Software ENgineering

Proceedings of the 2nd workshop on linking aspect technology and evolution

T. Tourwé, D. Shepherd, A. Kellens, M. Ceccato

REPORT SEN-E0604 SEPTEMBER 2006

ISSN 1386-369X

Proceedings of the 2nd workshop on linking aspect technology and evolution

ABSTRACT
Software evolution lies at the heart of the software development process, and suffers from problems such as maintainability, evolvability, understandability, etc. Aspect-oriented software development (AOSD) is an emerging software development paradigm, that tries to achieve better separation of concerns. It is often claimed that this is actually beneficial for the maintainability, evolvability and understandability of the software. This workshop aims to investigate and explore this relationship between software evolution and AOSD. In particular, the workshop's objective is to study the impact of AOSD on software evolution on the one hand, and the impact of software evolution on AOSD on the other hand. Both subjects raise several interesting issues that could/should be addressed and studied in detail during the workshop: how does applying AOSD affect the quality of the application, and how does this help software evolution? can we quantify when applying AOSD solutions is beneficial? how do we recognise crosscutting concerns in existing applications? which techniques (f.e. refactoring, slicing) exist to separate them from the base code? should these techniques be extended with AOSD-specific concepts, and if so, how? how can we ensure the behaviour of the existing application is preserved? what (aspect) language constructs are needed to express the detected concerns? Answers to these questions are important, as there are many applications that continue to miss the advantages of AOSD, because appropriate tools and techniques are not sufficiently mature, and the advantages are not yet entirely clear. The workshop is specifically intended to address these questions, identify other interesting issues and bring together researchers from academia and people from industry working on applying AOSD techniques to already-existing applications.

1998 ACM Computing Classification System: D.2.7.
Keywords and Phrases: AOSD, software evolution

Proceedings of the 2nd Workshop on Linking Aspect Technology and Evolution

Held in conjunction with the 5th International Conference on Aspect-Oriented Software Development (AOSD), Bonn, Germany.
March 20th, 2006.

Proceedings of the 2^{nd} workshop on Linking Aspect Technology and Evolution

Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
Tom.Tourwe@cwi.nl

David Shepherd
University of Delaware
Newark, DE 19716
United States
shepherd@cis.udel.edu

Andy Kellens
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel
Belgium
akellens@vub.ac.be

Mariano Ceccato
ITC-irst
Via Sommarive 18, 38050 Povo, Trento
Italy
ceccato@itc.it

Abstract

This technical report contains the papers submitted to and presented at the 2^{nd} workshop on Linking Aspect Technology and Evolution (LATE), held in conjunction with the 5^{th} International Conference on Aspect-Oriented Software Development, Bonn, Germany.

1 Introduction

Software evolution lies at the heart of the software development process, and suffers from problems such as maintainability, evolvability, understandability, etc. Aspect-oriented software development (AOSD) is an emerging software development paradigm, that tries to achieve better separation of concerns. It is often claimed that this is actually beneficial for the maintainability, evolvability and understandability of the software. ?This workshop aims to investigate and explore this relationship between software evolution and AOSD.

In particular, the workshop's objective is to study the impact of AOSD on software evolution on the one hand, and the impact of software evolution on AOSD on the other hand. Both subjects raise several interesting issues that could/should be addressed and studied in detail during the workshop: how does applying AOSD affect the quality of the application, and how does this help software evolution? can we quantify when applying AOSD solutions is beneficial? how do we recognise crosscutting concerns in existing applications? which techniques (f.e. refactoring, slicing) exist to separate them from the base code? should these techniques be extended with AOSD-specific concepts, and if so, how? how can we ensure the behaviour of the existing application is preserved? what (aspect) language constructs are needed to express the detected concerns?

Answers to these questions are important, as there are many applications that continue to miss the advantages of AOSD, because appropriate tools and techniques are not sufficiently mature, and the advantages are not yet entirely clear. The workshop is specifically intended to address these questions, identify other interesting issues and bring together researchers from academia and people from industry working on applying AOSD techniques to already-existing applications.

2 Programme Committee

The following people agreed to serve on the LATE programme committee and helped to ensure the quality of the workshop.

- Magiel Bruntink, Centrum voor Wiskunde en Informatica, The Netherlands

- Yvonne Coady, University of Victoria, Canada
- Serge Demeyer, Universiteit Antwerpen, Belgium
- Arie van Deursen , Centrum voor Wiskunde en Informatica, The Netherlands
- Jens Krinke, FernUniversitŁt in Hagen, Germany
- Kim Mens, Universite catholique de Louvain-la-neuve, Belgium
- Miguel P. Monteiro, Escola Superior de Tecnologia de Castelo Branco, Portugal
- Lori Pollock, University of Delaware, USA
- Awais Rashid, Lancaster University, UK
- Martin Robillard, McGill University, Canada
- Stanley M. Sutton, IBM T. J. Watson Research Center, USA
- Paolo Tonella, ITC-irst, Italy
- Herman Tromp, Universiteit Gent, Belgium
- Wim Vanderperren, Vrije Universiteit Brussel, Belgium

3 Workshop Format

Based on the quality and maturity of the work, the programme committee accepted two tiers of papers:

- The authors of the first tier papers (*full presentations*) gave a presentation on their work, and we required all workshop participants to give them written feedback to supplement the discussion that occurs after each talk. This focus on feedback, along with a more thorough review process, is supposed to help these authors build toward a strong conference submission in the future.
- The second tier of papers (*poster presentations*) were presented in one of two poster sessions, with an extremely small number of other posters (2-4 at a time). These sessions were designed to facilitate smaller group discussions and more informal interaction with authors, leading to quality feedback and possible brainstorming. We encouraged participants to prepare a short demo of their work, if possible, and we identified interesting topics for discussion and comparison of the different works.

The workshop's schedule, including the full presentation and the poster presentation papers, can be found at the workshop's URL: http://www.aosd.net/workshops/late/2006/ .

4 Outlook

The remainder of this technical report includes all of these papers for easy reference. The presentations that accompany the papers can be downloaded from the *LATE* website, to be found at the following URL: http://www.aosd.net/workshops/late/2006/.

The success of this workshop was mainly due to the people that attended it, presented their ideas and participated in the discussions. We would like to thank all of these people and hope you enjoy reading their contributions.

AOP on the C-side

Bram Adams

Bram.Adams@UGent.be

SEL, INTEC, Ghent University, Belgium

ABSTRACT

Although aspect-oriented programming originally emerged to overcome fundamental modularity problems in object-oriented applications, its ideas have long been backported to legacy languages like Cobol, C, . . . As systems written in these languages are prime targets for re(verse)-engineering efforts, aspects can now be used for these purposes. Before applying dynamic analysis techniques on an industrial case study (453 KLOC of C) using aspects, we devised a list of requirements for possible aspect frameworks. In this paper we explain why no existing framework for C fulfilled all our requirements. We discuss the problems we encountered with Aspicere, our own aspect language for C. We also suggest points of improvement for future reverse-engineering efforts.

Keywords

aspect-oriented programming, legacy software, C, reverse-engineering, comparison study

1. INTRODUCTION

Like any new technology, aspect-oriented programming (AOP) came to life to solve problems inherent to the current state-of-the art, in this case object-orientation (OO), and more in particular Java [15]. Crosscutting concerns were indeed fundamentally ignored in the OO paradigm, so together with their accompanying terminology, aspects revitalized general purpose language research.

As the first waves of enthusiasm set off, people [11, 16] noticed that AOP's ideas were not necessarily tied to OO (and Java). Phenomena like scattering and tangling, the usual indicators for crosscutting concerns, equally (or probably likelier) arise in other OO-languages and less modular paradigms like procedural programming. Soon, every self-respecting language started to get its aspect language, even legacy languages like Cobol [16] and C.

Nearly every organisation is stuck with a battery of mission-critical software written in these old languages. These systems' internal structure and operations are typically no longer known, as the original developers, experienced maintainers or up-to-date documentation are not available anymore. They are inevitably hard to evolve as is, making it nearly impossible to cope with new requirements without prior re-engineering efforts. As aspect technology is getting more widespread in these areas, it can be leveraged to enable reverse-engineering techniques.

In a concrete reverse-engineering case study [23], we used aspects to apply dynamic analysis techniques to a medium-sized (453 KLOC) legacy system written in a mix of K&R- and ANSI-C. Although the role of the aspects there was very light (mere tracing), we envisioned applying AOP for more complex tasks. That is why we did not settle with some ad hoc AOP solution, but thought about relevant requirements for both our purpose back then as well as future efforts.

As it turned out, no aspect language fit the bill completely and we decided to roll our own. Eventually, we succeeded in our experiments, but new requirements came up along the way.

In this paper, we present our requirements gathered thusfar (section 2), as well as their applicability on all currently existing AOP-frameworks for C (section 3). Afterwards (section 4), we discuss possible points of improvement, before concluding (section 5).

2. REQUIREMENTS

To find the most suitable aspect framework for the case study of [23], we devised a set of requirements the preferred aspect framework needed to comply with. Although the functionality expected from the aspects came down to plain tracing, future re(verse)-engineering case studies would be more demanding. Our requirements were designed with this in mind and try to expect the worst.

There are two groups of requirements. The first one deals with specific properties of the whole tool chain, which immediately applies to the aspect framework used. The second one has to do with the specific reverse-engineering techniques used. Some of them are irrelevant here, but others indirectly demand certain functionality from the underlying aspect framework.

These are the tool chain requirements (T1–T5):

T1. Besides "nice" ANSI-C code, the legacy environments we will tackle obviously contain lots of non-ANSI (or K&R[1]) C code, and maybe compiler-dependent extensions too.

T2. The semantics of the original applications should remain intact.

T3. We do not want to delve into the original source code to gain knowledge before deploying our tools, as knowledge mining is exactly what we are after in the first place. I.e. the tools should not require special preparation or exploration of the source code.

[1]Kernighan & Ritchie-style code, after their seminal work [14].

	T1	T2	T3	T4	T5	A2	A3
AspectC	?	+	-	-	+	-	+
AspectC++	-	+	+	-	+	+	+
Aspicere	+	+	+	-	+	+	+
C4	+	+	-	-	+	-	+
WeaveC	?	+	?	-	+	+	+
μDiner	-	+	-	-	-	-	+
TinyC	N/A	+	-	+	-	-	+
Arachne	N/A	+	-	+	-	-	+
TOSKANA	N/A	+	-	+	-	-	+
TOSKANA-VM	+	?	?	-	-	?	?

Table 1: Overview of existing aspect frameworks for C and their relation to our requirements. A1 and A4 do not apply here, while T1 does not for the dynamic weaving mechanisms (except for μDiner), as these operate on binary code. A question mark means we could not decide due to a lack of tools, documentation or both.

T4. The existing build system should remain in place, with only minimal alterations. To refactor it, considerable knowledge of its current internals is needed, which again is lacking.

T5. The tools should be deployable in other environments (operating systems, platforms, compilers, ...), so that it can be validated against other case studies.

Requirement T2 only depends on the particular advice code used in the experiments, as obliviousness is one of AOP's hallmarks. This remark also partially affects T3, but we will see there is more involved here.

Each of the analyses of [23] had the following number of requirements (A1–A4):

A1. We need a well-covering execution scenario in order to obtain a representative result set.

A2. The data fed into the analyses needs to be sufficiently fine-grained. In the context of C this means individual procedure calls. We also need context information, e.g. on the relevant source modules.

A3. We need information about the call- and return-sequences of procedure calls, in order to get an accurate picture of the dynamic behaviour of the applications.

A4. The analyses should be able to deal with initially unknown amounts of trace data in a reasonable amount of time (i.e. they need to be scalable).

When rephrasing these requirements for reverse-engineering techniques in general, similar demands would arise. We can immediately exclude A1 and A4 from further discussion, as these apply to the specific analysis techniques used. This leaves us with seven requirements.

Based on these two groups of requirements, we will discuss the state-of-the-art aspect languages and mechanisms for C known to us at the time of writing.

3. CURRENT AOP-FRAMEWORKS FOR C

There are various ways to classify aspect frameworks, but we will do this per type of weaving mechanism in a chronological way. Table 1 gives a general overview of all the aspect frameworks mentioned and their relation to the seven requirements.

3.1 Compile-time weaving

All compile-time weaving aspect frameworks operate on the level of source code and transform in some way the advised source code and relevant aspects into regular C code before handing the woven code off to a normal C compiler. Their weaver acts as a source code preprocessor, fullfilling T5 because source code is one of the most portable things in the context of C. All compile-time weavers except for AspectC++ (section 3.1.2) expect a working Java run-time environment, while AspectC++'s PUMA-framework is written in (Aspect)C++.

This scenario has one important drawback, one we unfortunately experienced and had to deal with ourselves [23]: the aspect framework itself crosscuts the existing build system. In order to use a weaver in combination with existing compilers and other processors (for embedded SQL, ...), the existing makefile hierarchy has to be severely altered, violating T4. Directories with include-files, linking dependencies, ... can make things really hard. Unless there is only one compiler or tool used or the makefiles are automatically generated and have not been modified manually afterwards (very unlikely), this issue can not be solved easily with some clever trick or one single pass through the build process.

Without special tools like makefile refactorers, rewriters, ... manual adaptation of the build system is inevitable. Or maybe an aspect framework for makefiles could prove useful?

3.1.1 AspectC

The original AspectC [6] was targeted at tackling crosscutting concerns in operating system code. To do so, the designers started from the original AspectJ [15] and stripped it down by removing all unnecessary (OO-related) features. The resulting language has [5] a special aspect construct, function call/execution join points and pointcuts (A2), before/after/around-advice (A3) consisting of normal C code and well-known pointcuts like "cflow" and "within". Arguments of advised procedures can be assigned to variables and used in advice (not in the remainder of the pointcut expression though, cf. AspectJ).

Although variable argument lists can be abbreviated using "..", return types or procedure names can not (even not using regular expressions). Other means of selecting the right join points, like pointcuts based on structural, semantical, dynamic, ... information [19] are not provided. Hence, all procedures one wishes to advise must be declared by their exact name in the pointcut, as illustrated in [4]. Also, for every possible return and argument type a separate pointcut *and* advice have to be written down, which is unpractical (T3). Due to performance reasons, there is no thisJoinPoint-struct, so advices cannot access any join point context uncaught in the pointcut (A2).

Equally important, AspectC seems unmaintained since 2003 without any official releases, ruling it out as a viable aspect framework. That is why we do not know much regarding T1.

