Mining Software Repositories for Evaluating Software Engineering Properties of Language Designs

Hridesh Rajan
Dept. of Computer Science,
Iowa State University
hridesh@cs.iastate.edu

ABSTRACT

Improved separation of concern is important for dealing with increasing complexity of today's software systems. A number of language designs have been proposed in the last decade with the common goal to improve the separation of concerns by providing better modularization mechanisms e.g. mix-ins, units, roles, layers, hyperspaces, events, aspects, etc. To understand the benefits of a new modularization mechanism, it is important to apply it to real world large scale software systems, where there are real needs for separation of concerns. However, large scale software projects are generally managed very cautiously and adoption of a new technique in these projects is generally harder to achieve. Typically such adoption is driven by demonstrated success of the technique in other large scale projects, a catch-22 situation. In this position paper, I discuss a software repository mining-based technique to achieve the effect of adoption in a large scale software project in a controlled setting. Rich change history available in the version control systems for open source software projects, and advances in software repository mining enable this technique for empirical evaluation of a modularization mechanism.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques — Modules and interfaces; D.2.8 [**Software Engineering**]: Metrics — Complexity Measures; D.3.3 [**Programming Languages**]: Language Constructs and Features — Control structures; Procedures, functions, and subroutines

General Terms

Design, Measurement, Human Factors, Languages

Keywords

modularity, empirical evaluation, software repositories, design for change, information hiding, programming language design, separation of concerns metrics

1. INTRODUCTION

Today, finding an appropriate *separation of concerns* [16] is understood to be perhaps the fundamental challenge in software design. It promotes studying different interests or concerns of a complex problem separately, with none or very little knowledge of other concerns. *Separation of concerns is the key to maintaining overall intellectual control in the face of complex problem and design solutions.* Over the years, the quest for better separation of concerns has led to different technologies, among which are well-established modularization techniques [43, 44] such as structured programming [14, 15, 33, 59], abstract data types [37], and object-orientation (OO) [11, 26, 34, 39, 58].

A whole new set of such modularization techniques as mix-ins [4], units [22], implicit-invocation [25, 54, 55], hyperslices [28, 42], composition filters [1], adaptive methods [36], roles [35], aspects [31, 48, 49, 45], open classes [10], etc, have emerged in the last decade or so with a common goal to enable improved separation of concerns. Although the Jury is still out on some of these techniques, the overwhelming academic and industrial interest makes it abundantly clear that there is a pressing need for improved separation of concerns techniques. This is primarily because implementation of some concerns are often hard to factor out into separate modules using classical decomposition techniques [56]. For example, the code for a thread policy is spread across the system. These emerging modularization techniques provide engineers with new possibilities for keeping such concerns separate in the source code.

Typical evaluation of a modularization mechanism examines it in the context of canonical examples. For example, figure editor example inspired from JHotDraw [23], a system originally developed as a design exercise by Erich Gamma and Thomas Eggenschwiler, is prevalent in the aspect-oriented programming literature. Evaluation using canonical examples has many advantages. Canonical examples act as a smoke test of the technique, "if the technique does not work in the context of canonical examples, it is unlikely to work in real projects." They also allows readers and researchers to focus on the problem at hand without getting distracted by the essential complexities of the problem domain [8].

Full promise of these techniques is, however, hard to evaluate with just canonical examples. If a technique doesn't work in the context of canonical examples, often we can say for sure that it is not going to work for real world projects, however, if it does work results obtained may not be directly applicable to real world projects. Scalability related issues are almost never examined in a study that uses canonical examples. A new modularization technique can be applied to a real world project to address some of these threats to the validity of its evaluation. A real world study is, however, extremely time consuming. Often it also requires insider

access to subject projects to make the case for adoption. Last but not the least, the case for adoption often depends on the demonstrated success of the modularization technique in the context of other projects, a chicken and an egg situation.

This position paper describes an empirical evaluation technique that we believe could be useful to overcome some of these hurdles. The key idea is to capitalize on the rich version control histories available for the open source software projects to realistically simulate a series of real world changes in software projects. First, an initial and a final version for the open source project are selected. Second, using existing techniques for software repository mining changes between initial and final versions are organized as a sequence of refactorings [13] such that by replaying the refactorings in the specified order on the initial version yields the final version. Third, this sequence of refactorings is transformed into all candidate modularization techniques such that by replaying the modified refactorings in the specified order on the initial version yields the final version, (possibly) improved using a candidate modularization technique. Finally, analyzing each refactored version produced in this manner is compared w.r.t. desired software engineering metrics to compare and contrast modularization techniques. The scope of our evaluation technique is limited to the following:

- *techniques*: language constructs for improved separation of concerns,
- *claims*: in terms of information-hiding modularity [43], design for change, etc, and
- *project settings*: steady-state projects with source control histories.

