

Proposed Investigation of Aspect-Oriented Brainwashing

Dr. Shailendra Narayan Singh

Department of Computer Science and Engg.
Amity School of Engineering and Technology
Amity University Uttar Pradesh Noida Campus

ABSTRACT

In this paper the Aspect-Oriented Programming (AOP) systems is that they allow programming by making Assessments of programmatic assertions over programs written by programmers ignorant to such assertions. But we have found many programming troubles for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the completion of those design decisions to be scattered throughout the code, resulting in “tangled” code that is excessively difficult to develop and maintain. We present an analysis of why certain design decisions have been so difficult to clearly capture in actual code. We call the properties these decisions address aspects, and show that the reason they have been hard to capture is that they cross-cut the system’s basic functionality. We present the basis for a new programming technique, called aspect-oriented programming that makes it possible to clearly express programs involving such aspects including appropriate isolation, composition and reuse of the aspect code. The discussion is rooted in systems we have built using aspect-oriented programming.

Keywords: Aspect-oriented design, design quality, design assessment, design quality assurance

1. Introduction

Object-oriented programming (OOP) has been presented as a technology that can fundamentally aid software engineering, because the underlying object model provides a better fit with real domain problems. But we have found many programming problems where OOP techniques are not sufficient to clearly capture all the important design decisions the program must implement. Instead, it seems that there are some programming problems that fit neither the OOP approach nor the procedural approach it replaces.

This paper is about Aspect-Oriented Programming (AOP) qua programming language. We are interested in determining what makes a language AOP.

Aspect-Oriented Programming [1], is an emerging family of techniques which address a central limitation of Object Oriented Programming, namely the breakdown of modularity that results when the core business logic is combined with other “concerns”, such as security, data persistence, transactions, high availability, logging, error handling, etc. as part of the implementation process. OOP is a success because it effectively decomposes the core business logic into minimally-coupled modules and classes, which can be developed separately and then combined to create applications. This is a “divide and conquer” strategy.

The core business logic is represented by an object structure. However, additional logic for the other concerns cuts across the boundaries of the core objects. Therefore, these concerns are called crosscutting concerns.

Hence, in the source code of an application, the pristine object model is often obscured by code that implements the crosscutting concerns. This tangling of code reduces modularity, increases complexity, and impacts quality. Maintenance becomes difficult because modifications in crosscutting concerns must be made across many code modules. In short, crosscutting concerns undermine the benefits of OOP.

The vision of AOP is to restore overall modularity by extracting the crosscutting concerns, modularizing them as aspects, and then weaving them back into the application at well defined locations, called join points.

Aspects can be discussed for applications that are not object-oriented, but AOP and OOP complement each other nicely. In a sense, AOP is a “superset” of OOP. The core business logic and the aspects can be developed separately using OOP techniques, suitably extended with AOP concepts. Then, AOP describes how to weave the logic and aspects together to create the application. OOP models provide a number of natural join points into which aspects can be woven.

Aspects appear in all levels of the development process. For example, security, persistence of data, transactions, and high availability concerns appear at the requirements level all the way down to the implementation. Conversely, logging and language-specific error handling are implementation concerns. AOP is a useful tool for quality-assurance issues. White-box testing can be implemented using test aspects that are woven into applications to drive test cases, inject faults, and examine program state.
Programming languages are about writing a structure of statements that a compilation or interpretation process will elaborate as a series of primitive directions. (The directions themselves will be a finite text, though their interpretation may be unbounded.) The earliest computer programs had a strict correspondence between the program text and the execution pattern. Generally, each programming language statement was unitary and local—unitary in that it ended up having effect in precisely one place in the elaborated program, and local in that it is almost always proximate to its neighboring statements.

We are addressing the structural essence of AOP here, not its application—somewhat similar to the difference between defining Object-Oriented Programming (OOP) systems in terms of polymorphic methods and inheritance, versus waxing euphoric about objects as the appropriate way to model the world. A minor exception to locality and unitary was the introduction of types. Type declarations can have non-local and quantified effects such as type-coercion and type consistency checks. Types in conventional languages are thus an example of a built-in separate concern.