3.1.2 AspectC++

As (nearly) every compilable C program is also a valid C++ application, AOP languages for C++ could be applied to C base code. AspectC++ [21] is the most mature and general-purpose aspect language for C++ to date, with join point, advice and pointcut types comparable to AspectJ (A3), although more structural pointcut types like "callsto" and "reachable" are available. Advising variable accesses is not supported, because of C/C++'s pointer mechanism.

The aspect construct is in fact the C++ class construct with added pointcut and advice abilities. InterType Declaration (ITD) of new members in classes, structs and unions is also possible. Both advice and ITD are declared in the same way.

As templates have been a part of C++ for years, AspectC++ offered generic (and generative) advice much earlier than AspectJ did [18] (T3). Join point context (A2) like types and values of advised function calls, is easily accessible by a join point API (both static and dynamic parts) and applicable in the advice body.

AspectC++'s weaver[2] is based on the PUMA-framework, a C++ source code transformation system [21] (T5). It processes the whole program at once, demanding drastic changes to the existing build system (T4). Theoretically, ANSI-C code can be advised and subsequently compiled using a (sufficiently template-capable) C++-compiler (with some glitches), but tests with K&R-code failed however (T1). Full C support will only be provided starting from a release near the end of january 2006, feature by feature.

3.1.3 Aspicere

At the time of our case study, only tools for AspectC (alpha), AspectC++ and Arachne (section 3.2.3) were available. As both Arachne's (section 3.2) and AspectC's pointcut and advice model were too restrictive and AspectC++ only partially supported C, we were forced to design and implement our own framework. Since we did not realize the extent of the problems related to T4, we opted for a straightforward preprocessor approach, as C code itself guarantees the best portability of our weaver to other architectures.

Aspicere [1, 23] originally started out as WICCA [22], an AspectC-clone without an explicit aspect construct: aspects are simple compilation units with the extra power of containing advice. Experiments pointed out the T3 shortcomings of AspectC's aspect language, i.e. the inability to write down sufficiently, "generic" advice. Also, our (slow) parser did not cope with T1 and the fixed set of low-level pointcut primitives like execution and args prohibited easy addition of new pointcut types for other research efforts.

We decided to build a new aspect framework based on Logic Meta-Programming (LMP) [13], a template mechanism and a mature ANTLR C parser capable of parsing both K&R- and ANSI-code (T1). Basically, all pointcuts are made up of Prolog predicates and can be mapped one-to-one onto a Prolog rule. The predicates' logic variables can be bound to context information and reused within the pointcut expression to express certain constraints (unification). These bindings can then be applied within the advice body and the advice signature as some sort of template parameter (like C++ has) to denote types, caught arguments, weaving metadata, ... This results in simple, generic advice for C (T3). The around advice type and thisJoinPoint-like struct support A3 and A2. Lexical order of aspects and advice determines their precedence.

Besides the makefile anomaly, the case study of [23] showed that Aspicere's weaver currently works too slow and that its type inferencing capabilities are not perfect yet. Also, the ability of extending the collection of join point types was not really pursued in the current prototype[3], which currently only supports call join points. For the moment, we are experimenting with LLVM in a less naive preprocessor-setup, without the heavy demands of TOSKANA-VM (see also sections 3.3.1 and 4).

3.1.4 C4

AspectC's ideas of aspectizing UNIX-like operating systems using a static (preprocessing) weaver, live on in the aspect language called C4 (CrossCutting C Code) [10]. It aims at replacing the traditional patch system by a simplified AOP-driven, semantic approach. Aspects describe "modifications" to a base system on a higher level than the pure lexical patch(1)-tool does. However, chances of adoption are reversely proportional to C4's complexity.

Basically, the idea is that a programmer writes down advice (so-called woven C4) in situ (A2) in the base program, without any quantification. The C4 unweaver extracts the changes into a separate unwoven C4 file (a semantic patch) which can be freely distributed to everyone or (if needed) converted to a plain patch first. At compile-time, the unwoven C4 is physically woven with the base code.

This unwoven C4 file is in fact a (tweakable) classic aspect written in an AspectC-based dialect capable of ITD in structs/unions and advising global variables, but lacking the "call" and "cflow" pointcuts. The rationale here, is that one can always extract code blocks and encapsulate them into their own methods, which can be advised directly. It is clear that the woven C4 severely violates T3, while the unwoven version suffers from the same disadvantages as AspectC (T3, T4 and A2). No special thisJoinPoint-construct exists.

C4 is based on the XTC-framework [12], an advanced macro facility for C, that takes care of the physical weaving. Domain-specific language extensions like the C4-language (AOP domain) are declared as macro's and mapped onto specific plain C structuresand extra type information. C4's unweaver and (logical) weaver are still under construction[4].

3.1.5 WeaveC

WeaveC[5] is a very recent aspect language, in which both pointcuts and advice are written in XML-files. As it aims at becoming a general-purpose language, it has the same join point types as AspectC. Pointcuts are name-based (and wildcarded) and the dynamic pointcut types like "cflow" are only provided in the advanced version of WeaveC. Advice is prioritized to handle conflicts at joint join points using a priority level mechanism. Advice bodies or ITD of types or functions are written down in CDATA-elements of the XML-file. Currently, there is no around-advice yet.

Some predefined context variables (function name, argument types, ...) are available, and variables appearing near the join point shadow can be used freely in the advice body (A2). It is unclear whether generic advice is possible using these context variables (T3).

WeaveC's weaver is implemented in Java, and transforms the AST of the base program. In the advanced version, CodeSurfer[6] is used to perform the necessary analyses.

3.2 Run-time weaving

Then, there are a number of aspect languages with dynamic weavers, based on instrumentation libraries or binary rewriting techniques. As they act on binary code, they fulfill T4 (except for μDiner) and T1, but not necessarily T5, as platform-independence is questionable. Different operating systems use other binary formats and each processor's instruction set potentially needs a modified weaver.

All dynamic approaches provide some sort of around advice or a combination of before and after (A3). They support procedure calls and variable access join point types, but typically there is not enough context information available at the binary level (A2). Generic advice is impossible as all these approaches' languages require duplication of advice and a priori knowledge of return types (T3).

[2]http://www.aspectc.org/

[3]http://users.ugent.be/~badams/aspicere/

[4]http://c4.cs.princeton.edu/

[5]http://weavec.sourceforge.net/

[6]http://www.grammatech.com/products/codesurfer/

3.2.1 *μDiner*

μDiner [20] requires that declarations of advisable funtions and global variables are annotated as "hookable", and that a support library is linked with the created (base) executable, so T4 potentially shows problematic.

Aspects are compiled into shared libraries which are loaded into the advised base application at run-time. Hooks are then constructed to connect advised join point shadows with the right advices, and measures are taken to avoid freezing the base application. The weaving process is processor architecture-dependent (T5).

μDiner's aspect language features around advice (written in C) which can access the arguments of an advised procedure call, the current value of a global variable and (for variable assignment) the assigned variable. Types are hard-coded in the pointcut (T3). The familiar cflow-pointcut is replaced by an explicit nested call hierarchy, always ended either by a function call or by a variable access.

As Arachne supersedes μDiner (see section 3.2.3), no tools are available for μDiner.

3.2.2 *TinyC^2*

TinyC^2 [24] was developed independently from μDiner, and relies on the DynInst-instrumentation library instead, which uses the UNIX debugging API (ptrace). Aspects are transformed into self-contained C++ programs driving DynInst. Once compiled, one can dynamically advise a running C application.

Disadvantages of this approach, are the relatively high cost due to DynInst's use of trampolines and ptrace, and the impossibility of modifying arguments and return values.

The pointcut language of TinyC^2 only supports function call join points. There is both onentry- and onexit-advice (comparable to before and after), containing regular C code (A3). Pointcuts use prefix- or regular expression-based matching of procedure names. Available context includes explicitly bound function arguments and global variables, but no thisJoinPoint-construct (A2). Return types are hardcoded (T3) in the onexit-advice.

No implementation of TinyC^2 is freely available on the Internet.

3.2.3 *Arachne*

Arachne [7] improves on the μDiner framework, as annotating the base code is now obsolete. Dependence on a particular architecture is now localised in so-called "rewriting strategies" which guide insertion of hooks for a particular join point type on a specific architecture. For some reason, Arachne did not function on our machines, but the prototype weaver is still under heavy development[7].

The pointcut language has been reworked, inspired by Prolog (unification). There is also a new join point type: sequences of function call and/or (in)direct variable access join points. This is a natural means for advising protocol-like behaviour, as each element of the sequence can be advised individually (A2).

Unfortunately, advice is just a normal C procedure, so bound variables cannot be used in the advice body like Aspicere allows, nor is there an explicit proceed()-statement. Worse, a procedure's return type cannot be hidden behind a predicate, nor is there any context data available, apart from captured arguments which are passed to the advice (A2). This means that advice needs to be repeated for all possible return and argument types (T3).

On the other hand, Arachne's pointcut and join point model should be easily extensible and some issues could be ironed out eventually.

[7] `http://www.emn.fr/x-info/arachne/`

3.2.4 *TOSKANA*

TOSKANA (Toolkit for Operating System Kernel Aspects with Nice Applications) [8] is another aspect language for C with a dynamic weaver (appearing more or less at the same time as Arachne), but targeted solely at NetBSD's kernel mode. The weaving mechanism is similar to Arachne's (i.e. "code splicing"), but aspects are compiled into kernel modules. No prototype is available.

Due to the low-level nature of kernel code, TOSKANA's aspect language is very limited. Basically, advice is a C procedure with a special return type ("ASPECT") and it can call special macro's to proceed with the advised code, access stack state, ... In general, this is too low-level for reverse-engineering purposes (A2). In an initialising function, the advice is then instantiated as e.g. around advice and applied to a specific procedure execution. There are no name patterns or other means to advise many procedures at once.

3.3 VM-weaving

The dynamic weaving approaches of the previous section are restricted by the available information in the advised binaries. As such, richer join point models are hard to provide. What is more, the dynamic weaving approaches result in unoptimized woven applications, as the weaving process happens way after the last compiler optimization passes.

3.3.1 *TOSKANA-VM*

The folks of TOSKANA decided to take a look at virtual machines and came up with the TOSKANA-VM approach [9]: on top of an L4 microkernel, a bunch of LLVM (Low-Level Virtual Machine) [17] instances and a weaver are running. LLVM is a compiler framework with a universal IR (Intermediate Representation) and life-long analysis facilities. It can be extended with optional components to emulate a real virtual machine. So, the LLVM instances are virtual machine instances, in which applications (or operating systems) are running which have been compiled in the LLVM bytecode format. This way, the optimization problem is fixed and more advanced join point types are possible, as LLVM's IR stands on a higher level than mere binary code.

A downside of this approach, especially in older environments, are the relatively high infrastructural requirements like the microkernel and suitable operating system personalities (T5).

No information on the aspect language is available, except for the available join point types: call, execution, variable assignment and access. Likewise, no prototype is available on the Internet.

4. POINTS OF IMPROVEMENT

It is important to stress again the fact that our requirements considered worst case scenarios. E.g. if the case at hand is situated on a modern Linux environment on top of a standard Intel processor, T5 is not an issue. If one can fall back on an expert to refactor the build system, then T4 is not important either.

In the general case, work is needed to address T4. Many Java projects are using Ant or other XML-based build systems which can more easily be transformed. Even genuine makefiles are much more recent there than is the case for C systems. Although T4 is not a unique issue for C, it will show up more often.

Besides T4, T3 remains the principal problem when using AOP on crosscutting concerns affecting thoroughly scattered, unrelated join points. Reverse-engineering is one example, especially during the initial phases where one tries to narrow down the focus to the places of interest. Programming conventions [4, 2] are another one.

Only AspectC++ (section 3.1.2), once these features are supported in C, and Aspicere (section 3.1.3) provide both generic pointcuts (i.e. beyond mere pattern matching) and advice. The other frameworks have been designed with more specific crosscutting concerns in mind, where advice reuse is not the issue.

Finally, although all approaches use the terminology and lots of features originally introduced by AspectJ [15], typical C features (or problems) like pointers, dealing with macro's, slightly different dialects (T1), ... have not yet been addressed extensively. This is unlike the Java world, where one can experiment e.g. with the abc-workbench [3]. Built on a solid analysis framework, basic weaving functionality is already provided and people just need to focus on new features which eventually can loop back to the real AspectJ.

Although AOP in C goes back to 2001 [6], no such mature, extensible and general-purpose AOP-framework really took off. This is probably due to C's higher complexity and the lack of a natural higher-level IR like bytecode. As a practical consequence, people need to create a new aspect language like Aspicere from scratch, facing the same low-level groundwork others came across.

5. CONCLUSION

We reviewed the ten currently known AOP-frameworks for C in the context of seven requirements set out for a reverse-engineering case study. Compared to the Java scene, no framework really stands out. Most of the problems are related to the impact on the existing build system and the absence of generic advice.

6. ACKNOWLEDGEMENTS

We would like to thank Kris De Schutter, Andy Zaidman, Marc Fiuczynski (C4) and Pascal Dürr (WeaveC) for their input. Bram Adams is supported by a BOF grant from Ghent University.

7. REFERENCES

[1] Bram Adams, Kris De Schutter, and Andy Zaidman. AOP for Legacy Environments, a Case Study. In *2nd European Interactive Workshop on Aspects in Software*, 2005.

[2] Bram Adams and Tom Tourwé. Aspect Orientation for C: Express yourself. In *3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD*, 2005.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[4] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. An initial experiment in reverse engineering aspects. In *WCRE*, pages 306–307. IEEE Computer Society, 2004.

[5] Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD*, pages 50–59, 2003.

[6] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, 2001.

[7] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. In *AOSD*, pages 27–38. ACM Press, 2005.

[8] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05*, pages 51–62. ACM Press, 2005.

[9] Michael Engel and Bernd Freisleben. Using a LowLevel Virtual Machine to improve dynamic aspect support in operating system kernels. In *4th AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, 2005.

[10] Marc Fiuczynksi, Robert Grimm, Yvonne Coady, and David Walker. patch (1) Considered Harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[11] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD*, pages 36–45. ACM Press, 2004.

[12] Robert Grimm. Systems need languages need systems! In *2nd Workshop on Programming Languages and Operating Systems (ECOOP-PLOS'05), ECOOP*, 2005.

[13] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD*, pages 60–69. ACM Press, 2003.

[14] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[16] Ralf Lämmel and Kris De Schutter. What does Aspect Oriented Programming mean to Cobol? In *AOSD '05*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[17] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[18] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In Gabor Karsai and Eelco Visser, editors, *Proc. Generative Programming and Component Engineering: Third International Conference*, volume 3286, pages 55–74. Springer, October 2004.

[19] Klaus Ostermann, Mira Mezini, and Christophe Bockisch. Expressive pointcuts for increased modularity. In *ECOOP*, 2005.

