

Aspects For Legacy Applications

Summary of Dagstuhl Seminar 06302

Leon Moonen
Delft University of Technology
The Netherlands

G. Ramalingam*
IBM Research
New York, USA

Siobhán Clarke
Trinity College
Dublin, IE

Abstract

This paper provides a summary of the objectives, structure, and the outcome of Dagstuhl seminar #06302 on Aspects For Legacy Applications, held from July 26th to July 29th 2006 at Schloss Dagstuhl, Germany. The goal of the seminar was to bring together researchers from the domains of aspect oriented software development, software reengineering (with a focus on reverse engineering, program comprehension, software evolution and software maintenance) and program analysis to investigate how aspects can help us to understand, maintain, and transform legacy software systems.

Keywords: aspect orientation, programming languages, software evolution, program analysis, reverse engineering, aspect identification (aspect mining), program understanding, refactoring, software reengineering, program transformation.

1 Motivation

Programming languages and programming technologies have continued to evolve since the birth of computing, bringing significant improvements to programmer productivity and addressing software engineering issues and concerns that have become apparent over time. Unfortunately, the vast majority of the applications in use today have not benefitted from many of these advances. For example, large-scale legacy applications written in Cobol still constitute the computing backbone of many large businesses. Such applications are notoriously difficult and time-consuming to update in response to changing business requirements. Similarly, a large body of technical software and system software is written in C. Maintaining and evolving such software is even harder.

*Current affiliation: Microsoft Research India, Bangalore, India

One of the fundamental reasons that such legacy applications are hard to understand, maintain, evolve, or reuse, is the scattering and tangling of code fragments addressing many different concerns. While this problem can show up even in well-written object-oriented code, this is an even bigger issue with applications written in legacy languages such as Cobol and C.

Aspect-oriented software development (AOSD) has emerged over the last decade as a paradigm for *separation of concerns*: avoiding the aforementioned problem of code scattering and tangling. It seeks to achieve a separation of concerns even for those *cross cutting concerns* that are difficult to decompose and isolate with earlier programming methodologies. Aspect-oriented programming permits programmers to organize code, design and other artifacts together in a more logical, natural, way according to the concern they address. A complete application is produced by composing, or *weaving*, separate concerns together.

Aspect-oriented software development promises significant benefits in the areas of software comprehension, maintenance and evolution and could therefore play an important role in the revitalization of aforementioned legacy systems. However, most of existing AOSD approaches seem to have focused primarily on identifying, specifying and implementing aspects for systems that are developed from scratch. Fully exploiting AOSD in the context of legacy software systems imposes different requirements and constraints and has its own set of challenges and open problems that need to be solved.

The seminar focused on the following question: how can aspects help us to understand, maintain, and transform legacy systems? The term “legacy systems”, in this context, includes both “classic legacy” (applications written in languages such as Cobol and C) as well as “modern legacy” (applications written in object-oriented languages such as C++ and Java) and “future legacy” (applications that were developed using AOSD and need evolution).

2 Goals

The goal of the seminar was to bring together researchers from the domains of aspect oriented software development, software reengineering (with a focus on reverse engineering, program comprehension, software evolution and software maintenance) and program analysis to investigate how legacy systems can benefit from aspect oriented software development techniques, discuss the state of the art in aspect identification and refactoring, identify open research questions and establish a roadmap for joint research in this area.

The following topics and problems were considered relevant to the topic, though not all of these were addressed in the seminar due to time limitations:

- Idioms and design patterns in (different kinds of) legacy systems, which serve to identify common *concerns/aspects* in legacy systems.
- Use of aspects as views of a legacy system to improve ones understanding of that system.
- Recovering aspects (i.e., identifying the lines of code that implement a particular concern) from legacy systems.

- Aspect languages and weavers for legacy languages (especially non-object-oriented languages).
- Using aspects to guide reimplementaion of a legacy system (also known as 'early aspects').
- Supporting migration from legacy implementation to an aspect-based implementation (either in the same or different base programming language).
- Evolution of systems that were developed using aspect oriented software development techniques (the legacy systems of the future).