Inheritance in object-oriented programming (OOP) was the second important introduction of non-locality. Executing inherited behavior is non-local. There are two different fashions of inheriting behavior, send super and mixins. Send-super systems like Java and Smalltalk allow the programmer to explicitly invoke the behavior of its parent class or classes, without knowing exactly what behavior is being invoked. Adding behavior to classes higher in the class structure allows a limited form of quantified program statements—that is, statements that have effect on many loci in the underlying code. For example, suppose we wish to introduce a “display” aspect to a program about simulating traffic movement. We will want to make quantified statements like “Whenever something moves (executes its move method), the screen must be updated.” Imagine that all things that move are descendants of the class of moveable-object. We can accomplish this with send-super inheritance, if we have a cooperative base-class programmer—that is, one who will consistently follow directions. We make the behavior of the move method in movable-object be the display update and request the programmers of derivative classes to invoke send-super at the end of their code. This requires the derived class programmers to know that they have to do something, but relieves them of having to know what exactly it is that they have to do. We’re also restricted with respect to the locus of behavior—we can ask programmers to do the send-super at the start of their code, or at the end, but our directions probably need to be consistent throughout the system.

In using inheritance to achieve aspects, single super class inheritance systems require all aspects to match the class structure of the original program, while multiple inheritance systems allow Assessments independent of the program’s dominant decomposition. Mixings with multiple inheritance are thus a full aspect-oriented programming technology. AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs.

We want to be able to say, “This code realizes this concern. Execute it whenever these circumstances hold.” This breaks completely with local and unitary demands—we can organize our program in the form most appropriate for coding and maintenance. We do not even need the local markings of cooperation. The weaving mechanism of the AOP system can, by itself, take our quantified statements and the base program and produce the primitive directions to be performed.

3. Assessments

AOP is thus the desire to make programming statements of the form

In programs P, whenever condition C arises, perform action A.

over “conventionally” coded programs P. This implies three major dimensions of concern for the designer and implementer of an AOP system:

Assessments: What kinds of conditions C can we specify?

Interface: What is the interface of the actions A.? That is, how do they interact with base programs and each other?

Weaving: How will the system arrange to intermix the execution of the base actions of program P with the actions A.?

In an AOP system, we make quantified statements about which code is to execute in which circumstances. Over what can we quantify? Broadly, we can quantify over the static structure of the system and over its dynamic behavior.

3.1. Static Assessments

The static structure is the program as text. Two common views of program text are in terms of the public interfaces of the program (typically methods, but occasionally also public variables) and as a parsed-program as abstract syntax tree. Black-box AOP systems quantify over the public interface of components like functions and object methods. Examples of black-box systems include Composition-Filters [2], synchronization advice. A simple implementation mechanism for black-box AOP is to wrap components with the aspect behavior.

Clear-box AOP systems allow assessments over the parsed structure of components. Examples of such system include Aspect, which allows (among other things) quantifying over both the calling and accepting calls in subprograms [3], and Subject-Oriented Programming, whose composition rules allow quantifying over elements such as the interpretation of variables within modules.

Clear-box and black-box techniques each have advantages and disadvantages. Clear-box techniques require source code. They provide access to all the (static) nuances of the program. Black-box techniques are typically easier to implement (in
environments like Lisp they can be downright trivial) and can be used on components where the source code is lacking.

3.2. Dynamic Assessments
Dynamic Assessments is tying the aspect behavior to something that happens at run-time. Examples of such things are

• The raising of an exception.
• The call of a subprogram X within the temporal scope of a call of Y.
• The size of the call stack.
• Patterns on the history of the program (e.g., after the “try password” routine has failed five times, with no intervening successes.)

Keep in mind that the abstractions most programming languages present about the structure and execution of a program are only a subset of the possible available abstractions: Scheme allows a programmer to capture the “current context” and re-invoke the current behavior[4]. The ability to program with respect to such properties is an aspect of programming language design. Even if such elements are absent in the underlying language, an aspect language may still allow assessments over them.