[20] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *AOSD*, pages 110–119. ACM, 2003.

[21] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[22] Stijn Van Wonterghem. Aspect-oriëntatie bij procedurele programmeertalen, zoals C. Master's thesis, Ghent University, 2004.

[23] Andy Zaidman, Bram Adams, Kris De Schutter, Serge Demeyer, Ghislain Hoffman, and Bernard De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *CSMR*, 2006. Accepted for publication. To Appear.

[24] Charles Zhang and Hans-Arno Jacobsen. TinyC^2:towards building a dynamic weaving aspect language for c. In *FOAL 2003*, Boston, MA, USA.

Version 2.*.* and Counting! The Toll of Evolution on Aspect-Oriented Distribution

Jennifer Baldwin
University of Victoria
jbaldwin@cs.uvic.ca

John Zigman
The Australian National University
john.zigman@anu.edu.au

Yvonne Coady
University of Victoria
ycoady@cs.uvic.ca

ABSTRACT
The distributed Java Virtual Machine (dJVM) is a cluster aware implementation of a JVM, designed specifically for evaluating distributed runtime support algorithms [6]. A prototype implementation of the dJVM is available as a patch file applied to IBM's Jikes Research Virtual Machine (RVM) [2] version 2.2.0, released in January 2003. This patch touches roughly 55% of the original 1500 files. In previous work, we considered the impact of AOSD on this implementation of distribution [3]. Factors we looked at included internal structure, external interaction and reduced code size.

But an important part of this study remained untested: as the RVM continues to undergo rapid evolution, what toll is this evolution going to take on the aspects? The current release of the Jikes RVM, as of November 2005, is version 2.4.2. This paper provides a case study characterizing the major evolutionary trends over the period between these two versions and considers the impact this evolution has on the implementation of distribution as a patch versus aspects. This work is preliminary as the new, full implementation of the dJVM for 2.4.2 is not yet available. Therefore, the analysis in this paper is based on code inspection of the two versions of the Jikes RVM, as far as distribution modifications are concerned.

1. INTRODUCTION
In previous work [3][1], we argued that aspects could ease the evolution of distribution in a virtual machine, but what about the evolution of the virtual machine? What kind of toll would evolution of the base code take on the aspects?

In this study, our base code is IBM's Jikes Research Virtual Machine (RVM) [2] – a unique open source project in Java. Its code-base affords researchers the opportunity to experiment with a variety of design and implementation alternatives within an otherwise stable and consistently well maintained infrastructure.

The distributed Java Virtual Machine or *dJVM* [6][2] is a ground-breaking effort to add distribution to the original Jikes RVM. The impetus behind distributing the JVM is largely performance-based. Many Java-based server applications are multi-threaded, with relatively little interaction between threads. This would indicate that a virtual machine running across a cluster of nodes may provide dramatically increased concurrency for better performance. This new functionality is made available for download as a *patch file* – a file containing a difference listing produced by the *diff* utility – enabling users to seamlessly introduce distribution to their RVM, given a compatible version. Unfortunately, in its current form, it is difficult to reason about the ways in which the dJVM patch modifies the RVM. Though the design has been carefully developed, it is difficult to map it to the implementation when it is presented in terms of line numbers and modifications to over half of the 1500 files in the RVM.

The work presented in [3] is a preliminary report on efforts to structure the implementation of distribution supported by the dJVM using aspect-oriented programming (AOP) [4,5]. This preliminary experiment revealed the concrete ways in which aspects can improve the internal structure of some of the distribution code, clarify external interaction between the distribution code and the RVM, and reduce code size relative to the patch. These results bode well for easing evolution of distribution in the RVM with aspects. This work was motivated not only by the inherent value in improving the modularity of the system, but also by the reality that *patch* actually shackles innovation to outdated versions of the Jikes RVM. That is, it requires nontrivial effort to both understand and maintain the code in this form, even across minor RVM upgrades, due to the lack of semantic information and the fragile nature of textual substitution.

This paper extends that work by evaluating the evolvability of the aspects throughout significant changes to the Jikes RVM. Since patches are dependent upon line numbers, they are not able to leverage principled points of execution within the system. Therefore, logically the aspects are more evolvable. This work aims to investigate just how evolvable these aspects might be by evaluating whether or not they will still apply to the new system. This analysis is a necessary step before the next phase in this project, which includes porting the aspects to the new version of the RVM while at the same time incorporating additional changes in the newest version of the dJVM patch, when it is released.

The paper is organized as follows. Section 2 overviews the design of the dJVM. The results of the original study and the way in which these results pertain to the argument considered here, are outlined in Section 3. Section 4 discusses how evolution of the RVM has impacted distribution in general. In Section 5, we look at how the aspects of the earlier refactoring need to be modified and provide an analysis of the aspect-oriented implementation in the face of evolution. We discuss future work in Section 6 followed by conclusions in Section 7.

2. DJVM DESIGN
The dJVM provides a cluster aware JVM using a Single System Image (SSI) to the Java programmer, hiding the underlying architecture and its complexities from the programmer. The

[1] Research page is located at http://www.csc.uvic.ca/~jbaldwin/diva/

[2] The dJVM patches and documentation can be obtained from http://djvm.anu.edu.au/

dJVM is aware of the cluster however, and must try to maximize opportunities to make applications run efficiently.

One important design decision that is particularly challenging to comprehensively map to the implementation is that of static (global) versus instance (local) variables. Static variables may be local within their node or within the entire application. Local variables will always be held within their host node. When data is being kept locally, an empty interface called `DVM_LocalOnlyStatic` must be implemented by any class that contains static data that is always accessed locally [6].

3. ASPECTS OF DISTRIBUTION

The current strategy for introducing distribution to the Jikes RVM is by using *patch*. As a development tool, patch allows new functionality to be developed *in situ*, relative to the existing functionality of the system. But in terms of semantic leverage, the patch file itself is hard to understand.

The first step in creating our AspectJ implementation [3] was to go through the distribution patch and extract related code segments, constituting individual crosscutting concerns, then attempt to structure them as aspects. The following subsection describes the *LocalOnlyStatic* aspect as an example.

3.1 Example: LocalOnlyStatic Aspect

Figure 1 shows the AspectJ implementation of the modifications to class hierarchy. Looking briefly at the constructs in the aspect, the *declare parents* construct in AspectJ is used to modify the class hierarchy in the dJVM to introduce `DVM_LocalOnlyStatic`. Though this aspect is very straightforward, these changes account for a large portion of the patch, about 44%, of the changes in the system. In total, the original patch file consisted of 13,509 lines of code to accomplish what the aspect has done in 5 statements.

By inspecting the patch file, the design intent behind the set of classes that should implement, for example, the `DVM_LocalOnlyStatic` interface, is not obvious. The classes which define global structures that are for intra-node runtime systems, such as the scheduler, thread, garbage collection, internal type information, Java Table of Contents and compilers, are those that fall into this category. The problem is that the rule does not always apply and therefore, some trial and error is needed in order to define the subset of classes accurately. This is due to the distinction between identifying the behavior, versus *how* the behavior is implemented. This approach leads to a large number of trivial modifications which are particularly hard to map between the design and the implementation in their scattered form.

3.2 Aspects Ease Evolution of Distribution

Perhaps the most obvious difference between the patch and the aspects is the number of lines of code with each approach. The reduced number, the patched version at 36,987 lines and the aspect-oriented version at 11,482, has intuitive benefits in the realm of maintainability and understandability.

Frequently throughout the patch, visibility modifications (from *private* to *public*) were made. In order to distribute the system, functionality between clustered nodes needed to be made less strict in order to facilitate their communication. This relaxation of access rights suggests that encapsulation must be rethought in the presence of distribution, especially when changes made to the code are fixed. An aspect can control these modifications since changes to visibility can be seen either solely within the aspect or within the aspect's defining package. These reassignments in access are then restricted within an aspect and can easily be added/removed from the system.

```
package com.ibm.JikesRVM;

import com.ibm.JikesRVM.memoryManagers.vmInterface.*;
import com.ibm.JikesRVM.librarySupport.*;

public aspect LocalOnlyStatic {

         declare parents: VM_Scheduler || VM_Wait || com.ibm.JikesRVM.*Thread || VM_CollectorThread ||
                          VM_*Lock* || VM_Processor* || VM_Proxy ||  VM_Synchronization ||
                          (*Map* && !VM_JNIGCMapIterator) || com.ibm.JikesRVM.memoryManagers.JMTk.* ||
                          VM_Barriers || VM_Memory || Scan* || SynchronizationBarrier || Util ||
                          VM_Finalizer || VM_Handshake || VM_Interface ||
                          (*Info && !VM_PendingJSRInfo) || *Table* || VM_DynamicTypeCheck ||
                          (*Compile* && !VM_JNICompiledMethod) || (VM_Pragma* && !VM_PragmaException) ||
                          VM_DynamicLink ||VM_Entrypoints ||  VM_BaselineException* || VM_Magic* ||
                          VM_OutOfLineMachineCode || VM_RecompilationManager || VM_Runtime ||
                          VM_StackTrace || VM_Verifier ||
                          (VM_JNI* && !VM_JNICompiledMethod && !VM_JNIFunctions) ||
                          *Header || *Profile* || *Monitor* || VM_*Class* ||
                          VM || VM_Array || VM_Atom || VM_BasicBlock || VM_BootRecord ||
                          VM_BuildBB ||  VM_Callbacks || VM_CommandLineArgs || VM_Configuration ||
                          VM_EdgeCounts ||VM_FileSystem || VM_EventLogger || VM_Field || VM_Lister ||
                          VM_Math || VM_Member || VM_Method || VM_ObjectModel || VM_Primitive ||
                          VM_Properties || VM_Reflection || VM_Services || VM_Statics ||
                          VM_Time || VM_Triplet || VM_Type || VM_UTF8Convert || JikesRVMSocketImpl ||
                          FileSupport || ReflectionSupport && !*Constants && !DVM*
                          implements DVM_LocalOnlyStatic;

         declare parents: *Constants && !DVM* extends DVM_LocalOnlyStatic;

}
```

Figure 1. Aspect code for files implementing dJVM hierarchy related modifications for DVM_LocalOnlyStatic.

4. OVERVIEW OF EVOLUTION

A more recent release of the dJVM is not yet available, for a variety of reasons. These include the replacement of the roll-your-own bytecode manipulation tool with BCEL[3], and converting over to use the non-blocking *java.nio* libraries. Both have introduced unforeseen complications. The first is that the *classpath* and RVM classloading mechanism are somewhat moving targets, the second is that the GNU classpath implementation of the non-blocking libraries is not yet stable. Recent efforts in the project have gone into porting the bytecode analysis tools to those that BLOAT[4] (a bytecode level optimizer) provides.

The overall approach of the dJVM was to leverage the classloading mechanism as much as possible to effect both VM and application mutations. This will remain the approach for the next version of the dJVM; however, a higher-level specification of those mutations, perhaps in the form of aspects, is potentially quite worthwhile. In such a case the mutated classloading mechanism would include and/or be essentially a runtime weaver. However, we must also bear in mind performance is a long term goal.

There will be some changes to the dJVM design itself. One such change will be the inclusion of proxy objects (these will be identical to the original in terms of layout) which can act both as a cache for an object or as a method for redirecting calls. The purpose of this is to enable easy, and hopefully efficient, plug in of caching and replication algorithms. There is potential for aspects in this area in particular, though this implementation will be particularly sensitive and could have a serious impact upon performance and prevent many compiler optimizations.

Another area that is receiving attention is the removal of the trivial preprocessor directives from the Jikes RVM. In particular those directives that conditionally execute Java code, since a reasonable number of these can be replaced with standard Java condition statements. The replacement of these statements should not incur any execution overhead for code generated by the optimizing compiler since it can do constant folding and dead code elimination.

Improving the package structure and the interfaces between those packages combined with the elimination of preprocessor directives, isolating any non-standard Java into a minimal number of small fragments, should facilitate research using the Jikes RVM. Furthermore, such improvements would provide a better platform for considering what role aspects can play in the Jikes RVM and separately to the dJVM.

Currently the use of preprocessor directives for introducing interfaces such as LocalOnlyStatic is really circumstance dependent. For example if the restructuring of the Jikes RVM allowed a fairly succinct rule for replacing where the LocalOnlyStatic is used (i.e. a rule that someone could keep in their head so that they knew why and where it would be applied) then it would be replaced. However, if the rule becomes complex or obscure then an explicit annotation may be preferred. It is really a matter of cognitive effectiveness (or at least the developer's perception of what that is). An aspect for LocalOnlyStatic is more likely if the RVM structure is a little cleaner.

The high-level specification of mutations instead of lower level annotations is preferable from the perspective of dJVM development. Aspects would be more appealing if the Jikes RVM package structure and interfaces between those packages were improved, this would in turn help reduce the complexity of the rules identifying where Aspects would be applied.

5. THE TOLL OF EVOLUTION

In the aspect-oriented prototype implementation of distribution, there were four other aspects in addition to LocalOnlyStatic. These other aspects included those that made changes to the optimizing compiler, those specific to the PowerPC architecture and two others which will be factored out as development continues. These two include newly added variables and methods to existing classes (inter-type declarations), and the other which modifies existing methods (currently implemented without refactoring the original system). There will of course be a lot of extra functionality required due to new classes introduced to the system during evolution, but that artifact is out of the scope of this paper until the new dJVM patch is available. The overview of evolvable changes is shown in Figure 2.

5.1 LocalOnlyStatic Aspect

Perhaps one of the largest changes to the latest version of the Jikes RVM in terms of distribution is that the Java Memory Management Toolkit (JMTk) has been made independent of the Jikes RVM and is now known as the Memory Management Toolkit or MMTk. Much work has been done on factoring out the VM-specific code, which now resides outside of the Jikes RVM source tree, and it has been reorganized to have an Eclipse friendly directory/package structure. But it is not just the MMTk structure that has changed. In fact, some of the code originally located inside of the root package of Jikes has now been migrated into two new packages, `classLoader` and `jni`, and three packages (including their code) have been removed from the old version of the Jikes RVM. These included the `BytecodeToolset`, `ClassTransformer` and `librarySupport` packages.

The inclusion of MMTk means that some of the distribution modifications for the dJVM are lost. However, the aspect which is affected most by this is the previously mentioned LocalOnlyStatic aspect in which all of the members of the JMTk package were affected (40% of the changes in this aspect). It may seem logical to attempt to weave into every file within the MMTk package instead. However, this is likely to select too many files and therefore cause problems so the changes to memory management in the LocalOnlyStatic aspect will most likely have to be completely revised as a result of this evolutionary change.

