

# Localizing Views for Separation of Concerns

Elissa Newman\*

School of Computer Science  
Carnegie Mellon University  
elissa@cs.cmu.edu

Position Paper for the Advanced Separation of Concerns Workshop at ICSE 2001  
April 2001

## Abstract

*Concern Oriented Programming seeks to decrease the cost of anticipated areas of change by isolating them from the rest of the code base. Although this reduces the cost of making anticipated changes, un-anticipated changes may result in higher costs of program understanding during the reengineering process.*

*Preserving the “chain of evidence,” a software design record semantically linked with annotated code, may help reduce the cost of program understanding and evolution. The semantic facets we emphasize deal with “mechanical” program properties such as concurrency, aliasing, effects, uses, and the like. Answers to questions about these facets enable many program transformations and restructurings.*

*We use the chain of evidence to create “localizing views,” which are dynamically created views addressing a particular concern. These allow a developer to view only the parts of the software artifact that will help in the task at hand. Our vision is that changes can then be made in any of the software life-cycle elements and eventually, with tool assistance, the changes will be propagated through the system.*

## 1. Introduction

Many researchers agree that traditional code and design representations suffer from the “tyranny of the dominant decomposition,” namely that the programs are decomposed in a single main format—as a set of objects, functions, and source files [1]. We would like to be able to access and organize code by additional concerns such as feature, non-functional requirements, or more global properties such as concurrency.

This area of research is called “Separation of Concerns” or “Concern-Oriented Programming” (COP). Some of the research in this area is leading to tool support for adding concerns to existing code such as

Multi-Dimensional Separation of Concerns [2] and Aspect Oriented Programming [3]. Other research identifies the types of concerns that programmers might be interested in isolating in the code for better understanding or in order to make specific changes.

Just as subroutine abstraction was introduced to foster greater understanding of purely sequential code, COP is attempting to increase the level of abstraction in programs to allow for isolation and increased understanding of concerns.

The premise of this paper is that the addition of syntax and purely structural reorganizational schemes to existing or future code bases will not be sufficient to allow for scalable understanding of concerns in relation to the program as a whole. Our guiding hypothesis is that the addition of semantic information that is explicitly linked to code elements will enable the definition of a broader range of localizations, that, in turn, enable greater understanding.

## 2. Program Understanding and Software Maintenance

Most software engineering is software evolution. The activities that comprise what is traditionally called the maintenance phase are said to total as much as 80% of overall software development costs [4]. The process of software evolution addresses repairing of flaws, enhancing functionality, and adding interactions with other systems. Reports assert that half of this cost is in reverse engineering: an activity that is mostly program understanding and comprehension [5].

COP is directed at decreasing the costs of software maintenance for pre-defined concerns in systems under development or in the process of being re-engineered. COP accomplishes this by identifying a

set of concerns that relate to the software system at hand and by isolating these concerns from the rest of the code base. The separation of the concerns from the code base allows changes to be more localized. The intent is that the view provide exactly the information needed—that all critical elements are present.

Unlike Aspect/J [6] and Hyper/J [7], we seek to provide task-based views that are designed for performing interactive semantic-based program revisions such as changing the local class hierarchy, changing an API, improving the use of concurrency, or making a change in data representation. Attempting to make such changes without adequate tools can require many hours of developer time spent comprehending the program structure and mentally performing dependency and binding analyses.

Part of our long term approach to this problem is to create the possibility of a large number of custom views that are dynamically configurable and that do not require code to be rewritten or repartitioned due to changes in the concern space. The first step in accomplishing this is to perform semantic manipulations on highly annotated abstract semantic structures (ASTs) [8]. Compilable code can be produced according to different parameters and desired semantic configurations of the program.

Obviously, program comprehension can be enhanced by providing information beyond source code. Design information, requirements, documentation, and annotations must be kept consistent with the code to provide the rationale necessary for adequate program understanding with respect to semantically non-trivial concerns. The complexity of large software systems may be due to the interactions of larger scale entrenched design decisions whose manifestation is pervasive in code. With the presence of the design record and semantic links, it may be possible to trace these decisions back to their source in order to make changes at the appropriate level of scope.

### 3. Shape of a Possible Solution

Black and Jones [9] propose a notion of “Perspectives” based on a more abstract notion of programs that they call the abstract program structure (APS). The APS allows for many views to be created dynamically from the semantic meaning of the program independent of its implementation.

I am exploring task-specific localizing views based on semantic analysis, annotation, and transformation [10, 11]. These views are intended to allow the developer to limit his or her consideration to those areas

of the system that relate to a contemplated analysis or change. Without these views, developers would have to perform restructurings to make structural changes. These views enable partial automation of the restructuring currently employed by developers to localize structural change in software by bringing together the code fragments and deduced properties that are significant to the change. The views would provide support for semantically dependent concerns such as data representation, invariants, and concurrency. To provide these views, we need a tool that can retain the design and code information necessary to complete the task.

Similar to the APS of Black and Jones, the code in this system would not be directly manipulated, but rather accessed through an abstract representation that can then be retranslated into code. This is transparent to the developer.