3 Format and organisation

To familiarize the different communities with each other's background and ways of thinking, the seminar started with three invited overview talks:

- The presentation "*The problem with Legacy Applications*" by Hausi Müller (University of Victoria, CA) started by summarizing major avenues of legacy systems research and highlighting selected success stories. The presentation then discussed the future of legacy systems which morph from being systems of systems into Ultra-Large-Scale (ULS) systems and socio-technical ecosystems. The presentation then concentrated on the software complexity problem inherent in these ULS systems and how autonomic computing technology could help alleviate this problem.
- The presentation "*A Gentle Introduction to Aspect Oriented Programming*" by Oege de Moor (Oxford University, UK) started off with an introduction to the concepts of aspect oriented programming. After a brief discussion of the advantages and disadvantages of aspects, two recent developments were presented, namely pointcuts that match on complete program traces, and pointcuts that capture semantic properties (as opposed to pointcuts that are entirely name-based).
- The presentation "*An Overview of Aspect Mining*" by Kim Mens (Univ. Cath. de Louvain, BE) offered an in-breadth survey and comparison of current aspect mining tools and techniques. The presentation focused mainly on automated techniques that mine a program's static or dynamic structure for candidate aspects and presented an initial comparative framework for distinguishing aspect mining techniques, and assess known techniques against this framework. This comparison helps aspect mining researchers to identify remaining open research questions, possible avenues for future research, and interesting combinations of existing techniques.

The remainder of the seminar was characterized by alternating sessions of short (15 minute) technical presentations and larger working sessions in which participants discussed particular topics of interest in breakout groups.

Technical Sessions The talks in the technicals sessions were:

- *Revitalizing legacy business software using aspects* by Kris De Schutter and Bram Adams

- *Combining and Evaluating Aspect Mining using Crosscutting Concern Sorts* by Marius Marin and Leon Moonen
- *Language Reverse Engineering - Separating concerns in legacy applications* by Jean-Marie Favre
- *Mining Aspects from CVS History Using Concept Analysis* by Silvia Breu
- *Investigating aspect opportunities in a 15MLOC industrial C system* by Magiel Bruntink,
- *Coarse-grained concern management* by Martin Robillard,
- *Toward a datalog-based programming model for composing independently-specified updates in reactive business applications* by John Field
- *Mining Control Flow Graphs for Crosscutting Concerns (But Finding Delegation)* by Jens Krinke
- *Software Reengineering at the Architectural Level* by Rui Correia
- *Graph transformations for software evolution/refactoring* by Tom Mens
- *Mining Aspects in Legacy System Documentation* by Americo Sampaio.

Working sessions The topics to be discussed in the working sessions were collectively established in the first working session. During this session, each participant came up with three topics for discussion. These were then pairwise ranked in a distributed fashion by teams of participants that changed over a number of iterations. Finally, the participants collectively grouped together similar topics to produce a ranked list of open issues of interest, as to identify a research agenda. The first five topics in this agenda, shown below, were addressed by breakout groups during the remaining working sessions.

- Representation of crosscutting concerns in legacy systems: (Champion: Martin Robillard, 10 participants)
 - Aspects in real-life legacy languages such as cobol. (27 points)
 - What are examples of aspects in legacy systems? Are they expressible with current technology? (24 points)
 - Do legacy languages have a comparable join point model? Do we need to extend the aspect language?
 - Alternative ways to better cope with crosscutting concerns than migration to AOP? (26 points)
 - Specifying identified aspects in legacy code; should be lightweight, application/domain specific, and low-tech. (27 points)
 - Cross-cutting concerns should not be refactored; instead, the IDE should provide views corresponding to aspects. (24 points)
 - Visualisation of aspects.
- Program Comprehension: (Champion: Andy Zaidman, 5 participants)