The main objective of our technique are:

- *overcome the threats to the validity*: to reduce the threat to the validity of empirical evaluations of language designs, our technique should reduce/eliminate biases from the experimental settings, and
- *automation/semi-automation*: to reduce the cost of empirical evaluation and to encourage rigorous, extensive evaluation of modularization techniques, it should be possible to automate much of this technique at a low cost.

In the rest of this position paper, we explain our technique in detail. To make the ideas concrete, we will discuss them in the context of the Ptolemy language recently proposed by Rajan and Leavens [47]. Ptolemy combines ideas from implicit invocation and aspect-oriented languages and has several advantages compared to both. Next section briefly presents this language design. Section 3 presents the proposed evaluation technique and Section 4 concludes.

2. PTOLEMY: A BRIEF INTRODUCTION

Ptolemy is an extension of object-oriented languages with support for quantified, event types [47]. Ptolemy's design is inspired by II languages such as Rapide [38] and AO languages such as AspectJ [30]. It also incorporates some ideas from Eos [50, 48] and Caesar [40]. The key ideas in the language are that it lets programmers declare named *event types* that contain information about the names and types of event arguments (exposed context). An event type identifies an expression as an event in a declarative manner. This event type can then be used to quantify over all such events. Event types reduce the coupling between the observers of the events and the set of events by eliminating the name dependence between the two.

An example Ptolemy program is shown in Figure 1. This code is part of a larger editor that works on drawings comprising points, lines, and other such figure elements [30, 32]. The program is adapted to be more Java-like, whereas the language presented in our previous work on Ptolemy was an expression language [47].

```
1 FElement evtype FEChange{ FElement changedFE; }
2 FElement evtype MoveUpEvent {
3  FElement targetFE; int y; int delta;
4 }
5 interface FElement{}
6 public class Point implements FElement{
7  int x, y;
8  public void setX(int newX){
9   FElement changedFE = this;
10  event FEChange{ x = newX;}
11 }
12 public void moveUp(int delta){
13  FElement movedFE = this;
14  event MoveUpEvent{ y = y + delta; }
15 }
16 public void makeEqual(Point other){
17  FElement changedFE = other;
18  event FEChange{
19   other.x = this.x; other.y = this.y;
20  }
21 }
22 }
23 public class Update{
24 FElement last;
25 public Update Update(){ register(this); }
26 public void update(FEChange next,
27                    FElement changedFE){
28  proceed(next); this.last = changedFE;
29  Display.update();
30 }
31 public void check(MoveUpEvent next,
32  FElement targetFE, int y, int delta){
33  if (delta < 100)){ proceed(next); }
34 }
35 when FEChange do update;
36 when MoveUpEvent do check;
37 }
```

Figure 1: Drawing Editor in Ptolemy

In companion papers [46, 47], we discussed the limitations of II and AO languages. To summarize, compared with AO languages, II languages have three limitations [47]. First, while subject modules are decoupled from observer modules, observer modules remain coupled with subjects. Second, their is no construct equivalent to AO "around advice" that allows to replace the code for an event. Instead, unnecessarily complex emulation code to simulate closures in languages such as Java and C# is required in II languages. Third, *quantification* can be tedious in II languages. The code that describes how each event is handled can grow in proportion to the number of objects from which implicit invocations are to be received.

Compared with II languages, AO languages have four limitations [47]. These limitations are not conceptual, rather, they stem from the fact that implementations of most current AO event models use PCDs based on pattern matching (on names [30], lexical structures [51, 20], program traces [17], etc). First problem is commonly known as the "fragile pointcut problem", which is caused by the use of pattern matching as a quantification mechanism [52, 57]. Pattern matching based PCDs are coupled to the code that

implements the implicit event that they describe. Thus, seemingly innocuous changes break aspects [29]. Recent research results such as Aspect Aware Interfaces (AAIs) [32], Crosscut Programming Interfaces (XPIs) [53, 27], Model-based Pointcuts [29], Open Modules (OM) [2], etc, have recognized and proposed to address the fragile pointcut problem, but none address it completely [47].

Second problem is that current AO event models do not implicitly announce some kinds of events [53, pp. 170]. Therefore, they also do not provide PCDs that select such events. Alternative AO approaches such as LogicAJ provide a finer-grained event model [51], however, PCDs in such techniques become strongly coupled with the structure of the base code and therefore become more fragile [47].