4. Implementation issues
Several other dimensions of aspect-oriented language design bear mentioning, though we lack the space to discuss them more fully. The popularity of Java byte-coding allows the possibility of quantifying with respect to byte code. We can clearly quantify over byte-code features demanded by the JVM architecture (e.g., method entry points)[5]. To the extent that recognizable features of the surface structure are realized in recognizable ways in the byte code, we can quantify over these features and work with class files. Good luck if someone used a different compiler. The call of X within the context of Y problem is an instance of the “jumping aspect” problem.

• Context. What context of the underlying program can action A reference? Clear-box systems can go so far as to make the aspect action be a seamless part of the resulting code, though tangling the aspect expression with the specifics of an implementation raises questions of “real” separation of concerns[6].

• Assessments scope. Over what elements in the program can one quantify? Typical choices include all methods on all objects in this class, all methods on all objects in this package, all methods with a given name within the objects of all subclasses of a class, and instance level variations of the above.

• Dynamic Assessments. Can quantified assertions be made in a running system, dynamically adding and removing actions?

• Incomplete Obliviousness. Though we want the application programmer to be mostly oblivious of aspect system, there are times that the application needs to communicate with the aspects. For example, consider an action that provides a higher quality of service to higher priority tasks. The application needs a mechanism for specifying the current task’s priority.

• Action interaction. How do actions communicate? Both different actions on the same locus and the same action on different loci may need to exchange information. (For example, the authentication and access control aspects on method M of object X may need to interact. Similarly, the authentication aspect on all objects may need to share a common space.)

• Relative aspect orderings. How do we specify the order of multiple actions that apply to the same locus?

• Inconsistent aspects. Sometimes one action may violate the semantics of another. For example, an action that logs progress may violate the “all-or-nothing” semantics of a transactional action. An AOP system may have some linguistic mechanisms for warning or forestalling such inconsistencies.

• Weaving. In an implementation sense, how do we arrange for the behavior of the actions to be intermixed with the behavior of the base system code? The answer may include compile-time weaving and altering the run-time interpretation process.

5. Conclusion
In this paper we have identified AOP with the ability to assert quantified statements over programs written by oblivious programmers. This implies:
AOP is not about OOP. OOP is the current dominant programming language technology. Most implementations of new language ideas are done in the context of OOP. However, “oblivious quantification” is independent of OO concepts [7]. Therefore, it would be perfectly reasonable to develop AOP for a functional or imperative language.

AOP is not useful for singletons. If you’ve got an orthogonal concern that is about exactly one place in the original code, and you’re sure that that orthogonal concern will not propagate to other loci as the system evolves, it is probably a bad idea to use AOP for that concern. Just write a call to the aspect procedure into that one place in the code, or permute the source code in whatever way you thought necessary to achieve the aspect. The quantity of communication (among programmers) required to do aspects in general probably equals the quantity of communication required to modify just one program. The homogenized code leaves no ambiguity about what’s happening, but may be less clear than what’s happening as written as separate aspects.

Better AOP systems are more oblivious. They minimize the degree to which programmers have to change their behavior to realize the benefits of AOP. It’s a really nice bumper sticker to be able to say, “Just program like you always do, and we’ll be able to add the aspects later.”

6. References


AUTHOR

Dr. Shailendra Narayan Singh

Dr. Shailendra Narayan Singh working in Amity University Uttar Pradesh and having experience in teaching and administration. He did his B. Tech, M. Tech, Ph.D. in the field of Computer Science and Engineering. His research area is software reusability for aspect oriented and object oriented software.

Dr. Shailendra Narayan Singh is having more than fifteen years of teaching experience in the academic field of computer science and engineering in Indian and Abroad Universities. Dr. S N. Singh is author of four book titles Operating System, Fundamentals of Internet, Software Project Management Part-I and Part-II. Teaching is his first love and participates in various conferences, seminars in National and International.