- Concern understanding. Information needed? Improve or hinder understanding/evolution? Easier to maintain? (27 points)
- Is there a possible reduction in program comprehension as a result of inclusion of aspects? (25 points)
- Measurement and evaluation of AOSD: (Champion: Tom Mens, 4 participants)
 - Evaluation of AO as a paradigm: how do we know that it produces “better systems” (26 points)
 - What are good metrics for assessing the quality of AOSD systems? (25 points)
 - Prove AOP is better and quantify improvement? (24 points)
 - Do we really need aspects? How do we know? (24 points)
- Evolution of aspects: (Champion: Silvia Breu, 8 participants)
 - How do aspects evolve over time? (26 points)
 - Evolution of aspects as a consequence of source code evolution (26 points)
 - How can we avoid aspects becoming fragile due to evolution? (24 points)
 - Maintenance of aspect-based programs. How can we ensure that aspects do not complicate software maintenance, instead of simplifying it? E.g., the aspect version and weaved versions as views between which user can switch. (30 points)
 - Do aspect-oriented languages decrease maintenance effort? (25 points)
 - Evidence that AOP-based software will be easier to maintain? (27 points)
- Aspect mining: (Champion: Rui Correia, 5 participants)
 - Validation of aspect mining techniques (24 points)
 - In which sense does prior knowledge of aspects influence the methods/techniques to recover them? (24 points)
 - Towards a benchmark set for aspect mining. (24 points)

4 Breakout Sessions

4.1 Evolution of Aspects

Aspects were introduced to simplify software evolution, but they introduce a new range of evolution problems. Have aspects introduced more evolution problems than they solve? Do we really want aspects? This breakout session focused on the following questions: How does aspect-oriented software evolve? What problems occur in such evolution? How can one manage/solve these problems?

Evolution of aspect-oriented software can include evolution of pointcuts, advice, base code, and interactions between aspects. Different modes of evolution (evolution of aspects and base code together, evolution of just base code, evolution of just the aspects) are possible, with correspondingly different problems that can arise in the context of such evolution.

Problems that can show up in evolution of aspect-oriented software include the following. The problem of co-evolution, or evolving the base code and the aspects. The problem of fragile pointcuts, where the specified pointcut ends up being incorrect when the base code evolves, resulting in erroneous captures or misses during the pointcut

matching. Interaction problems between different aspects, as well as the problems posed by dynamic loading on aspects, were also noted.

The solutions discussed include the following.

An explicit model of aspects, how aspects and code interact, and how aspects interfere, at a higher level of abstraction can be valuable. It improves readability, comprehensibility, reusability, evolvability and modularity. Perhaps more importantly, it allows for *conformance checking* of the base code against model, of aspects against the model, and for internal consistency of the model itself.

Such conformance checking is essential for detecting and correcting breaches of assumptions or conventions, especially during evolution. The conformance checking can be done either statically or dynamically.

Hardwired aspects are pretty useless when the base code evolves, while computed aspects are somewhat more robust during software evolution. A more expressive language for specifying cutpoints, one that enables more precisely capturing the intended semantics, would be of value in supporting evolution.

The more programs use techniques such as dynamic loading, the more relevant reflective techniques become to aspects.

The group concluded that much more research was required with regards to aspect evolution. There is a need for a systematic characterization of aspect evolution. Aspect evolution poses sufficient problems that the value of migrating to aspects is unclear. However, there was common agreement on potential approaches to these problems such as the use of explicit models and conformance checking, as well as more expressive pointcut mechanisms.

4.2 Measurement and Evaluation of AOSD

This breakout session focused on the problem of empirically evaluating AOSD. Little or no scientific evidence exists today that AOSD lives upto its promises. Some aspect-oriented metrics exist, but these are mostly simple adaptations of traditional object-oriented (code level) metrics. It would be desirable to empirically determine if applying AOSD techniques improves software quality.

This requires addressing several questions. What is software quality, and how can we measure it? How can we measure improvement in software quality? Code-level metrics (for coupling, cohesion, scattering, tangling) are one option for measuring quality. Process-level metrics, such as the time required to fix bugs, time to market, time required for testing, are another option. How well do these metrics, especially code-level metrics, apply to aspect-oriented systems?

Measuring improvement directly requires having two versions of the system, one that is aspect-oriented and one that is not. *AJHotDraw* and *jHotDraw* were presented as possible candidates in this regard. Measuring the time taken to fix important bugs (the time taken for the bug's status to change from *open* to *fixed*) in the two systems could then be used as a basis for evaluating the improvements due to the use of AOSD.

Some potential weaknesses of using such an approach were also discussed. These include differences between open-source developers and closed-source developers, the fact that the bugs may not be the same in the two systems, and that the notion of importance of a bug might be subjective.