Another interesting point about the LocalOnlyStatic aspect is that it takes advantage of wildcard expressions which capture different files in each version of the system. For example, `VM_NativeDaemonThread`, which does not exist in version 2.4.2, fell under `*Thread` in version 2.2.0. The AspectJ compiler [1] will not throw a compiler error telling us that this class does not exist since we didn't explicitly define it, therefore

[3] http://jakarta.apache.org/bcel/index.html

[4] http://www.cs.purdue.edu/s3/projects/bloat/

making the aspect more robust. This property of AOP makes it evolvable to new versions of a system. Each of these wildcard expressions were evaluated to make sure that they were not specifying any new files that perhaps were not supposed to be woven in the new version. In only 2 out of 15 cases, did the wildcard expression match any new files. In those 2 cases, it is highly likely that the new files should indeed be matched. In 8 out of 15 of those cases, the new version was missing files that were originally caught by the wildcard expressions in version 2.2.0. However, by not explicitly naming these files, our aspect will still compile and run.

This aspect modified 44% of the original 1500 files, many more than shown in Figure 2. This is due to the fact that many of these files were not in our configuration. However, the files we used and files from other configurations tend to have the same class names but reside in different directories within the source. Therefore, we expect the same number of changes to be unusable, no matter which configuration is used. This also applies to the following aspects, although not to the same extent. For more information on Jikes configuration options, see [2].

5.2 Optimizing Compiler

The files and methods which were modified in this aspect all still exist in the new version of the Jikes RVM. Additionally, the types of changes that were made within this aspect are highly evolvable. This is due to the fact that the changes include the addition of getter methods, before advice that is not dependent upon the code that follows it in the original method, and lastly, around advice which never proceeds to the original method's implementation. This type of around advice means that if the implementation of the actual method has changed in the new version of the Jikes RVM, we are less likely to need to change the aspect since we never use the original method's code.

5.3 PowerPC

The PowerPC aspect adds the functionality necessary to generate inline code for `VM_MagicNames.invokeStubMethod` when the `VM_MagicCompiler.generateInlineCode` method is executed. However, not only does the `invokeStubMethod` no longer exist within `VM_MagicNames`, but the `VM_MagicCompiler` class itself no longer exists and has been absorbed into `VM_Compiler`. As a result, this aspect will need to change completely. Additionally, with the release of Apple's Intel Processor machines, it is likely that PowerPC support will no longer be supported. If this were the case, it shows how useful it can be to provide architecture specific aspects which can easily be evolved or even unplugged in response to hardware evolution.

5.4 Inter-type Declarations

In regards to inter-type declarations, where we are adding entirely new methods and variables to an existing class, we really only need to see whether or not the target classes still exist within the system. As mentioned previously, some of the original RVM files were migrated into a `classLoader` package. 9 of the files modified in this aspect are now within this new package. However, if we import all needed packages at the top of the aspect, it will find the files and weave into them no matter which package they are in. This means the only changes needed in this aspect will be different imports statements. However, 11 out of 36 of the files modified in this aspect no longer exist in the new system so the changes made to them will likely need to be migrated to new classes.

5.5 Advice on Methods

Lastly, changes to methods were evaluated by inspecting the join points and seeing if, (a) the classes and methods themselves still

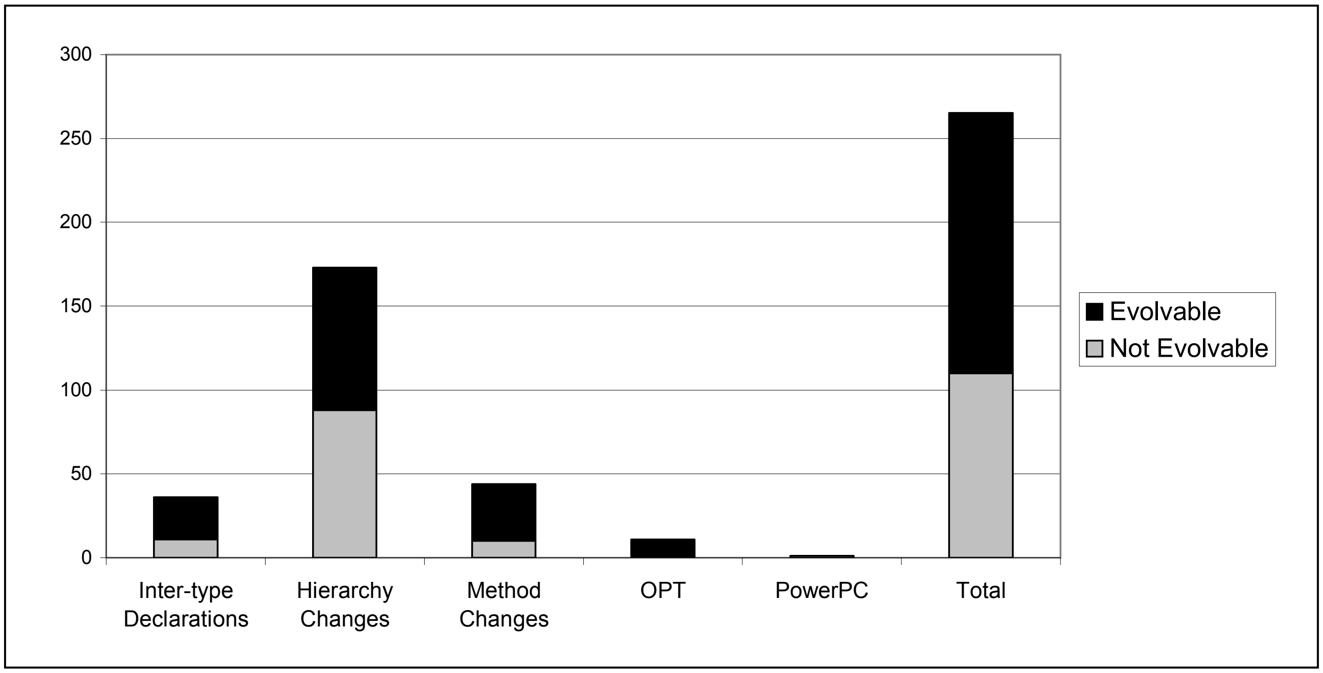


Figure 2. Number of classes which have modifications that are applicable to Jikes 2.4.2

existed, and (b) judging by the implementation in the methods, whether or not the advice would still be applicable.

In regards to (a), 10 out of 44 changes are no longer usable because their methods or classes are no longer present. In evaluating (b), since some of the advice was before or after advice, then as long as the method signature was the same, we would still be able to bind to it. Problems arose, however, when we had around advice. Due to difficulty in implementing changes to the middle of a method, and because no refactoring was done to incorporate aspects, the whole method needed to be copied to the around advice with no proceed functionality. In this case, if the method implementation had changed at all, our aspect would be out of date. Because of this, 7 out of 44 changes may not work but with refactoring, made especially easy if we had automated refactoring tools, our aspects would have a better chance of survival.

The final results of this indicate that for the most part, the aspect would be applicable to the new system with approximately 35% of the changes being those made to non-existent methods or those in around advice that are likely to require modification.

6. FUTURE WORK

The work present in this paper is a quantifiable, logical argument based upon code inspection. In order to really evaluate the arguments put forth in this paper, it is important to apply the existing aspects to the new version of the Jikes RVM. Clearly, these aspects will have to be modified and extended in order to implement a functional distributed virtual machine. In order to do this, when the new patch is released, it will be refactored into aspects in much the same way as it was for the original dJVM system. The aim of reimplementing distribution for another version of the Jikes RVM is to answer the question: would it be less work to modify the aspects rather than reimplement distribution for the new version of the Jikes RVM? If the effort involved in effecting a large change such as distribution into an evolving system can be significantly eased, then changes such as these are less likely to be left behind as the system evolves.

In this particular case study, evolvability may not be the most important factor in deciding whether to use patches or aspects. Another factor to consider is the effect that aspects have on performance. Previous work has shown that aspects may decrease performance [7]. Since the impetus behind distributing the RVM was to boost performance, this may not be acceptable. Therefore, performance testing will also be an important factor to consider in the future.

7. CONCLUSIONS

The dJVM [6], which effects distribution by means of a patch, is based on version 2.2.0 of the Jikes RVM [2]. However, only 2 years later, we are now on version 2.4.2 of the Jikes RVM and the developers of the dJVM are trying to catch up. When this task is complete, what version of the Jikes RVM will be the newest? It is a constant struggle for developers to keep large system changes up to date if the system they are based upon is continuously changing. Aspects have been touted as being evolvable due to the fact that they are based on principled points of execution within the system, whereas the patch is rigidly based on line numbers. Therefore, as part of an entire study of using AspectJ to implement distribution in the Jikes RVM [3], it is important to look at how evolvable those aspects really are in practice.

This preliminary study has shown that these aspects are more evolvable than the patch but are not perfect. As the system changes, package structure changes, functionality changes or even entire files are removed. It is estimated that roughly 40% of our aspects can no longer be applied for the previously mentioned reasons. Using wildcard expressions such as *Thread in the LocalOnlyStatic aspect can help by continuing to capture design intent rather than statically applying changes to named files. In order to really leverage these expressions, we need the Jikes RVM to have a clean structure, basing packages on design or at the least, having an understandable naming convention on which modifications to design can be based.

A future study will implement distribution on a newer version of the Jikes RVM, using the aspects from version 2.2.0 and the new dJVM patch file, when it is released. We can then measure whether the task of recreating distribution from older aspects is easier than reimplementing distribution across all of the new source files. We hope to work closely with tool developers to establish new means of linking aspects to evolution, both in terms of evolving aspects independently of the base code, and in terms of evolving the base code independently of the aspects.

8. REFERENCES

[1] AspectJ compiler 1.2, May 2004. http://eclipse.org/aspectj/.

[2] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, Vol 44, No 2, 2005.

[3] J.Baldwin and Y.Coady. Are Patches Cutting it? Structuring Distribution within a JVM using Aspects. In *Proceedings of the IBM Center for Advanced Studies Conference (CASCON)*, Markham, Ontario, Canada, October 2005. In *Proceedings of CASCON*, 2005.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP),* pages 327 – 353, Jyväskylä, Finland, 2001.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP),* pages 220 – 242, Jyväskylä, Finland, 1997.

[6] J. N. Zigman and R. Sankaranarayana. Designing a Distributed JVM on a cluster. In *Proceedings of the 17th European Simulation Multiconference*, Nottingham, United Kingdom, 2003.

[7] P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, 2005. Optimising aspectJ. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation,* Chicago, IL, USA, June 2005.

An Evaluation of Coupling Measures for AspectJ

Marc Bartsch
School of Systems Engineering
The University of Reading, Reading
RG6 6AY, UK
+44(0)118 378 8617
m.bartsch@reading.ac.uk

Rachel Harrison
School of Systems Engineering
The University of Reading, Reading
RG6 6AY, UK
+44(0)118 378 8617
rachel.harrison@reading.ac.uk

ABSTRACT

The maintenance of aspect-oriented software requires measures that are theoretically valid. Management or project decisions made using metrics that have not been validated may be detrimental or unhelpful. Recently, measures have been suggested that focus on aspect-oriented concepts, such as the crosscutting behaviour of aspects. Before these new measures can be put to use they should be evaluated to determine how far they indeed measure what they purport to quantify. This paper focuses on the theoretical evaluation of five aspect-oriented coupling measures with the aim of constructively increasing the quality of software evolution.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Measurement – *product metrics.*

General Terms

Measurement.

Keywords

Measurement, aspect-oriented, coupling.

1. INTRODUCTION

Aspect-orientation is an emerging paradigm that is based on the separation of concerns principle. It offers the idea of a new modular unit that encapsulates crosscutting concerns which would otherwise be scattered across multiple modules. Aspect-oriented languages introduce new forms of coupling which are unknown to object-oriented languages. The execution of base code might trigger the execution of aspect code leading to coupling relationships between classes and aspects which are not transparent. In addition to that, intertype declarations can change class implementations by adding attributes or methods.

Measuring these new kinds of coupling relationships is an issue which has been addressed lately with the definition of coupling measures specifically designed to support aspect-oriented concepts. A maintenance process for aspect-oriented software that relies on the results of these measures must have the confidence that the measures involved do indeed measure what they purport to quantify. Also, the comparison of measurement results is an error prone task if measures can be interpreted in different ways.

The position of the authors is that all measures including coupling measures need to be validated to gain confidence in the results taken from measurement. However, others have pointed out that measures that cannot be validated may still be useful [6]. Since research into measurement of aspect-oriented systems is at an early stage, it is particularly important to validate aspect-oriented measures thoroughly. Also, the authors would like to stress the fact that the validation of aspect-oriented coupling measures as presented in this paper depends on at least two frameworks. First, the validation criteria need to be agreed upon. A measure that validates successfully in the context of one framework might not validate in another. Second, aspect-oriented coupling measures depend on a specific idea of coupling. Mechanisms that constitute coupling in one aspect-oriented language might not exist in another. Hence, a specific aspect-oriented language implementation has to be considered, when aspect-oriented coupling measures are validated. In this paper we focus on measures for AspectJ [1].

This paper presents the evaluation of five selected aspect-oriented coupling measures suggested by Ceccato and Tonella [4]. They define five coupling measures, CAE, CIM, CFA, CMC and CDA, in order to investigate the trade-off between the advantages obtained from a separation of concerns and the disadvantages caused by coupling introduced by aspects. We selected the five measures for two reasons: first, two of the five measures (CMC, CFA) are derived from a well-known object-oriented measure (CBO). Second, measures like CAE, CIM and CDA focus on aspect-oriented core concepts such as static and dynamic crosscutting. We believe that the five measures are good representatives for measuring different kinds of coupling in AspectJ and that they deserve further investigation to make sure they are theoretically valid.

The paper is structured as follows: section 2 presents related work, section 3 introduces two evaluation frameworks used in this paper; section 4 introduces and evaluates the coupling measures. Finally, section 5 offers conclusion and points to further research.

2. RELATED WORK

Aspect-oriented measures derived from Chidamber and Kemerer's suite of object-oriented measures [5] have been suggested by Ceccato and Tonella [4] and by Sant'Anna et al. [9].

Zhao defines coupling [10] and cohesion [11] measures for aspect-oriented systems.

