Proceedings of the sixth workshop on

Software Engineering Properties of Languages and Aspect Technologies

(SPLAT 2008)

held at the Seventh International Conference on
Aspect-Oriented Software Development,
March 31 – April 4,
Brussels, Belgium

SPLAT 2008 Workshop Chairs:
Lodewijk Bergmans (University of Twente, Netherlands),
Erik Ernst (University of Aarhus, Denmark),
Kris Gybels (Vrije Universiteit Brussel, Belgium)

ACM International Conference Proceedings Series
ACM Press

The Association for Computing Machinery
1515 Broadway
New York New York 10036

ACM ISBN: 978-1-60558-144-6

Overview of the Workshop

The workshop consisted of three main ingredients: paper presentations, group work, and presentation and discussions about the results obtained during group work. The group work was based on examples that we asked all participants to prepare about one week before the workshop. The workshop started with a paper session session, after which the participants were divided into groups according to the nature of their examples, and the groups then started discussions. After the lunch break, everybody reconvened for a second paper session, again followed by group discussions. Finally, the groups presented their findings in a plenary session.

Paper presentations were based on a selection of the position papers by the program committee. There were 6 presented papers, selected from 11 submissions. The program committee contained the following members:

Mehmet Aksit	Lodewijk Bergmans	Johan Brichau	Shigeru Chiba
Erik Ernst	Bruno De Fraine	Kris Gybels	Stephan Herrmann
Robert Hirschfeld	Ralf Lämmel	Istvan Nagy	Klaus Ostermann
Kevin Sullivan	Kris De Volder		

Paper Sessions

Each paper was presented in a 20 minute slot, with 15 minutes for the presentation itself and 5 minutes for discussions. The papers themselves can be found later in this document, but at this point we wish to mention a few developments that played a special role during the workshop.

In the first paper presentation, C.H.P. Kim introduced the notion of materializable aspects, i.e., elements of the system which are conceptually parts of an aspect, but in fact directly included in the code. This is useful in cases where the desired join points are so complex that it becomes easier to "manually weave" the advice into the base code than it is to write several complex single-target pointcuts. The underlying trade-off is that of using an abstraction mechanism versus writing the simple, base-level code, and the point is that abstraction mechanisms should not *always* be used because there are cases where it increases the size and complexity of the software rather than reducing it. However, it is still useful to be able to check that the manually written code follows some rules, so the manually written code should be identified as an advice and checked for some notion of correctness. During the day we returned to this balance several times: using abstraction mechanisms to obtain solutions to problems automatically, versus using a specification to enforce that a manually provided solution is "nice".

Another topic that resurfaced several times was that of balancing negotiations among people with strict, mechanical approaches. In particular, S. Hermann argued that encapsulation as we know it is too inflexible, and that aspect-orientation in many ways is all about how to break the encapsulation. He argued that encapsulation should be negotiated among stakeholders, including the developers and users associated with each concrete software development project. This gives rise to a gradual and controllable violation of encapsulation - designated by the new word 'decapsulation' - and Hermann's proposal provides support for managing decapsulation in a flexible way over time.

Reasoning about systems where aspects are used is a well-known challenge, and we had two papers dealing with issues in this area. S. Agostinho discussed how to integrate contracts (pre- and post-conditions) with advice, and some issues which were raised during the following discussion established some connections: If contracts should be negotiated among stakeholders, how could they be kept meaningful (what good is an "almost" satisfied contract)? Could there be contracts

for inter-type declarations, too? How are contracts connected with advice ordering or more general notions such as advice composition? Could there be contracts for aspects per se, rather than only contracts for base code that takes aspects into account? Should it be possible to change a contract using an aspect? Next, A. Marot's presentation discussed a number of issues connected with composition of aspects, in particular how to specify the ordering of advice execution and other topics related to priorities and dominance. This raises a host of similar discussions as the previous paper, and also raises the issue of conformance: Marot's proposal allows for changes to contracts in such ways that the original contract is no longer enforced, which provides great flexibility but at the same time reasoning about the system may become harder or at least less modular.

C. Kaewkasi's presentation introduced an area which had not been covered so far, namely that of dynamically typed languages. The language Groovy is dynamically typed and contains sophisticated constructs related to closures, but at the same time it is executed on a standard JVM. This dichotomy is mediated by dynamic creation of methods and JIT compilation techniques.

Finally, E. Sousa presented his experiences with CaesarJ and family polymorphism, achieved through expression of a number of well-known design patterns in CaesarJ and comparison of the results with similar efforts in other aspect languages. In a number of cases, the design patterns turned out to disappear entirely, being reduced to a simple application of a feature in the language itself.

Group Work

During the workshop, participants discussed in a number of breakout groups on one of the focus -illities. Participants were asked before the workshop to prepare an example demonstrating why their chosen -ility is important or hard to solve. Five groups were formed, one on reusability, reliability/testability, and conceptual integrity, and two groups on evolvability.

The conceptual integrity group came up with the *SPLAT rule of conceptual integrity*: "continue decomposing up to the point that the cost of composition exceeds the benefits of separation". They took the well-known concept of model-view-controller to illustrate their view on conceptual integrity: a class Customer which implements all three concepts is one in which conceptual integrity is lost, to retain conceptual integrity, we except an implementation to have separate implementations for all the concepts involved in MVC. As such, conceptual integrity is linked in a positive sense to comprehensibility, evolvability and reusability. On the other hand, conceptual integrity was expected to negatively affect testability and performance. A question was raised on whether conceptual integrity and comprehensibility are actually different. While positively linked, we should still distinguish between them as a system may very well exhibit conceptual integrity while being not comprehensible, and vice-versa.