Another possible experiment discussed was that of measuring metrics, such as those above, for a number of different systems, aspect-oriented and object-oriented, in the same domain (e.g., J2EE) and relying on statistical factors to make a comparison (even though the systems do not perform the same function).

4.3 Cross-cutting Concerns in Legacy Systems

One of the primary issues discussed in this breakout session was that of cross-cutting concerns in legacy systems. *Aspects* and *concern views* were discussed as potential solutions for working with cross-cutting concerns in a legacy system. An aspect, as in its usual sense, represents a syntactically disentangled module that explicitly encapsulates a cross-cutting concern, and using aspects in legacy systems involves transformation of the existing code into base code and aspects. A *concern view* looks like an aspect, but is just a view generated from the underlying code by the IDE. It was also noted that aspects and concern views are just two points in a spectrum. An updatable view, a view that can be modified by programmers so that the modifications are reflected back in the actual underlying code, would lie somewhere in the middle of this spectrum. Domain-specific languages and generative techniques can also be viewed as points in this spectrum.

A key question that arises in this context is that of the relative value of these different points in the spectrum in dealing with legacy systems. When is it better to use one approach over another? The following are some of the distinctions noted in this respect. Views can not be composed to construct a final application, but aspects can. Aspects are useful for composing scattered elements from one application for reuse in another. It was also argued that the degree of coupling between a concern and its context (what assumptions does the concern make about the base program) was a key factor in determining whether it was suitable for being refactored out as an aspect. An aspect should support modular reasoning about the cross-cutting concern.

On a slightly different note, the possibility of using an aspect to add functionality to a legacy system, and to adapt and configure it, while keeping the changes separate for deployment and source-code management purposes was mentioned.

The following are other questions raised during the discussions that were considered interesting and worth pursuing. How can one automatically derive a view from source code that represents a cross-cutting concern? How can one make such a view an updatable view? Can one capture information about programmers' navigation through a program and the changes made to it and use this information to identify cross-cutting concerns? Given a list of affected source code locations or fragments, how can one find a good, or optimal, or generalized pointcut from this information? How does one define a notion of interference between features or aspects or concerns? How does one deal with the heterogeneity inherent in a legacy system? What kind of metrics for aspects and concerns can we define for criteria such as composability and modular reasoning? It was also noted that aspects may sometimes just be making up for missing language features.

4.4 Aspect Mining

Aspect mining comes in several different flavors. The mining can be done over source code, or it could be done at a higher level of abstraction (such as requirement or architecture), which is then related to the underlying code. A taxonomy driven mining is also a possibility. Furthermore, the mining could stop with the identification of code realizing a cross-cutting concern, or it could include the subsequent step of refactoring and restructuring the code to isolate the aspect as well.

Adopting the classical definition of aspect, one specific version of the problem decomposes into the identification of concerns (and the code realizing them), and the identification of appropriate cut-points for the concern.

A distinction can also be made between supervised and unsupervised aspect mining. Previous knowledge of the aspects one is looking for makes mining easier. Task-directed aspect mining is worth considering as it may be easier.

This leads to questions about the goals and applications of aspect mining. Potential goals motivating aspect mining include program understanding, program migration, including conversion to AOP. Aspects may also help with testing.

Some other questions and issues that were discussed were: the need for benchmarks to evaluate aspect mining techniques, and for metrics or techniques to measure the effectiveness of proposed aspect mining techniques; the kind of static analyses needed to help mine aspects.

4.5 Program Comprehension

This breakout session focused on the question of whether aspects can help with program comprehension. The concepts of cross-cutting concerns leads to the potential for the use of *aspect views* to help comprehend a program. Thus, while one view could remove certain cross-cutting concerns to reveal just the base code and enable a better understanding of the base code, another view could illustrate the code with all interactions between the different concerns and base code visible, enabling a better understanding of the interactions between the concerns.

The following are some other observations that were made. The process of modularization itself can, sometimes, lead to a better understanding of a system. In a similar way, an attempt to identify and mine cross-cutting concerns in a system could itself help improve one's understanding of a system. Sometimes it is the interaction between a human and the computer (or the interaction between a human and another human), e.g. to mine cross-cutting concerns, that is of great value in understanding. The number of cross-cutting concerns could be a potential indicator of the difficulty of understanding a system.