The issue of validating measures has been approached from different viewpoints. Kitchenham et al. define a set of criteria that all measures must obey to be considered a valid measure from a measurement theory point of view [8]. Other validation criteria have been suggested by Briand et al. [3]. They define mathematical criteria that all coupling measures must conform to. As far as construct validity is concerned, these criteria are only necessary but not sufficient [3]. In contrast to the criteria put forward by Kitchenham et al. they do not consider the relationship between the attribute being measured and the mathematical formula that is used to calculate the measure. Instead, they focus on properties of the mathematical formula only. Such criteria can verify that a certain algorithm does indeed measures coupling. They cannot, however, detect whether all coupling relationships that are necessary to measure a certain coupling attribute will be counted. We have chosen the validation framework by Kitchenham et al. in this paper since we are more interested in the question of whether an algorithm indeed represents a specific kind of coupling attribute than in the question of whether an algorithm represents coupling at all.

In [7], Kaner and Bond define a list of 10 questions that they consider relevant for any measure from a practical viewpoint and also raise the issue of construct validity.

3. EVALUATION FRAMEWORKS

The evaluation process for the five coupling measures will be guided by two frameworks. First, a coupling framework for AspectJ will be used that offers a common terminology to express the different coupling measures in a uniform way [2]. The coupling framework is a means to investigate ambiguities associated with the algorithm that implements each of the different measures.

Second, the validation framework by Kitchenham et al. is used for the purpose of a theoretical validation of each measure [8]. The validation framework consists of four criteria that all valid measures must obey.

3.1 Coupling Framework

Bartsch and Harrison [2] suggest a coupling framework for AspectJ which is an extension of a coupling framework for object-oriented systems put forward by Briand et al. [3]. Our coupling framework [2] focuses on coupling mechanisms that do not exist in object-oriented languages or that exists between members which are unknown in object-oriented languages.

The criteria of the framework are:

1. *Type of Connection*. The type of a particular coupling connection is determined by the mechanism that is used to establish the coupling connection. The use of a class identifier as a return type of a piece of advice, for example, is a mechanism that leads to coupling between the aspect that implements the advice and the class whose identifier has been used.
2. *Locus of Impact*. If an aspect is used in a coupling connection, a distinction is made between import and export coupling which defines a counting rule for a coupling connection. If aspects invoke methods of other classes, then import coupling counts the number of classes whose methods would be called by a given aspect. Export coupling counts the number of aspects that make calls to a method of a given class.
3. *Granularity*. Granularity refers to the level of detail at which coupling information is collected. It indicates the components that are counted and how to count multiple occurrences of a connection.
4. *Stability of Server*. Stability of server expresses whether components at the receiving end of a coupling connection are subject to modifications and might influence coupled classes if modifications are applied. This criterion is independent of the distinction between object-oriented or aspect-oriented languages and is beyond the scope of this paper. For the remainder of this paper, only stable classes and aspects will be considered.
5. *Direct or Indirect Connections*. Direct or indirect connections refer to whether to count direct coupling connections only or also indirect connections. For the remainder of this paper, only direct connections will be considered.
6. *Inheritance*. The following questions should be answered with regard to inheritance: First, does the use of members, such as the use of methods, attributes or pointcuts, of an ancestor class or aspect constitute coupling or not? Second, does a measure consider both polymorphically and statically invoked methods and, third, do inherited members belong to the inheriting aspect or not?
7. *Static Crosscutting/Intertype Declarations*. Similar to the question of whether an inherited method or an attribute belongs to a class or an aspect, we have to determine whether *introduced* methods or attributes belong to a class or not. There are three options. First, an intertype declaration only belongs to the aspect that defines it. Second, an intertype declaration only belongs to the class it is defines for or, third, an intertype declaration belongs to both the class and the aspect.
8. *Dynamic Crosscutting*. Join points can reference an executing object and a target object. When a coupling measure is being defined a choice has to be made whether to count only the executing object, the target object, or both.
9. *Instantiation*. Instantiation refers to the question whether to count aspects at a per-instance level or not. This criterion refers to runtime measures and will not be considered further.

Ambiguities can arise from the fact that certain criteria are not addressed in a measure where a choice is possible. For example, criterion 7 refers to a decision of how to assign intertype declarations. A coupling measure that does not address this issue might be interpreted in different ways. We will apply the coupling framework to investigate the well-definedness of each measure, i.e. to find and to resolve ambiguities of each measure.

3.2 Validation Framework

Kitchenham et al. propose a set of four criteria that all valid measures must obey [8]:

1. For an attribute to be measurable, it must allow different entities to be distinguished from each other.
2. A valid measure must obey the Representation Condition [6], i.e. it must preserve our intuitive notion about the attribute and the way in which it distinguishes different entities.
3. Each unit of an attribute contributing to a valid measure is equivalent.
4. Different entities can have the same attribute value (within the limits of measurement).

In this paper, we will focus on criterion 2, the Representation Condition. Since all coupling measures which we will discuss are based on counting coupling connections only, criteria 1, 3 and 4 can be validated intuitively and will not be considered any further.

Notions about attributes are subjective and can vary. Not everyone will agree upon what constitutes coupling in aspect-oriented systems or in a particular language like AspectJ. In order to validate a measure against the Representation Condition, first, the notion of the attribute being measured has to be defined. Second, an algorithm needs to be specified that defines how the values of the measure will be computed. The validation process consists of verifying that the values support the agreed notions about the measure. The Representation Condition is also highlighted during consideration of Construct Validity [7]. Construct Validity generalises from a measure or the algorithm of a measure to the concept of it, and asks whether the algorithm really captures the notion of a certain measure. The process of validation is then an investigation into how far the algorithm agrees with the attribute being defined for a certain measure.

4. EVALUATION

As a result of the discussion in section 3, the evaluation process will include the following steps for each measure: first, the attribute being measured by each measure will be presented. Second, the original algorithm will be stated and, third, the well-definedness of each algorithm will be investigated. Fourth, each measure will be validated by an investigation of the Representation Condition, i.e. an investigation into the notion and the algorithm involved. Usually this involves the question of whether a certain measure includes all the necessary coupling mechanisms that are associated with a certain notion of coupling. If necessary, changes to the definition of the measure will be suggested.

4.1 CAE (Coupling on Advice Execution)

Attribute. Aspects can cause control flow shifts so that advice is executed in the course of a program's execution. CAE [4] quantifies the amount of coupling caused by these shifts for a given class or aspect.

Algorithm. CAE counts the number of aspects containing advices possibly triggered by the execution of methods, advices or method intertype declarations, attribute and attribute intertype declarations in a given class or aspect.

Well-Definedness. The measure uses selected join point coupling mechanisms, i.e. all mechanisms that refer to executions: *method*, *constructor* or *advice execution* and *attribute* and *attribute intertype declaration join point coupling*. The granularity is *aspect* and every aspect will be counted only once. The locus of impact is *export coupling*. We interpret the phrase *possibly triggered* as a hint to the dynamic nature of pointcuts. Pointcut designators such as *cflow* cannot be determined at design time and have to be approximated. As far as intertype declarations are concerned, we assign them to the aspect they are defined in. Advice that is executed due to the fact that a method intertype declaration is executed will be considered coupled to the aspect that defines the intertype declaration. With these interpretations, it is possible to classify CAE as well-defined.

Validation. CAE counts only certain selected join point coupling mechanisms that can lead to the execution of advice. AspectJ supports more types of join points that can also cause the execution of advice, such as object initialization join points, exception handler join points, call join points and advice execution join points. A valid measure of coupling on advice execution needs to counts all of these join point coupling mechanisms.

Suggested Changes. In order to achieve theoretical validity we suggest including all types of join point coupling mechanism, i.e. all join points that can cause advice to be executed. The following table summarizes all the join points that may need to be counted by CAE:

Table 1. Join Points

#	Mechanism
1	method execution join point
2	method call join point
3	constructor call join point
4	constructor execution join point
5	object initialization join point
6	attribute reference join point
7	attribute assignment join point
8	handler execution join point
9	advice execution join point

If the suggested changes are applied, we can classify CAE as a theoretically valid measure.

4.2 CIM (Coupling on Intercepted Modules)

Attribute. CIM [4] quantifies the explicit knowledge that an aspect has in its pointcuts of another class or interface that it crosscuts. It indicates the direct knowledge an aspect has of the rest of the system. High values indicate high coupling, due to high crosscutting.

Algorithm. CIM counts the number of classes or interfaces explicitly named in the pointcuts of a given aspect.

Well-Definedness. This algorithm can be mapped unambiguously to the criteria of the coupling framework. The type of connection is *type pattern coupling*, the granularity is *class* or *interface* and every class or interface will be counted once. The locus of impact is *import coupling*. An aspect that selects a join point will be coupled with the *executing object* exposed by that join point. With this addition, we can classify CIM as well-defined.

Validation. CIM cannot be validated successfully. This measure makes the assumption that an explicitly mentioned class or interface in a pointcut always leads to crosscutting or to crosscutting with the mentioned class or interface. This is not always the case. Consider the following examples:

```
pointcut execMethods() :   execution( void *.add() )
                      && ! execution( void Manager.add() );
pointcut callMethods() :   call( void Manager.add() );
```

Figure 1. Type pattern based coupling.

Although the *Manager* class is explicitly mentioned in both pointcuts, it is excluded from the set of join points in the first example and does not lead to crosscutting with the *Manager* class. In the second example, *Manager* is also explicitly mentioned, but it also does not lead to crosscutting with the Manager class. A minor problem is that the mere definition of a pointcut is not enough to crosscut base code. It also needs to be referred to in at least one piece of advice. As a result, the given algorithm does not measure the attribute defined.

Suggested Changes. The problems with this measure can be resolved in two ways. A first approach would be alter the attribute (and the name) so that CIM only quantifies the explicit knowledge an aspect has of the rest of the system, regardless whether this knowledge leads to crosscutting. An alternative approach would be to alter the algorithm to count only explicitly mentioned classes or interfaces that in fact contribute to crosscutting and that are referred to in at least one piece of advice. With one of these changes, CIM can be considered a theoretically valid measure.

4.3 CMC (Coupling on Method Call)

Attribute. CMC [4] is based on Chidamber and Kemerer's CBO measure. CMC focuses on method calls only and quantifies the amount of coupling due to the call of methods or method intertype declarations of other classes or aspects.

Algorithm. CMC counts the number of classes or aspects that declare methods which are possibly called by a given class or aspect, including those methods that have been introduced by intertype declarations.

Well-Definedness. The type of connection is *method execution*, the granularity is *class* or *aspect* and every class or aspect will be counted once. The locus of impact of CMC is *import coupling*. We associate intertype declarations with the aspect that defines them and we associate inherited methods with the class that defines them. We interpret the phrase *possibly called* as potential coupling due to polymorphism. If a polymorphic method is called, we associate the calling class with all possible classes that could be called due to polymorphism. With these additions, we can consider CMC to be well-defined.

Validation. The theoretical validity of this measure depends to a great extent on how much the attribute that CMC measures is based on CBO. The attribute that CBO measures is *coupling between objects*: two objects are coupled if and only if one of them acts upon the other. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a thing traverses in time [5]. In terms of software, *acting upon* is defined as calling an object's method or accessing an object's attribute. Obviously, CMC does not support the entire attribute that CBO measures, since it only considers method calls. Another deviation from the CBO measure is the fact that CMC only counts *import coupling* whereas the *acting upon* relationship implies *import* and *export coupling*. We draw the following conclusion: if CMC is modeled on CBO it should count *import* and *export coupling* in order to be a theoretically valid measure. If it is not modeled on CBO, the current definition of CMC can be considered theoretically valid.

Suggested Changes. If CMC is based on CBO, we suggest changing the algorithm to include *import* and *export coupling*.

4.4 CFA (Coupling on Field Access)

Attribute. CFA [4] is designed similarly to CMC. Where CMC measures coupling due to method calls, CFA measures coupling due to attribute accesses.

Algorithm. CFA counts the number of classes, interfaces or aspects declaring attributes that are accessed by a given class or aspect.

Well-Definedness. The type of connection is *attribute access*, the granularity is *class*, *interface* or *aspect* and every class, interface or aspect will be counted only once. Just like CMC, we associate methods and attributes with the class or aspect that defines them. With these additions we consider CFA to be well-defined.

Validation. The theoretical validity of this measure follows the same argument as for CMC. If CFA is modeled on CBO, it should consider import and export coupling to be theoretically valid. If it is not modeled on CBO, it can be considered theoretically valid without any changes. This consideration is important if values of CBO are compared with the sum of the values of CMC and CFA. Since they do not measure the same attribute, a comparison might be problematic.

Suggested Changes. If CFA is based on CBO, we suggest changing the algorithm to include import and export coupling.

4.5 CDA (Crosscutting Degree of an Aspect)

Attribute. CDA [4] captures the overall crosscutting impact that an aspect has on the rest of the system through pointcuts and intertype declarations. The difference between CDA and CIM gives the number of modules that are affected by an aspect without being referenced explicitly by the aspect.

Algorithm. CDA counts the number of classes or aspects affected by the pointcuts and by the intertype declarations of a given aspect.

Well-Definedness. The locus of impact is *import coupling*, the granularity is *class* or *aspect* and every class or aspect will be counted only once. We interpret the crosscutting degree of an aspect as being dynamic and static crosscutting. A *class a* or an *aspect b* is affected by an *aspect c*, if the *aspect c* defines join points whose executing object can be associated with either *class a* or *aspect b*. The set of join points can only be approximated at design time. A *class a* is also affected by an *aspect b* if the aspect defines an intertype declaration for *class a*. With these additions, we consider CDA well-defined.

Validation. CDA can be considered a theoretically valid measure since it considers both dynamic and static crosscutting.

As far as the difference between CDA and CIM is concerned, it does not indicate the number of modules that are affected by an aspect without being referenced explicitly. Since CDA considers intertype declarations, which CIM neglects, it counts modules which are affected by an aspect but which are referenced explicitly. The degree of generality intended by CDA can only be maintained if it considers modules affected by dynamic and not by static crosscutting.

4.6 Summary

The following table gives an overview of the measures and validation status. For CMC and CFA two validation states are given. As discussed, these refer to the extent to which they are based on the underlying CBO measure. We consider CDA to be a valid measure without any changes.

Table 2. Overview of all measures

Measure	Validation status before changes	Validation status after changes
CAE	No	Yes
CIM	No	Yes
CMC	Yes/No	Yes
CFA	Yes/No	Yes
CDA	Yes	-

5. CONCLUSION

In this paper, we investigated the theoretical validity of five coupling measures and identified areas of improvement for four of the five measures. In two cases the improvements depend on how the attribute being measured is defined. We also suggested changes where necessary and conclude that with these changes all five measures can be considered theoretically valid. In particular, we conclude from our research that measures like CIM can now be applied with more confidence in a software evolution process.