The reliability group concluded with stating a trade-off on explicitness of composition versus understandability of individual modules. According to this group, aspect technology has both a positive and negative effect on reliability of programs. The positive effect results from a reduction in code duplication and the general improvement of comprehensibility by clear separation of concerns. The negative effect results from not always knowing what aspects affect which parts of the code. One can state this as the surprising result that reliability and testability are not linked one-to-one: the pervasive use of aspects may make the individual modules more understandable and reliable, but the overall system may become harder to test.

The people in the first evolvability group discussed each other's research on aspect composition, the fragile pointcut problem and the verification of preconditions respectively. Antoine Marot

discussed the problem of improving the languages used for composing aspects in a declarative style, as also discussed in his presentation. Walter Cazzola, Marco Valente and Kouhei Sakurai proposed three alternative solutions for the fragile pointcut problem: Walter put forth a technique in which pointcuts are given as marker points in a blueprint activity diagram, Marco suggested the use of writing pointcuts on annotations that are introduced by aspects as well, while Kouhei advocated a solution that makes use of a program's test cases. Sergio Castro presented his work on verifying preconditions and invariants, which is more general than AOP but applicable to it as well and which he is currently pursuing using the intensional views technique.

The second evolvability group analyzed an example provided by Jan Wloka on the problems of refactoring base code in the presence of aspects. One question posed was whether it is technically possible for an automated refactoring tool to always rewrite pointcuts such that each refactoring is behavior-preserving. Some refactorings are clearly difficult because they may remove joinpoints, such as the inline method refactoring. No clear conclusion was drawn, as some group members felt it should be possible to add preconditions to each refactoring that disallow its application in such cases, while others felt this would not always be possible. An important secondary problem that was discussed however, is that even if a refactoring tool would be able to change pointcuts as necessary to preserve behavior, this may not always preserve the intent of the pointcuts. An extreme example would be if two different intents for a pointcut result in the same pointcut. Because of these problems, Stephan Herrmann made the remark he would like to see more focus on researching real-life productivity problems, rather than on extreme research challenges. Jacques Noyé suggested the concept of quantification should be researched on its own, independent of a specific aspect language. Kris Gybels and Jan Wloka pointed out the relevance of ontology or model-based pointcuts since otherwise it's hard for a pointcut to distinguish between a bank account class and an oil vat class except by relying on names, since the classes have the same operations.

Finally, the reusability group discussed several cases where reuse is very difficult to achieve. One case presented by Miguel Monteiro et al. was related to parallelization of computations involving large objects. The challenge is that the state of these objects needs to be partitioned into many small objects in order to facilitate parallel computations without shared memory; a further challenge is to be able to express this state partitioning process in multiple steps, such that objects already divided can be divided into even smaller parts. Current aspect abstraction mechanisms (as well as other kinds of abstraction mechanisms) seem to be unable to meet this challenge. The consequence is that this kind of refactoring must be done manually, which causes a large amount of redundancy. Another challenge which is also related to reuse and parallelization was presented by Erik Ernst, namely the challenge of writing software which may be deployed on architectures as different as the Cell Broadband Engine and a standard computer with one or two identical processors. This is difficult for many reasons, e.g., because the Cell is heterogeneous, uses low-level message sending and per CPU local storage rather than shared memory, and in general because the software seems to have to change in many detailed and irregular ways in order to fit one or the other hardware platform. Finally, Peter Kim used a database related example to illustrate a number of cases where a customization would be so detailed and specialized that abstraction would be useless: The program becomes larger and harder to understand using (currently known) abstraction mechanisms, as described along with Kim's paper. All in all, there is a rich supply of cases where reusability is made very difficult because the variants which are needed differ at such a detailed level and in so irregular ways that it cannot be made transparent by means of today's abstraction mechanisms.

Participants

Group 1: Erik Ernst, University of Aarhus, Denmark; Peter Kim, University of Texas at Austin, USA; Miguel P. Monteiro, FCT/UNL, Portugal; Muhammad Immad Naseer, University of British Columbia, Canada; Andre Santos, Tampere University of Technology, Finland; Joao Sobral, Universidade do Minho, Portugal; Edgar Sousa, Universidade do Minho, Portugal. Group 2: Sergio Agostinho, Fundacao de FCT, Portugal; Arjan de Roo, University of Twente, The Netherlands; Emilia Katz, The Technion, Israel; Tom Staijen, University of Twente, The Netherlands. Group 3: Lodewijk Bergmans, University of Twente, The Netherlands; Wilke Havinga, University of Twente, The Netherlands; Chanwit Kaewkasi, University of Manchester, United Kingdom; Chokchai Phatharamalai, Asian Institute of Technology,Thailand; Chenchen Xi, School of University of Manchester, United Kingdom. Group 4: Sergio Castro, Université Catholique de Louvain, Belgium; Walter Cazzola, University of Milano, Italy; Antoine Marot, Université Libre de Bruxelles, Belgium; Kouhei Sakurai, University of Tokyo, Japan; Marco Tulio de Oliveira Valente, PUC Minas, Brasil. Group 5: Kris Gybels, Vrije Universiteit Brussel, Belgium; Stephan Herrmann, Technische Universität Berlin, Germany; Jacques Noyé, Ecole des Mines de Nantes, France; Andre Restivo, University of Porto, Portugal; Veronica Uquillas Gomez, Vrije Universiteit Brussel, Belgium; Jan Wloka, Rutgers University, USA.

Contents