The system I envision would ultimately interlink code, design, requirements, documentation, and other artifacts in a way that enables a change in one area to propagate further changes to other affected areas in the software design record. In the long term, this will create a multi-faceted software artifact in which changes to any of the traditional software process development products can be made through the view (e.g. design, code, etc.) most suited to expressing the change. We call this idealized design record the *chain of evidence*. Artifacts that we would like to link include requirements, use cases, structural design information (class diagrams, etc.), behavioral design information (sequence diagrams, activity diagrams, state charts), other UML diagrams, test cases, user manuals, code annotations, and comments (design rationale).

The framework of the system builds on formal code annotations, analyses, and transformations. Within the framework, the user can create multiple views (representing concerns) that are not predefined (cf. Hyperspaces [2]) and are not external to the code base (cf. Aspects [3]). Unlike SOP [12], localizing views are intended to function on many levels of change—from changes made entirely within a method body to changes made to an entire API or program structure. Unlike AOP, localizing views permit large numbers of meaning-preserving code-level structural renderings of a system, allowing dynamic revisions and restructurings based on tasks. These views can be created dynamically based on the configuration of the system or code that is needed for the current view or change scope.

Constructing localizing views will incur some costs. The compromise is that views are created through an interaction between a software system and a tool. Complex views may require work to create the inter-linking of code, annotations, and design. But once created, consistency among view elements can be tracked semi-automatically as updates occur.

## 4. The Fluid Project

The Fluid project is presently focusing on code, mechanical design annotations [13], and low-level design representations and diagrams for inclusion in the system. Semantic concerns now addressed include known uses of aliased and unique references, uses of objects, state mutability, and others [14]. My work is presently focused on creating concurrency diagrams that relate to the concurrency annotations that have been developed by other members of the team. Next, I plan to focus on dependency tracking and truth maintenance research to provide semantic links for our system.

A recent specific focus of the Fluid project is on Java-style concurrency. We want to provide concurrency views composed of code, annotations, diagrams, and other documentation such as policy matrices. We work with the notions of regional shared state, thread identification, and concurrency policy. We are now creating annotations with corresponding diagrams that will help us answer basic concurrency questions. These are questions such as:

- What is the shared state in concurrent programs?
- Which threads/objects/methods can access this shared state?
- How is the shared state protected?
- Which threads can/cannot execute this code?
- Which methods can execute concurrently?

Later views would allow for concurrency policy to be viewed and changed, as well as the amount of synchronization to be viewed and controlled. With the annotations in place in even parts of the code base, analyses can be performed to provide information about race conditions and potential deadlocks, as well as thread safety.

The Fluid tool prototype has been under development for several years. It has been used to support research on annotations, analysis, and transformations in Java code. It also supports fine-grained versioning, and support for multi-user coordination is being developed.

## 5. Conclusion

We are exploring an approach to design information management based on a vision of an interlinked design record that encompasses semantic information beyond code, with merged dependency relationships. Our initial steps are focused on “mechanical” program properties in general, and Java concurrency in particular.

Our success will be determined by what kinds of questions can be answered about a system and what kinds of programmer-guided program transformations can be supported. The code in the system is manipulated in an abstract form that includes semantic annotations, allowing multiple semantically meaningful views to be created based on the evolution task to be performed.

## 6. Acknowledgements

Much of the work described is jointly undertaken with members of the Fluid team: John T. Boyland, Edwin Chan, Aaron Greenhouse, Matthew Messner, William Scherlis, and Dean Sutherland.

## References

- [1] M. Chu-Carroll. *Separation of Concerns: An Organizational Approach*. In OOPSLA 2000 Workshop on Advanced Separation of Concerns, September 2000.  
<http://trese.cs.utwente.nl/Workshops/OOPSLA2000/>
- [2] P. Tarr, H. Ossher, W. Harris, and S. Sutton. *N Degrees of Separation: Multi-dimensional separation of concerns*. In Proceedings of the 21<sup>st</sup> International Conference on Software Engineering, 107-119, May 1999.
- [3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. *Aspect Oriented Programming*. In Proceedings of ECOOP'97, Lecture Notes on Computer Science 1241, 220-242, Springer-Verlag, 1997.
- [4] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- [5] R.K. Fjeldstad and W.T. Hamlen. *Application Program Maintenance Study: Report to our Respondents*. In 48<sup>th</sup> GUIDE.
- [6] AspectJ™ Web site: <http://www.aspectj.org>
- [7] HyperJ™ Web site:  
<http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm>
- [8] W.G. Griswold, M. I. Chen, R. W. Bowdidge, and J.D. Morgenthaler. *Tool support for planning the restructuring of data abstractions in large systems*. In FSE'96.

- [9] A. Black and M. Jones. *Perspectives on Software*. In OOPSLA 2000 Workshop on Advanced Separation of Concerns, September 2000. <http://trese.cs.utwente.nl/Workshops/OOPSLA2000/>
- [10] W.L. Scherlis. *Analytic Views: Embedded Components and Managed Change*. Workshop on Software Evolution at ICSE'98.
- [11] W.L. Scherlis. *Structural Views, Structural Evolution, and Product Families*. From ARES Workshop on Product Families and Software Evolution, Springer-Verlag, 1998.
- [12] H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds. *Subject-oriented programming: Supporting decentralized development of objects*. In the 7<sup>th</sup> IBM Conference on Object-Oriented Technology.
- [13] E.C. Chan, J. T. Boyland, and W.L. Scherlis. *Promises: Limited Specifications for Analysis and Manipulation*. In ICSE'98.
- [14] A. Greenhouse and J. Boyland. *An Object-Oriented Effects System*. In ECOOP'99.