The coupling framework used in this paper describes decisions that need to be considered when a measure is being defined. It served as a tool to explore the expressiveness of all five measures with regard to the criteria defined in the framework. We investigated the well-definedness of the measures and gave our interpretation of each in terms of this framework. For example, we followed the principle to assign methods and attributes to those classes and aspects that define them. These decisions are important when inheritance or intertype declarations are involved, because they greatly influence a measure: CMC counts method calls, but if a method being called is an intertype declaration then there is a coupling relationship towards the aspect that defines the method and not towards the class the method was defined for. Our interpretation of each measure highlights the degrees of freedom that designers of coupling measures need to consider.

A theoretical evaluation of measures can point to critical areas. We have shown that CMC and CFA do not cover the same attribute that CBO measures and that a comparison of CBO values with the total of CMC plus CFA is likely to be problematic. Only if CMC and CFA are defined in a similar way to CBO can comparability be guaranteed.

Future work will include the formal definition of aspect-oriented coupling measures and an investigation into the validity of other sets of coupling measures. Also, we would like to work towards an empirical validation of the measures discussed in this paper.

6. ACKNOWLEDGMENT

We would like to thank all anonymous reviewers for their valuable comments.

7. REFERENCES

[1] AspectJ, 2005: http://www.eclipse.org/aspectj

[2] Bartsch, M. and Harrison, R., ‘A Coupling Framework for AspectJ’, Extended Abstract, *Proceedings of EASE 2006*, Keele, UK. http://www.personal.rdg.ac.uk/~sir04mb2/Publications/bartsch-harrison-couplingFramework.pdf.

[3] Briand, L.C., Daly, J.W. and Wüst, J.K. (1999) A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1), 91-120.

[4] Ceccato, M. and Tonella, P., ‘Measuring the Effects of Software Aspectization’, *CD-Rom Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE 2004).* Nov. 2004. Delft, The Netherlands.

[5] Chidamber, R. and Kemerer, C.F., ‘A Metrics Suite for Object-Oriented Design, *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] Fenton, N.E. and Pfleeger, S.L., ‘*Software Metrics: A Rigorous and Practical Approach (2nd Edition)*’, International Thomson Computer Press, 1996.

[7] Kaner, C. and Bond, P., ‘Software Engineering Metrics: What Do They Measure and How Do We Know?’, *10th International Software Metrics Symposium (Metrics 2004)*, Chicago, IL, September 14-16, 2004.

[8] Kitchenham, B.A., Fenton, N. and Pfleeger, S.L., ‘Towards a Framework for Software Measurement Validation’, *IEEE Transactions on Software Engineering*, 1995, Vol.21, No.12, pp. 929-944.

[9] Sant’Anna, C., Garcia, A., Chavez, Ch., Lucena, C. and von Staa, A., ‘On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework’, *Proceedings of the Brazilian Symposium on Software Engineering (SBES'03)*, Manaus, Brazil, Oct 2003, pp.19-34.

[10] Zhao, J., ‘Measuring Coupling in Aspect-Oriented Systems’, *10th International Software Metrics Symposium (METRICS'2004)*, (Late Breaking Paper), Chicago, USA, Sept. 14-16, 2004.

[11] Zhao, J. and Xu, B., ‘Measuring aspect cohesion’. *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, LNCS 2984, Barcelona, Spain, March 2004. Springer Verlag, pp.54-68.

Aspect-Oriented Refactoring: Classification and Challenges

Jan Hannemann
University of Tokyo
jan@graco.c.u-tokyo.ac.jp

ABSTRACT

This paper provides an overview of the three different kinds of AOP refactorings: aspect-aware OO refactorings, refactorings for AOP constructs, and modularizations of crosscutting concerns. We discuss recent developments for each of them and highlight their commonalities with respect to associated challenges and ties to related research, such as program analysis and aspect mining.

1. INTRODUCTION

Useful software systems are constantly evolving and changing, and often those changes require that the software be re-modularized, so that the system becomes easier to understand, extend, or maintain [8]. The technique for disciplined program transformations that change a system's structure while preserving its behavior is called refactoring. Refactorings are parameterized transformations of a system's source code intended to improve a system's structure with regards to informally expressed goals, such as maintainability, changeability, or readability [5, 8, 21].

While most refactoring research focuses on object-oriented (OO) system transformations, refactorings play a similar role in an aspect-oriented programming (AOP) context. In addition, refactorings are instrumental for the migration of legacy OO systems to use AOP. To distinguish AOP refactorings from traditional OO refactorings, and to differentiate between individual approaches to aspect-oriented refactoring, we will use the simple AspectJ example shown in Figure 1. It shows part of a `Book` class for a library system. The code presented focuses on signing out books. In the method `signOut(Customer)`, race conditions are prevented by acquiring a lock before the actual changes are executed, which happens in method `helper(Customer)`. After that, the lock is released. If the book is not available (signed out already, for example), an exception is raised. Further, an aspect logs all successful transactions.

We use the term traditional refactorings to refer to parameterized, behavior-preserving transformations of object-oriented systems (although refactorings are not limited to OO systems, the majority of research targets that paradigm). Such refactorings usually focus on single program elements or small sets of non-scattered program elements (plus their associated references), and they can often be automated. Despite the limited number of elements targeted, such refactorings may cause program-wide changes. For example, a *Rename Method* refactoring [5] requires all call sites to the targeted method to be updated. Renaming the method `signOut(Customer)` in the example system represents a traditional refactoring.

2. AOP REFACTORINGS

Aspect-oriented refactorings differ from traditional refactorings in that they involve/consider AOP constructs. They can be divided

```
public class Book {
 . . .
 public void signOut (Customer c)
  throws UnavailableException {
   log("attempting to sign out "+this);
   acquireLock(this);
   try {
    helper(c); }
   finally {
    releaseLock(this); }}

 private void helper(Customer c)
  throws UnavailableException {
   if(! isAvailable) {
    throw new UnavailableException(..); }
   else {
    owner = c;
    isAvailable = false;
    signOutCounter++; }}}


public aspect LibraryLogger {
 . . .
 pointcut signOutBook(Book b, Customer c):
  execution(void Book.helper(Customer)) &&
    target(b) && args(c);

 after(Book b, Customer c) returning:
   signOutBook(b, c) {
    log(b + " signed out " + getDate()); }}
```

Figure 1. A simple library example

into three distinct groups as outlined below. Their properties and differences are summarized in Table 1.

1. Aspect-aware OO refactorings, which are traditional object-oriented refactorings that are extended to ensure they do not break AO programming constructs.
2. Refactorings for AOP constructs, which are refactorings explicitly involving AOP program elements, either as the targeted elements or in the resulting code
3. Modularizations of Crosscutting Concerns, which are composite refactorings aiming to transform non-modularized CCCs into aspects.

In the following, we describe each kind of aspect-oriented refactoring in detail.

2.1 Aspect-Aware OO Refactorings

Traditional refactorings focus on object-oriented code. When applied as-is to an AOP system, references in AOP constructs are not taken into account, potentially introducing errors.

For instance, applying the *Rename Method* refactoring to the (poorly named) `helper(Customer)` method requires updates to references of that construct not only in OO code elements, but also in AOP constructs, like the `signOutBook(..)` pointcut. Similarly, inlining the `helper(Customer)` method would also

Table 1. Overview of AOP refactoring approaches.

Approach / Technique	Target	Focus / Motivation	Examples
Aspect-Aware OO Refactorings	OO constructs	Ensure OO refactorings update references in AOP constructs properly	Rename/Inline Method etc.
Refactorings for AOP Constructs	OO and AO constructs	Provide new refactorings involving AOP constructs	Replace Member with Intertype Declaration, Pull up Pointcut, etc.
CCC Modularizations	CCC implementations	Provide refactorings to replace non-modular CCC implementations with equivalent aspects	[Replacement of any non-modularized CCC implementation with an aspect]

require aspect-oriented version of an OO refactoring. This case is more complex, as the join points identified by the pointcut cease to exist [9].

Research in the area of aspect-aware OO refactorings focuses on extending traditional refactorings with appropriate steps to properly update references in AOP constructs. Making OO refactorings aspect-aware is the topic of several research projects, for example [9, 14, 25].

2.2 Refactorings for/to AOP Constructs

While the aforementioned refactorings target OO constructs, this class or refactorings explicitly involves AOP constructs, either directly (e.g., inlining a pointcut and advice body), or in the resulting code (e.g., replacing an object method with an appropriate inter-type declaration). Many of these parallel existing OO refactorings: with respect to the kind of refactorings that can be applied to them, pointcuts can be compared to methods and aspects to classes. It is straightforward to envision the meaning of OO refactorings such as *Add Parameter*, *Pull Up Method*, or *Push Down Method* [5] when applied to a pointcut declaration. Similarly, equivalents of *Collapse Hierarchy* or *Extract Super/Subclass* may be conceived for aspects, as shown by Monteiro et al. [20], for example.

In addition, AOP programming constructs allow for a set of new refactorings without OO equivalents, like the merging or splitting of advice and/or pointcuts. For example, imagine an additional pair of pointcut and advice in the library system for logging the returning of books. The pointcuts, the advice, or both could be merged with the ones for logging signing books out. Several researchers have proposed new refactorings for AOP constructs, for example [14, 20].

Finally, there are low-level object-to-aspect program transformations that 'move' small OO code fragments to aspects by replacing them with their aspect-oriented equivalents. These

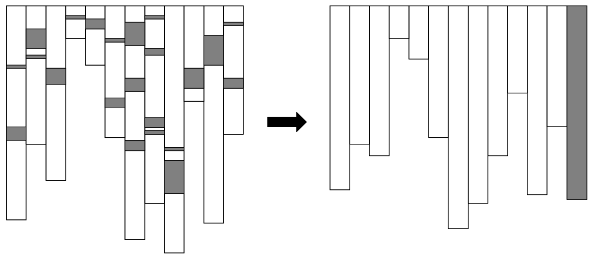


Figure 2. CCC modularization (schematic). A non-modularized CCC implementation is replaced with an equivalent aspect.

target OO program elements, but involve the creation of AOP ones. For example, the *Replace Implements with Declare Parents* refactoring [20] moves an `implements` declaration to an aspect by replacing it with the equivalent AspectJ construct `declare parents`. Replacing the explicit call to `aquireLock(..)` at the beginning of `signOut(Customer)` with a corresponding pointcut and advice pair is an example for such a 'move' refactoring.

Much like aspect-aware refactorings, refactorings for AOP constructs have a similar scope as traditional refactorings, i.e., they target single program elements or localized sets of them.

2.3 Modularizations of CCCs

Modularizations of crosscutting concerns (CCCs) transform scattered CCC implementations into a modularized form (an aspect), which is schematically shown in Figure 2.

For instance, replacing all code pertaining to preventing race conditions in the entire library system with an equivalent, modular AOP implementation constitutes a CCC modularization. Or, consider that the `Book` class would incorporate an update mechanism, in which interested objects can register themselves and are notified whenever the book's status is accessed or changes. Such functionality would in OO likely be implemented using the Observer design pattern [7] (not shown). Replacing the code pertaining to this mechanism with an AOP version of the pattern [12] is another example for such a modularization.

Non-modularized CCCs consist of multiple related program elements. Their relationships can be either explicit through code-level dependencies, such as method calls, subtyping, or *contains*-relationships, or implicit, such as access to the same data files. These refactorings are generally composite refactorings, consisting of multiple aspect-aware OO refactorings and refactorings for AOP constructs. Other than mere examples[1] for and experience reports (e.g., [19]) of CCC modularizations, two approaches – with differing tradeoffs – actively support them.

2.3.1 Extract-to-Aspect Refactorings

Recently Binkley et al. have developed a partially implemented, human-guided approach to support OO-to-AO refactorings [1], which focuses on the specific problem of extracting OO code fragments into aspects. Their approach relies on code bases in which the code segments pertaining to the implementation of the target CCC have been specially marked. Identifying and marking

[1] E.g., R. Laddad's *Aspect-oriented Refactoring Series.* Online document at The Server Side, http://www.theserverside.com/, 2003.

the CCC code is currently done manually, but the authors plan to employ automated concern mining approaches in future versions.

Their tool supports an interactive aspect extraction workflow consisting of *discovery* (determines applicable refactorings for the marked code), *transformation* (applies enabling OO refactorings), *selection* (lets the developer select appropriate AOP refactorings), and finally *refactoring*, which transforms the code. The tool does not yet create aspect code, but the authors assure this poses no conceptual difficulties.

The discovery phase uses TXL transformation rules to identify and suggest candidates for one of the five supported extraction refactorings (such as the *Extract Beginning/End* refactoring, which replaces a code block at the start/end of methods with appropriate `before`/`after` advice, respectively). A developer can review the suggestions or perform two different OO transformations (*Statement Reordering* and *Extract Method*) to potentially enable further suggestions. This process of discovery and selection is iterative, and meant to be performed until all marked code is refactored.

In the library example, consider refactoring the synchronization concern: all code pertaining to acquiring and releasing locks when signing out / returning books would initially be marked by the developer. The the tool would then suggest applicable refactorings (such as an *Extract End* refactoring for the call to `releaseLock(..)` in the `signOut(..)` method), and the developer can further perform statement reorderings so that *Extract Beginning* applies to the call to `aquireLock(..)`, or move on to the next marked code segment.

Binkley's approach treats CCC modularizations a series of low-level AOP refactorings that are restricted to the replacement of code fragments with pointcut and advice pairs; the overall structure of the CCC and the relationships between extracted code segments is not explicitly taken into account. The approach has very few requirements with respect to the structure of the concern to be extracted, and applies whether the marked code represents a single concern, multiple concerns, or just parts of one concern. On the other hand, the resulting aspect code has very little structure to it, as it consists of an accumulation of pointcut and advice pairs. If more meaningful aspect structures are desired, developers have to manually refactor the resulting aspect code, or utilize role-based refactoring, as outlined in the next section.

2.3.2 Role-Based Refactorings

Prior to Binkley's work, role-based refactoring (RBR) was proposed as a technique to both capture and refactor CCCs [13]. RBR describes the refactoring based on an abstract model of the target CCC's principal elements (called *role* elements) and their relationships. Role elements represent sets of program elements, such as classes, methods, and fields, which fulfill the same purpose within a concern's implementation. The idea behind RBR is that for all concrete program elements corresponding to a given role element, refactoring instructions are the same. RBR allows for a description of the refactoring specific to the target concern, yet independently from an actual implementation. Refactorings thusly take the concern structure into account and can further apply to multiple possible implementations of the same CCC.

In the library example, the logging CCC would be described as consisting of two role methods (e.g., *getLock(..), releaseLock(..)*), and potentially their enclosing type *CurrencyControl* as a role type[2]. Refactoring instructions are then defined in terms of these elements, one of which would likely be: "apply the *Replace Method Call With Pointcut and Advice*" refactoring [10] to calls to the *getLock(..)* and *releaseLock(..)* role methods". In order to apply the refactoring to a concrete system, like the library example, a mapping between role elements and program elements in the system is required. For example, by specifying that the concrete method `aquireLock(Book)` corresponds to the role method *getLock(..)*, the refactoring instructions defined on *getLock(..)* can be applied to `aquireLock(Book)`.

