimum complicate the concern code with the code required to emulate aspect instances and instance-level advising. The unified model will not incur this design-time overhead.

7. RELATED WORK

The subject of this evaluation, the unified aspect language model [28], is related to AspectJ [18], AspectWerkz [2], and Caesar [22]. In at least one early version of AspectJ, there were no separate aspect construct. In this version, the class was extended to support advice. To the best of our knowledge, the synthesis of OO and AO techniques achieved by our unified model was not present there. Advice bodies and methods were still separate constructs; and it is unclear to what extent advising as a general alternative to method invocation and implicit invocation was supported. In addition, flexible aspect instantiation and instance-level weaving were not supported. Rajan and Sullivan showed that first-class aspect instances and instance-level advising improved the modularization of integration concerns [26, 30]. Current work reinforced our earlier findings.

Another closely related design is that of AspectWerkz [2]. The aim of this project was to provide the expressiveness of AspectJ [18] without sacrificing pure Java and the supporting tool infrastructure. The solution is to use normal Java classes to represent both classes and AspectJ-like aspects, with advice represented in normal methods, and to separate all join-point-advice bindings either into annotations in the form of comments, or into separate XML binding files. AspectWerkz provides a proven solution to the problem of AspectJ-like programming in pure Java, but it does not achieve the unification that we have pursued.

First, and crucially, the system does not support the concept of aspects as objects under program control; rather it is really an implementation of the AspectJ model. Instead, the use of Java classes as aspects is highly constrained so that the runtime system can maintain control. A *class* representing an aspect must have either no constructor or one with one of two predefined signatures, and a method representing an advice body has one argument of type JoinPoint. AspectWerkz uses this interface to manage aspect creation and advice invocation. AspectWerkz also lacks a single-language design, in that it uses both Java and XML binding files. Third, AspectWerkz lacks static type checking of advice parameters. Rather, reflective information is marshaled from the JoinPoint arguments to advice methods.

The design of Caesar [22] is also closely related to our approach. The aim of Caesar was to decouple aspect implementation and the aspect binding with a new feature called an aspect collaboration interface (ACI). By separating these concepts from aspect abstraction, Caesar enables reuse and componentization of aspects. This approach is similar to ours and to AspectWerkz in that it uses plain Java to represent both classes and aspects; however, it represents advice using AspectJ like syntax. Methods and advices are still separate constructs, and advice constructs couples crosscut specifications with advice bodies. Consequently, as in AspectJ, advice bodies are still not addressable as individual entities. They can be advised as a group using an advice-execution pointcut. In Caesar, as in Eos, advice can be bound statically or dynamically; however, aspects in Caesar cannot directly advise individual objects on a selective basis.

Aspect languages such as HyperJ [32, 24] have one unit of modularity, classes, with a separate notation for expressing bindings. However, they do not support program control over aspects as first-class objects, and to date the join point models that they have implemented have been limited mainly to methods [14].

Several others have evaluated aspect-oriented programming techniques on different benchmarks. Early assessments were conducted by Mendhekar et al [21], Kersten and Murphy [17], Walker et al. [33], etc. Mendhekar et al [21] used RG, an environment for creating image processing systems to evaluate aspect-oriented programming. Kersten and Murphy [17] used Atlas, a web-based learning environment to evaluate aspect-oriented programming. Walker et al. [33] also conducted an initial assessment of aspect-oriented programming. Most recently, Hannemann and Kiczales [13] compared the object-oriented and aspect-oriented implementations of the Gang of Four design patterns [7] using qualitative metrics. Garcia et al. [8] used a set of quantitative metrics to compare the object-oriented and aspect-oriented implementations.

8. CONCLUSION

Earlier, we showed that the notion of aspect and class as embodied in the AspectJ model can be unified in a new module construct that we called *classpect* and that this new model is at once simpler and more compositional. The core contribution of this work is to provide a challenging assessment of the unified model, the Eos language, and the Eos compiler in the context of real world systems. Our benchmarks were two significant systems: ConcernCov, a tool for concern-based code coverage analysis of test suites (20K LOC), and the Eos compiler, a near-industrial strength *classpect-oriented* extension to the C# language (50K LOC). In both systems the elegance and the simplicity of the resulting designs provides evidence for the potential design structuring benefits of the Eos model, the usability of the Eos language, and the practical utility of our language implementation in the Eos compiler. The evaluation exhibits the immediate practical value of our conceptual work reinforcing our previous demonstrations that shows that at least at the programming language level, the separation of aspects and classes is harmful. What remains to be seen is whether aspects are valuable as a separate conceptual category or should we instead think in terms of a single conceptual building block for program design?

9. ACKNOWLEDGMENTS

This work was supported in part by NSF grants ITR–0086003 and FCA–0429947. The comments from William G. Griswold and Gary T. Leavens were very helpful in deciding the scope of these case studies and in presentation.

10. REFERENCES

[1] AspectJ programming guide. http://www.eclipse.org/aspectj/.

[2] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM Press.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[4] A. Bryant and R. Feldt. AspectR - simple aspect-oriented programming in Ruby, Jan 2002.

[5] ECMA. *Standard-335: Common Language Infrastructure (CLI) Specification*, December 2001.

[6] Eos web site. http://www.cs.virginia.edu/ eos.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena, E. Figueiredo, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM Press.

[9] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[10] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[11] B. Hamilton. *NUnit pocket reference*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2004.

[12] J. Hamilton. Language integration in the common language runtime. *SIGPLAN Not.*, 38(2):19–28, 2003.

[13] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[14] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.

[15] B. Hayes-Roth. A blackboard architecture for control. *Artif. Intell.*, 26(3):251–321, 1985.

[16] R. Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[17] M. Kersten and G. C. Murphy. Atlas: a case study in building a web-based learning environment using aspect-oriented programming. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 340–352, New York, NY, USA, 1999. ACM Press.

[18] 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.

[19] J. Lamping. The role of base in aspect-oriented programming. In C. V. Lopes, A. Black, L. Kendall, and L. Bergmans, editors, *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.

[20] H. Masuhara and G. Kiczales. Modular crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743, pages 2–28, Berlin, July 2003.

[21] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC, Feb. 1997.

[22] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

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

[24] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. IBM Research Report 21452, IBM, Apr. 1999.

[25] S. Pter. ECT (EOS Compiler Tool): an add-in for Visual Studio, 2005.

[26] 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.

[27] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 181–191, New York, NY, USA, 2005. ACM Press.

[28] 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.

[29] O. Spinczyk, A. Gal, and W. Schroeder-Preikschat. AspectC++: an aspect-oriented extension to the c++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[30] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages: integration as a crosscutting concern for aspectj. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 19–26, New York, NY, USA, 2002. ACM Press.

[31] 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.

[32] 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.

[33] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 120–130, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.