Third and fourth problems have to do with the interface for accessing contextual (or reflective) information about an event. In some AO approaches, this interface is fixed by the language designer and does not satisfy all usage scenarios [53]. In other AO languages (e.g. LogicAJ [51]), virtually unlimited reflective access to the context surrounding the lexical structure using meta-variables is possible but requires that the events form a regular structure [47]. Furthermore, contextual information that fulfills a common need (or role) in the handlers is not available uniformly to PCDs (and handlers) [47].

In Ptolemy, `evtype` declaration allow programers to declare named event types. An event type (`evtype`) declaration p has a return type, a name, and zero or more context variable declarations. These context declarations specify the types and names of reflective information communicated between announcements of events of type p and handler methods. These declarations are independent from the modules that announce or handle these events. The event types thus provide an interface that completely decouples subjects and observers. Events are explicitly announced using `event` expressions. These expressions enclose a body expression, which can be replaced by a handler. This functionality is similar in expressiveness to **`around`** advice in AO languages.

Finally, the names of `evtype` declarations can be utilized for quantification, which simplifies binding and avoids coupling observers with subjects.

3. EMPIRICAL EVALUATION APPROACH

Conducting an empirical evaluation of the software engineering properties of a new language design is a challenge. For example, claims such as "the quantification based on quantified, event types are less fragile compared to traditional syntactic quantification mechanisms" or that "quantified event-types improve the robustness of the handler code against base code changes, and makes it easier for the handlers to uniformly access reflective information about the event," may not be validated without large-scale use of the language design over an elongated period of time. Conclusions drawn from small examples, although helpful, may not correctly reflect the anticipated software engineering benefits of the language design.

A reasonable evaluation necessitates enough experience with the design to really say for sure it is right. In order to conduct such an evaluation, the precondition is the adoption of the language design in real world large-scale projects. Ironically, such adoption is often driven by demonstrated success of the language design in other projects. Taking these considerations into account, our proposed empirical evaluation technique is as follows:

3.1 Select Candidate Software Projects

The primary criteria for selecting a project as a candidate for our empirical evaluation technique is that they should be open source i.e. the source code is available for analysis, presence of existing version history that can be analyzed, large-size i.e. improved separation of concerns has real and perceived value in the context of this project, an active community contributing frequent releases and bug-fixes, which in turn translates to a rich change history in CVS.

For Ptolemy an additional requirement must be imposed. Our current Ptolemy infrastructure is based on Java, therefore, the project should be a Java project. A bonus would be an open and accepting community that can be persuaded to adopt based on demonstrated results.

Current candidate projects for Ptolemy include Eclipse [65], NetBeans [66], Azureus [64] and Ant [63]. We already have some experience with Eclipse [65], Azureus [64] and Ant [63] projects and their current build systems in the context of another research project [18, 19].

3.2 Select an initial version for candidate projects

An older version of the project is extracted from its repository and used as a baseline for the empirical evaluation. For example, a 2001 version of Eclipse with the sticky tag `v20011218` will be selected as the baseline for Eclipse. A challenge in this task is that often earlier versions of a software system use a different set of libraries, runtimes, etc. For example, all of our candidate projects made a transition from using Java 1.4 to Java 1.5 and then to Java 1.6 in the last few years. Recreating the build environment for such projects will be the key, however, once such a build environment is created, it could be reused for a number of candidate projects.

3.3 Use Concern Mining Techniques to Semi-automatically Identify Fragmented and Scattered Concerns for Modularization

After identifying an old version of a candidate project, we use automatic concern mining techniques on this version to extract fragmented and scattered concerns for modularization. Although automatic concern mining is still an emerging area, a number of techniques are available [5, 6, 61, 60]. One such technique by Breu, Zimmermann, and Lindig [7, 5, 6] is also applied to identify fragmented and scattered concerns in Eclipse [65]. Much of the reported results is directly applicable.

3.4 Use Ptolemy and Alternative II and AO Techniques to Modularize Identified Concerns

We then use the hints from automatic concern mining techniques to modularize the fragmented and scattered concerns in candidate projects. Three different starting versions are created in this step, one that uses implicit invocation techniques to modularize, a second that uses AO techniques to modularize, and a third that uses quantified, event types of Ptolemy.

3.5 Replay Changes on All Versions Using Version History

Once we have three versions (Ptolemy, AO and II) of the candidate project's old release, we will extract real changes from the project's CVS repository.