Replacing crosscutting OO design pattern implementations with their AOP equivalents (see [12]) has been shown to be a useful application of this technology [13], but the approach is not limited to design pattern refactorings, and can be used to capture and refactor other, non-pattern concerns (such as logging) as well [10]. The RBR approach has been realized in a refactoring plugin for the Eclipse IDE. The structural model of the concern is further useful for determining the concern extent by comparing the abstract concern structure to the structure of the actual program it is applied to, helping to identify the entire concern implementation.

The concern model, while allowing for complex transformations, is also the main limitation of the approach. For example, if the actual implementation varies too much from the expected concern structure (as can be the case with design pattern variants, for example), the refactoring description or the target system may need to be adjusted before the RBR refactoring can be applied. In case that the concern structure and its principal elements are not known to the developer, the extract-to-aspect approach mentioned above is more suitable.

3. CHALLENGES

Even though the three groups of refactorings have different purposes, there are many common technical challenges. This section outlines the challenges and potential ways to address them.

3.1 Utilizing Synergy

Some sub-problems are similar to all three areas of AOP refactoring research; all have to transform aspect-oriented programs. There is considerable overlap in the transformations used. For example, refactorings of crosscutting concerns generally consist of several smaller transformation steps, each of which is either an aspect-aware OO refactoring or a refactoring of an AOP construct. But on a very fine-grain level, all refactorings consist of multiple transformation steps. Insights regarding these *elementary* refactorings should therefore be shared.

Ideally, the wheel should not be re-invented. For example, Binkely's *Extract Beginning/End* [1] and Monteiro's *Extract Fragment to Advice* [20] perform basically the same function[3]. Similarly, Hannemann's *Move Object Member to Aspect* [10] is equivalent to Monteiro's *Move Field/Method from Class to Inter-Type* [20] and Iwamoto's *Move Class Method/Field to Advice*

[2] Names for role elements in a role-based refactoring description are not tied to an actual implementation, but are meant to abstract and describe principal functionality.

[3] Monteiro's version appears more general, but given the limitation of AspectJ advice, the two are essentially the same.

[14]. The above example illustrates that the same technique can be assigned multiple names, which can be confusing for AOP refactoring research. In other cases, refactorings have different granularity. For example, Monteiro's *Move Method from Class to Inter-Type* and *Replace Inter-Type Method with Aspect Method* are combined in Hannemann's *Replace Object Method with Aspect Method*. A refactoring catalog, like the one proposed by Monteiro [20], can help to systematically collect elementary AOP refactorings, and give researchers an idea what transformations have already been considered, thus avoiding duplicating work.

3.2 Suggesting AOP Refactorings

Part of the challenge of adopting AOP for an OO system is not only the transformation of non-modularized (latent) CCCs into aspects, but also their prior identification in the code. Since latent CCCs have scattered and tangled implementations, differentiating concern code from the rest of the system is a complex task. Combining refactoring approaches with support for identifying latent CCCs in a software system serves two purposes: first, such support can provide suggestions to the developer on how to improve the structure of their code. This can be particularly helpful if the developer is not familiar enough with the code base to identify latent aspects by hand. Second, it can help to ensure that the concern is identified completely. This is important, since if the not all program elements comprising the concern are correctly identified, the resulting refactoring will be incomplete.

Aspect mining approaches can provide a starting point for CCC modularizations and suggest refactoring candidates by identifying program elements pertaining to the implementation of non-modularized crosscutting concerns. Such mining approaches are either based on textual patterns [24], patterns in execution traces [2], high fan-in methods [17], or duplicated code fragments [3]. A qualitative comparison of aspect mining techniques is provided by Ceccato [4]. Recent approaches utilize or plan to utilize aspect mining to identify candidate sets of code fragments for extraction to aspects [1, 10, 22]. RBR, for example, uses an algorithm to help map role elements to concrete program elements, which helps to determine the extent of a crosscutting implementation by comparing the structure of the target program and the abstract concern description.

Both CCC modularization approaches mentioned above can conceivably be extended so that aspect mining techniques can be utilized in future versions. In the case of Binkely's approach, this is already part of the refactoring model [1]. The output of the aspect mining algorithm can be directly used to avoid manually making the code to be extracted. For the RBR case the information can help choose a proper refactoring from the library of CCC modularizations and provide an initial role mapping.

3.3 Preservation of Behavior and Intent

For AOP refactorings there is occasionally a trade-off between behavior preservation and preservation of *intent* of the code. Consider, for instance, inlining the `helper(Customer)` method in the library system shown in Figure 1 (an aspect-aware OO refactoring). This transformation would require changes to the `signOutBook(..)` pointcut since the pointcut references that method. The intent of the pointcut is to capture all books being signed out. To preserve the original behavior, we could define the logging to happen before the next program statement is executed (i.e., before the call to `releaseLock(..)`). Alternatively, we could decide that the logging should happen at the last statement of the original, inlined method, where the counter gets incremented. Both approaches fail to capture the original intent of the pointcut and make it less readable and self-explanatory. To maintain readability of our pointcut, we could instead specify that logging should take place after the `isAvailable` field is modified. This would require accepting a minor behavior variation, which, while being against the fundamental principle of refactoring, would keep the pointcut readable and the intent of the pointcut intact. Hanenberg shows how pointcuts can become very complex as part of an aspect-aware behavior-preserving refactoring [9], which can be seen as a motivation for intent preservation (as opposed to behavior preservation).

3.4 Aspect Generation

When creating aspect code, refactoring tools should aim for generic and reusable implementations to facilitate the later evolution of the software system. Pointcuts in particularl can hinder evolution if not chosen properly. The potential problem of generating pointcuts automatically is already apparent in Hanenberg's work [9]. Kellens and Gybels touch on the issues involved with the specific subproblem of generating pointcuts in the course of *Extract Method Calls* refactorings and propose a machine leaning approach to overcome them [15].

3.5 Program Analysis for AOP Refactorings

Modifying an AOP system can have a number of undesired effects (for a partial list, see [11]), if the transformation not performed carefully. Program analyses can help identify the applicability of a refactoring, specifically whether a planned refactoring step has an adverse effect on the behavior or intent of the affected program. This applies both to low-level program transformations and (composite) CCC refactorings. Thankfully, AOP refactorings can utilize techniques for OO program analyses to a large extent.

3.5.1 Resolving Generalizations

Several AOP refactorings involve generalizations. For example, aspect-aware versions of the *Extract Interface* or *Pull Up Member* [5] refactorings, but generalizations can also be included in CCC refactorings like *Extract Interface Implementation* [10]. Tip et al. present an approach for the detection, analysis, and resolution of (object-oriented) generalization refactorings [23] based on type constraints, including updating variable declarations in the course such refactorings. Although their approach is designed for use in an object-oriented context and an extension to AOP might introduce new challenges (consider, for example, how `declare parents` statements might affect the type constraints used), it can provide the basis for an equivalent aspect-oriented analysis.

3.5.2 Adapting AOP Systems to Use Generics

Java 1.5 introduced generic types, and AspectJ 5 generic aspects. Adapting an existing system to make use of generics is a challenge in itself, and can involve all three kinds of AOP refactorings. For the analyses involved, however, the type-constraint-based approach presented by Fuhrer [6] for OO transformations could be a useful basis. If Fuhrer's work would be extended to account for AOP constructs, it can be utilized for all three kinds of AOP refactorings.

3.5.3 Classifying Behavior Deviations

An approach presented by Mens et al. formalizes behavior preserving program transformations as basic graph rewriting operations and allows to statically analyze the dependencies between these operations at a semantic and syntactic level [18]. Especially interesting about this approach is the differentiation between various kinds of behavior preservation. It is conceivable that by classifying these and other kinds of behavior preservation according to their impact on the rest of the system, a tool could estimate which refactorings are behavior-preserving, which might produce “acceptable” behavior deviations, and which might violate the intent of the implementation.

3.5.4 Composing AOP Refactorings

CCC refactorings consist of series of lower-level AOP refactorings. When taking into account multiple, consecutive program transformations, an analysis of their overall effect is difficult because of potential effects of the individual steps on each other. A promising analysis approach for this problem is provided by Kniesel and Koch [16]. Their generic formal model for the automatic composition of conditional program transformations is program independent and applies to arbitrary conditional program transformations (even non-behavior-preserving ones).

4. CONCLUSION

Current research in AOP refactoring focuses on three related problems: extending existing OO refactorings to properly handle references to the affected program elements in AOP constructs, developing new refactorings that explicitly target AOP constructs, and providing refactoring support for the transformation of non-modular CCC implementations into aspects. The first two kinds represent low-level program transformations that can be employed similarly to traditional refactorings and serve as building blocks for composite AOP refactorings, such as CCC modularizations.

Research in the area of AOP refactoring is very synergistic, and it is important that the overlap between the different refactoring techniques is not only recognized, but also utilized. Additionally, related work that has been done for object-oriented refactorings can be leveraged to a large extent, as outlined in this paper.

5. ACKNOWLEDGMENTS

Thanks to Hidehiko Masuhara for providing valuable comments on earlier drafts of this paper. This work was funded by a JSPS fellowship.

6. REFERENCES

[1] Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P. Automated refactoring of object-oriented code into aspects. Proc. ICSM ’05, pp. 27–36. IEEE Computer Society, 2005.

[2] Breu, S., Krinke, J. Aspect mining using event traces. Proc. ASE’04, pp. 310–315. IEEE Computer Society, 2004.

[3] Bruntink, M., Deursen, A., Tourwe, T., van Engelen, R. An evaluation of clone detection techniques for identifying crosscutting concerns. Proc. ICSM ’04, pp 200–209. IEEE Comp. Society, 2004.

[4] Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., Tourwe, T. A qualitative comparison of three aspect mining techniques. Proc. IWPC ’05, pp. 13–22. IEEE Computer Society, 2005.

[5] Fowler, M. Refactoring: Improving the Design of Existing code. Addison-Wesley, 1999.

[6] Fuhrer, R., Tip, F., Kiezun, A., Dolby, J., Keller, M. Efficiently refactoring Java applications to use generic libraries. Proc. ECOOP’05, LNCS vol. 3586, pp. 71–96. Springer, 2005.

[7] Gamma, E, Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[8] Griswold, W. Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington, Seattle, WA, USA, 1991.

[9] Hanenberg, S., Oberschulte, C., Unland, R. Refactoring of aspect-oriented software. NetObject Days ’03, Erfurt, Germany, 2003.

[10] Hannemann, J. Role-based refactoring of crosscutting concerns. PhD thesis, University of British Columbia, BC, Canada, 2005. http://www.cs.ubc.ca/~jan/

[11] Hannemann, J., Chitchyan, R., Rashid, A. Report on the WS on analysis of aspect-oriented software. ECOOP’03 WS Reader, LNCS vol. 3013, pp. 154–164. Springer, 2004.

[12] Hannemann, J., Kiczales, G. Design pattern implementation in Java and AspectJ. Proc. OOPSLA ’02, pp. 161–173. ACM Press, 2002.

[13] Hannemann, J., Murphy, G., Kiczales, G. Role-based refactoring of crosscutting concerns. Proc. AOSD ’05, pp. 135–146. ACM Press, 2005.

[14] Iwamoto M., Zhao, J. Refactoring aspect-oriented programs. WS on AOSD Modeling With UML at UML ’03, 2003.

[15] Kellens, A., Gybels, K. Issues in Performing and Automating the "Extract Method Calls" Refactoring. WS on Software Engineering Properties of Languages and Aspect Techn. at AOSD ’05, 2005

[16] Kniesel, G., Koch, H. Static composition of refactorings. Science of Computer Programming, 52:9–51, 2004.

[17] Marin, M., van Deursen, A., Moonen, L. Identifying aspects using fan-in analysis. Proc. WCRE ’04, pp. 132–141. IEEE Computer Society, 2004.

[18] Mens, T., Demeyer, S., Janssens, D. Formalising behaviour preserving program transformations. Proc. ICGT ’02, LNCS vol. 2505, pp. 286–301. Springer, 2002.

[19] Monteiro, M. Fernandes, J. Refactoring a Java Code Base to AspectJ: An Illustrative Example. Proc. ICSM ’05, pp. 17-26, IEEE Computer Society, 2005

[20] Monteiro, M. and Fernandes, J. Towards a catalog of aspect-oriented refactorings. Proc. AOSD ’05, pp. 111–122. ACM Press, 2005.

[21] Opdyke, W., Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[22] Shepherd, D. Pollock, L. Ophir: A framework for automatic mining and refactoring of aspects. TR 2004-03, Dept. of Computer and Information Sciences, Univ. of Delaware, DE, USA, 2003.

[23] Tip, F., Kiezun, A., Bäumer, D. Refactoring for generalization using type constraints. Proc. OOPSLA’03, pp. 13–26. ACM Press, 2003.

[24] Tourwe, T., Mens, K. Mining aspectual views using formal concept analysis. Proc. WS on Source Code Analysis and Manipulation (SCAM ’04), pp 97–106. IEEE Computer Society, 2004.

[25] Wloka, J. Refactoring in the presence of aspects. WS for PhD Students in Object-Oriented Systems. In ECOOP ’03 WS Reader, LNCS vol. 3013, pp. 50–61. Springer, 2003

Reasoning about assessing and improving the candidate-seed quality of a generative aspect mining technique

Marius Marin
Software Evolution Research Lab
Delft University of Technology
The Netherlands
A.M.Marin@ewi.tudelft.nl

Abstract

We propose a new measure for assessing generative *aspect mining techniques: the* candidate-seed quality. *We show the relevance of this measure and investigate how it can be improved for* fan-in analysis, *a generative mining technique that we have proposed in a previous work. The investigation results in a number of properties aimed at improving the quality of the candidate-seeds reported by fan-in analysis.*

1. Introduction

Aspect mining is a research area aimed at developing techniques for automatic identification of crosscutting concerns in existing code. A number of approaches to aspect mining provide support for identifying and investigating the code pertaining to a crosscutting concern starting from a *seed*: a program element (method, interface or group of statements) that is part of the crosscutting concern implementation. We describe these approaches as *explorative* or *query-based* approaches.

A second category of aspect mining approaches focus on identifying crosscutting concern seeds by looking in the source code for symptoms of crosscutting behavior, like tangling and scattering. We group these techniques in the *generative* category.