At least three frameworks are available today for this task APFL [62, 12], which is an open source framework for the ECLIPSE programming environment that facilitates the analysis of CVS archives, Kenyon [3], a common extraction, preprocessing, and storage platform for software configuration management and analysis, and Molhado [41], a configuration management and

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components (CDC)	This metric is defined as the total number of classes (including those that announce events, and those that provide handlers that register to these events) the contribute to the implementation of a concern and those that access these classes.
	Concern Diffusion over Operations (CDO)	This metric is defined as the total number of methods and handler methods that mainly contribute to the implementation of a concern and the number of other methods and handlers that access them.
	Concern Diffusions over LOC (CDLOC)	This metric is defined as the total number of points for each concern in the code where there is a transition from one concern to another.
Coupling	Coupling Between Components (CBC)	Total number of classes with which a given class is coupled.
	Depth Inheritance Tree (DIT)	This metrics essentially represents the depth of the inheritance hierarchy in an application.
Cohesion	Lack of Cohesion in Operations (LCOO)	Measures the lack of cohesion of a class in terms of the amount of method and handler methods that do not access the same instance variable.
Size	Lines of Code (LOC)	Total line of code in the application excluding comments.
	Number of Attributes(NOA)	Total number of attributes of each class.
	Weighted Operations per Component (WOC)	Total number of methods of each class and the number of its parameters.

Figure 2: The metrics suite to be used in this project and their definitions[9, 24, 21].

version control infrastructure and methodology.

In the context of Ptolemy we use Molhado as it supports a higher-level of abstraction for analyzing changes between versions. For example, a variant of Molhado, MolhadoRef [13] has been used to replay refactorings of object-oriented programs to detect conflicts.

Extracted real changes in the candidate projects will be replayed to mimic software evolution. The key challenge in this activity is that Ptolemy's, II, and AO versions of candidate projects will differ from the baseline old version that we extracted from the repository in that some fragmented, scattered and tangled concerns are now modularized in this version. This is likely to create conflicts when we replay changes.

Our current insight into resolving this issue is to divide modularization of fragmented, scattered and tangled concerns as a set of refactorings. We also divide real changes exhibited in the project as refactorings.

We then use the MolhadoRef [13] tool to reconcile these refactorings with each other. As of today, we perceive this to be the biggest threat to the feasibility of our empirical evaluation.

3.6 Software Evolution Analysis and Metrics for Evaluation

The final step is then to analyze the effect of replaying the changes on all three versions of the candidate software project. This analysis will primarily use the set of metrics suite developed by Garcia *et al.* [24]. Garcia *et al.* [24] refined the object-oriented metrics proposed by Chidambar and Kemerer [9] and described by Fenton and Pfleeger [21] for advanced separation of concerns techniques.

We have adapted these metrics for evaluating II and Ptolemy's solutions. For AO solution, the original metrics defined by Garcia *et al.* [24] will be used. The adapted metrics and their definitions are shown in Figure 2.

The adapted definitions are closer to Chidambar and Kemerer's [9] original definition due to the unification of aspects and classes [50] in Ptolemy's language model.

4. CONCLUSION

In this position paper, we discussed a technique for empirical evaluation of software engineering properties of new language designs that claim to provide software engineers with new capabilities to modularize their concerns. The key idea behind the empirical evaluation is to use the real changes in a open source software project's life time to model software evolution. This promises to reduce the biases in the evaluation. With the help of advances in software repository mining techniques much of the evaluation process could be automated, which is an added advantage. We presented our technique in the context of Ptolemy's evaluation, however, it would be interesting to see whether it generalizes beyond II and AO-like language designs.

There are several questions that remain to be answered and the author would like to direct the attention of the workshop participants to most important of these.

1. Proving equivalence of a refactoring across language designs would be important. This risk could be mitigated to a certain extent if the base language for these advance modularization mechanisms is the same. An additional technique could be to give semantics to the new language constructs by translation to existing construct. Then the issue of the equivalence of refactoring would reduce to the issue of equivalence of two snippets in one language.

2. Is the quality of subject projects important? Yes and no. Although following better software practices in a software project often increases the benefits obtained from a new modularization, the metrics measured here are relative. Furthermore, it would be sensible to see the performance of a new modularization technique in the context of projects that may not have been designed to make such modularization easy to begin with.

3. Last but not the least, the need for a community-driven framework is apparent here. The efforts required to materialize this kind of validation framework may be beyond the scope of a single research program and may require cooperation among several research projects with complementary expertise.

Acknowledgements

This work is supported in part by the National Science Foundation under grant CNS-06-27354.

5. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[2] J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. 2005 European Conf. Object-Oriented Programming (ECOOP 05)*, pages 144Ű–168, July 2005.

[3] J. Bevan, S. Kim, and L. Zou. Kenyon: A common software stratigraphy system. |http://dforge.cse.ucsc.edu/projects/kenyon|.

[4] 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. ACM Press, 1990.

[5] S. Breu and T. Zimmermann. Mining aspects from version history. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, 2006.

[6] S. Breu, T. Zimmermann, and C. Lindig. Aspect mining for large systems. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 714–715, 2006.

[7] S. Breu, T. Zimmermann, and C. Lindig. Mining eclipse for cross-cutting concerns. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 94–97. ACM, 2006.

[8] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison Wesley, Reading, Mass., second edition, 1995.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[10] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145. ACM Press, 2000.

[11] O.-J. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[12] V. Dallmeier, P. Weigerber, and T. Zimmermann. APFL: A preprocessing framework for eclipse. |http://www.st.cs.uni-sb.de/softevo/apfel/|.

[13] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware software merging and configuration management. 2007.

[14] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[15] E. W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[16] E. W. Dijkstra. On the role of scientific thought. *EWD 477*, August 1974.

[17] R. Douence, P. Fradet, and M. Sudholt. Trace-based aspects. *Aspect-oriented Software Development*, pages 141–150.

[18] R. Dyer, H. Narayanappa, and H. Rajan. Nu: preserving design modularity in object code. *SIGSOFT Softw. Eng. Notes*, 31(6):1–2, 2006.

[19] R. Dyer and H. Rajan. Modular program transformations for aspect-oriented constructs. Technical Report 434, Iowa State University, Department of Computer Science, July 2006.

[20] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *APLAS 04*, pages 366–381.

[21] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[22] M. Flatt and M. Felleisen. Units: cool modules for hot languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 236–248. ACM Press, 1998.

[23] E. Gamma and T. Eggenschwiler. Jhotdraw: a java gui framework for technical and structured graphics. |http://sourceforge.net/projects/jhotdraw|.

[24] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, 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. ACM, 2005.

[25] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 31–44, London, UK, 1991. Springer-Verlag.

[26] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[27] W. G. Griswold, K. J. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software, Special Issue on Aspect-Oriented Programming*, Jan/Feb 2006.

[28] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 411–28. IEEE Comput. Soc, Los Alamitos, CA, October 1993.

[29] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP '06*, pages 501 – 525.

[30] 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.

[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[32] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[33] D. E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974.

[34] B. B. Kristensen, O. L. Madsen, B. Muller-Pedersen, and K. Nygaard. Abstraction mechanisms in the beta programming language. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 285–298. ACM Press, 1983.

[35] B. B. Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2(3):143–160, 1996.

[36] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Co., Boston, MA, USA, 1995.

[37] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.

[38] D. C. Luckham, J. J. Kennedy, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–54, April 1995.

[39] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[40] 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. ACM Press, 2003.

[41] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 215–224. ACM, 2005.

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

[43] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, December 1972.

[44] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–66, March 1985.

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

[46] H. Rajan and G. T. Leavens. Quantified, typed events for improved separation of concerns. Technical Report 07-14, Iowa State University, Department of Computer Science, July 2007. In submission.

[47] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08: 22nd European Conference on Object-Oriented Programming*, July 2008.

[48] 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.

[49] H. Rajan and K. Sullivan. Need for instance level aspect language with rich pointcut language. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, mar 2003.

[50] 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.

[51] T. Rho, G. Kniesl, and M. Appeltauer. Fine-grained generic aspects. In *FOAL'06*. 2006.

[52] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.

[53] K. J. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *The Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 166–175, Sept 2005.

[54] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. *SIGSOFT Software Engineering Notes*, 15(6):22–33, December 1990.

[55] 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.

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

[57] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the aosd-evolution paradox. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, March 2003.

[58] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.

[59] N. Wirth. *Systematic Programming: An Introduction*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973.

[60] I. Yuen and M. P. Robillard. Bridging the gap between aspect mining and refactoring. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, page 1, 2007.

[61] C. Zhang and H.-A. Jacobsen. Efficiently mining crosscutting concerns through random walks. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 226–238, 2007.

[62] T. Zimmermann. Fine-grained processing of cvs archives with apfel. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 16–20, 2006.

[63] Ant website. | http://ant.apache.org/|.

[64] Azureus website. | http://azureus.sourceforge.net/|.

[65] Eclipse website. | http://www.eclipse.org/|.

[66] NetBeans website. | http://www.netbeans.org/|.