The mining techniques, and the generative ones in particular, face serious challenges in describing, comparing and combining their results due to (1) the lack of a clear definition of the crosscutting concerns, and (2) their focus on generic symptoms (tangling and scattering) that can occur in various concerns, of different level of granularity. This poses further questions about how the results of a combination of aspect mining techniques can be assessed, how to show improvement of results, and what measures are relevant for assessing these results and a mining technique in general.

To address these problems, we have proposed a classification system of the crosscutting concerns based on *sorts* (atomic, generic crosscutting concerns) and a preliminary set of sorts [4, 5]. The crosscutting concern sorts are described by their specific symptoms (i.e., implementation idiom in a non-aspect-oriented language), and a (desired) aspect mechanism to refactor a sort's concrete instances to an aspect-based solution. Sorts are relevant for aspect mining because they associate symptoms to generic concerns and are able to provide a focus for mining techniques and criteria for comparing their findings. Moreover, they are by definition *atomic* elements, and hence able to ensure a consistent granularity level for comparison.

In this work we further consider how the (generative) aspect mining techniques can be assessed, by proposing and discussing a new measure: the *(candidate-)seed quality* is an indicator of the human effort involved for analyzing the results of an aspect mining technique. We look in more detail at this measure and how it can be improved for a specific mining technique, namely fan-in analysis. Our investigation results in a number of properties to be considered for improving the (candidate-)seed quality for this technique.

2. Assessing generative mining techniques

The results of a generative aspect mining technique are *candidate-seeds*: program elements that might pertain to a crosscutting concern implementation, and which can be marked as either seeds or *false positives* by an human analyzer examining the output of the technique. The decision of the human analyzer is typically supported by a number of elements provided by the mining technique for reasoning about its output: grouping of results by some criteria (e.g., naming), (structural) relationships between properties describing the results (e.g., related method calls), etc.

For simplification, we will refer to candidate-seeds in the following sections simply as *candidates*. We will also use *seed quality* if we discuss about candidates that have been marked as seeds.

A serious limitation of the (generative) aspect mining techniques consists of lacking a clear definition of the crosscutting concerns and thus of those concerns they aim to identify. As a result, most of the techniques focus on generic symptoms of crosscutting behavior, like tangling and scattering. This makes the comparison and description of their results difficult:

- How is the reported candidate (program element) related to a crosscutting concern, and to which concern?
- How relevant is this relation for the mining technique: Is this an expected result or just an accidental match?
- Are the reported candidates relevant for the crosscutting nature of the associated concern, or just part of its broader, more complex implementation?
- Is the technique able to find similar concerns and is it able to easily distinguish between concerns that are not similar?

To answer these questions we believe that the focus of a mining technique should not be (only) on general symptoms but (also) on expected results (and specific symptoms): generic crosscutting concerns (sorts) that the technique expects to identify. This would allow for a clear assessment of the results (whether they are a relevant element of an expected sort instance) and the technique.

2.1. Sorts of crosscutting concerns

A *crosscutting concern sort* is a generic, atomic crosscutting concern described by a number of properties common to all its (concrete) instances like: (1) a general intent, (2) an implementation idiom in a non-aspect-oriented language (i.e., a specific symptom), (3) and a (desired) aspect mechanism to support the modularization of the sort's concrete instances.

Table 1 shows several sorts from a longer list of proposed canonical sorts [4]. These are described by the three elements defining them as well as several examples of concrete instances.

Complex examples of crosscuttingness described in literature, like design patterns[2], are compositions of sort instances: an Observer pattern implementation, for example, involves instances of the *role superimposition* sort for the two defined roles (Subject and Observer), as well as instances of *consistent behavior* for the consistent actions of listeners notification and registration.

2.2. Defining "Target" crosscutting concern sorts for fan-in analysis

Fan-in analysis generates candidates based on the fan-in metric of a method: if a method is called from many, scattered places, the method is considered a potential seed. Hence fan-in is essentially a metric for the scattering symptom of the crosscutting concerns.

To focus our analysis, we define the targeted crosscutting concerns of fan-in analysis as instances of the Consistent behavior sort. The sort is shown in Table 1; it is described by an action and a number of (method) elements that consistently execute the action as part of their complete functionality. These elements, which are crosscut by the invocation of the specific action, are part of a context that can be formalized by a pointcut definition.

Concrete instances of the sort comprise credentials checks as part of the authorization mechanism, or logging of exception throwing events in a system.

2.3. (Candidate-)seed quality

We propose the (candidate-)seed quality as a measure for assessing the results of an aspect mining technique. The measure is defined as the percentage of elements in the (candidate-)seed that belong to the implementation of the crosscutting concern associated with the candidate, if any. The percentage of these elements gives a measure of the effort involved in reasoning and deciding about a candidate-seed.

In order to measure the (candidate-)seed quality, a technique has to describe the way one should reason about a candidate (i.e., the output of the technique); that is, we need to know what are the elements describing the candidate-seed and what is their relation with the targeted concerns.

Returning to the fan-in analysis example, the results of the analysis are described by the method with a high fan-in value and the callers of this method. However, not all the calls to the method with a high fan-in value are necessarily crosscutting, or part of a (same) Consistent behavior instance. A fan-in candidate for Consistent behavior is labeled as seed if its callers are part of a relationship (i.e., context definition) that can be formalized in a pointcut expression. The call reported by the technique crosscuts the elements in this context.

3. Candidate-seed quality for fan-in analysis

We discuss the quality measure for the fan-in technique and investigate what attributes could be relevant for improving it. The discussion uses for exemplification a number of selected results from the JHOTDRAW case-study.

JHOTDRAW [1] is an editor for 2D graphics developed as an open-source project and a show-case for how to apply design patterns [1]. We used the application as a case-study for fan-in analysis in a previous work, in which we also describe the concerns associated with the results of the analysis [3].

3.1. Seed quality for a number of concerns in JHOTDRAW

The selected examples from JHOTDRAW (typically) involve several crosscutting elements (i.e., sorts instances), like, for instance, Role superimposition and Consistent behavior instances for the Observer pattern implementation. We look at how the elements pertaining to Consistent behavior instances in these complex examples are identifiable by fan-in analysis, and what is the quality of the seeds for these instances.

[1]www.jhotdraw.org

Sort	Intent	Object-oriented Idiom	Aspect mechanism	Instances
Consistent Behavior	Implement consistent behavior as a controlled step in the execution of a number of methods that can be captured by a natural pointcut.	Method calls to the desired functionality	Pointcut and advice	Log exception throwing events in a system; Wrap/Translate business service exceptions [3]; Notify and register listeners; Authorization;
Contract enforcement	Comply to design by contract rules, e.g., pre- and post-conditions checking	Method calls to method implementing the condition checking	Pointcut and advice	Contract enforcements specifi c to design by contract
Redirection Layer	Defi ne an interfacing layer to an object (add functionality or change the context) and forward the calls to the object	Declare a routing layer (decorator/adapter), and have methods in this layer to forward the calls	Pointcut and around advice	Decorator pattern, Adapter pattern [2]; Local calls redirection to remote instances (RMI) [6];
Role superimposition	Implement a specifi c secondary role or responsibility	Interface implementation, or direct implementation of methods that could be abstracted into an interface defi nition	Introduction mechanisms	Roles specifi c to design patterns: Observer, Command, Visitor, etc.; Persistence [3]

Table 1. Sorts of crosscuttingness.

Seed	Composite concern	Targeted sort instance	Fan-in value
UndoableAdapter.undo()	Undo	YES	24
UndoableAdapter.UndoableAdapter(DrawingView)	Undo	YES	25
Undoable.isRedoable()	Undo	YES	24
Figure.addFigureChangeListener(FigureChangeListener)	Figure Change Observer	YES	11
Figure.changed()	Figure Change Observer	YES	36
Figure.listener()	Figure Change Observer	NO	21
Figure.removeFigureChangeListener(FigureChangeListener)	Figure Change Observer	YES	10
Figure.willChange()	Figure Change Observer	YES	25

Table 2. Selected fan-in seeds from JHOTDRAW

The selection of the concerns is aimed at showing various relations between the elements (callers) provided by the technique for reasoning about its results.

3.1.1. Undo

The *Undo* concern involves around 30 classes like *commands*, *tools*, and *(figure) handles* elements. The changes spawned by the execution of the activities associated with these elements can be undone by specialized, dedicated *UndoActivites*. The *UndoActivity* classes are nested within their associated activities and implement the *Undoable* interface.

Fan-in analysis reports three seeds for the *Undo* concern, all of them part of instances of the targeted sort(s). The seeds are shown in Table 2. The first seed corresponds to a method implementing undo functionality and which is called by 24 methods. Most of the callers (22) are implementations of the *undo* method in the nested (*UndoActivity*) classes. The 22 callers follow the same idiom to invoke the reported seed: the invocation is the first call in the caller to check if the specific activity can be undone. The other two callers (*UndoCommand.execute* and *UndoRedoActivity.redo*) do not follow this idiom and the call is not part of a consistent behavior like for the other callers.

In order to decide about this candidate, we have to be able to observe specific symptoms of consistent behavior, like the structural relation between the first 22 callers to define a context, or the similar positions of the calls. The other two callers, on the other hand, make the analysis of the candidate harder, because they have to be investigated although they turn out not to be part of the consistent behavior concern. The relevant elements in the analysis of the callers are the first 22 callers and hence the quality is 22/24 (92%).

Similar, the quality for the other two candidates is 22/25(88%) and 20/24(83%), respectively.

3.1.2. Figure Change Observer

The *Figure* elements in JHOTDRAW participate in an implementation of the Observer pattern (Figure Change) by implementing the Subject role. A number of elements listen for changes in *Figure*s by implementing the *FigureChangeListener* interface. The concrete figures implement or inherit the methods specific to the Subject role for allowing (de-)registration of listeners, and notification of changes in their state.

The (de-)registration action is a consistent behavior for the listeners of figure changes. This is implemented as a call to the method that adds/removes listener objects for a *Figure*: *Figure.add-/remove-FigureChangeListener*.

The notification of changes occurred in the *Figure*s' state is also a consistent behavior applying to elements changing this state. The concern is implemented as a call to the notification method: *Figure.willChange*, before making the changes, and *Figure.changed*, after the changes were made.

The consistent behavior for these instances of the sort is due to the relation between the callers and the callee in the context of the pattern: the listeners interested in changes have

Seed	**Orig. Quality**	**Same hierarchy(same method)**	**Role relation**	**Same Call Position**
UndoableAdapter.undo()	22/24 = 92%	22/23(22/22) = 96% (100%)	22/23 = 96%	22/22 = 100%
UndoableAdapter.UndoableAdapter(DrawingView)	22/25 = 88%	22/22(22/22) = 100%(100%)	22/22 = 100%	22/22 = 100%
Undoable.isRedoable()	20/24 = 83%	19/21(18/19) = 90%(95%)	19/21 = 90%	18/21 = 86%
Figure.addFigureChangeListener(FigureChangeListener)	11/11 = 100%	Not relevant	10/10 = 100%	Not relevant
Figure.changed()	36/36 = 100%	Not relevant	Not relevant	33/33 = 100%
Figure.listener()	-	-	-	-
Figure.removeFigureChangeListener(FigureChangeListener)	10/10 = 100%	Not relevant	9/9 = 100%	Not relevant
Figure.willChange()	25/25 = 100%	Not relevant	Not relevant	20/20 = 100%

Table 3. (Candidate-)seeds analysis for selected concerns

to register in order to be notified, and the actions changing the state of the Subject object have to notify about this change. Hence all the callers of the reported seeds are relevant because the calls occur due to the participation into the pattern implementation. This participation is a secondary concern for the callers. Moreover, the concern implemented by the callees (the reported high fan-in methods) implies calls only from participants in the pattern, in the context of the pattern. The quality of these seeds is thus 100%.

Fan-in analysis reports several implementations of the described candidates. Table 2 shows the results omitting the multiple implementations that occur due to polymorphism.

The last reported method, *Figure.listener*, provides access to the listener reference in the class implementing the Subject role. The method is part of this role, however, it does not belong to a consistent behavior instance in the analyzed implementation of the Observer pattern.

3.2. Improving the quality of the fan-in candidates

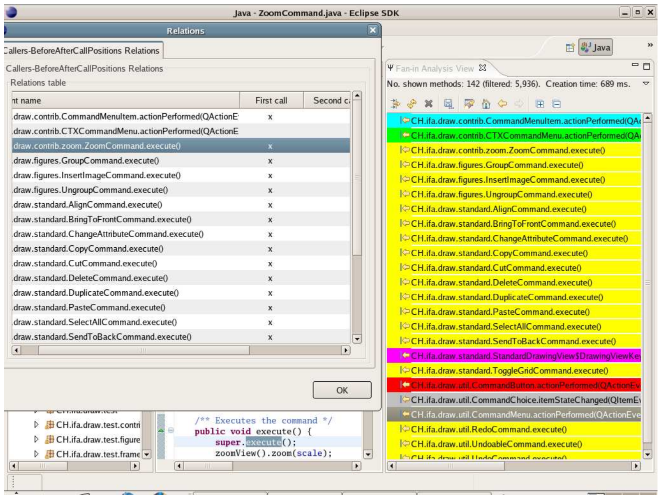
To improve the quality of the fan-in candidates, we propose a number of properties to be considered for analyzing the callers of a candidate. These properties show possible relations between the callers of a method with a high fan-in, and are aimed at reducing the percentage of irrelevant elements describing a candidate, and hence reduce the effort of reasoning about the candidate.

The list of proposed properties comprises:

- structural relations between the callers. These relations include:
  - same hierarchy: The methods (callers) are defined by the same interface (/super class). As a particular case, the callers could be implementations of the same method. The (callers of the) seed for the *Undo* concern previously discussed falls into this category.
  - common roles: A method is associated with all the roles implemented by its class, so that the methods can share common roles. A role is typically defined by an interface. Methods that belong to the same hierarchy will also share the role that defines the hierarchy: the callers of the discussed *Undo* seed are declared by the *Undoable* interface, which defines the main role of the classes implementing the concrete callers of the seed. Similarly, the callers of the registration method in the previous Observer example are associated with the *FigureChangeListener* role. The main role for these callers is, however, defined by other interfaces, like the *Figure* interface.
  - same class: The callers belong to the same class, as for the case of a class level contract.
  - etc.
- consistent call position: The position of the call, relative to the caller's body, is consistent for the callers of the reported method with a high fan-in value. Such a case has been shown for the seeds discussed in Section 3.1.
- naming-based relations: The callers have similar names. The naming-based and the structural relations can be expressed by an AspectJ-like pointcut definition, while the call position could be an indication of the advice type (before/after).
- relations based on the structure of the call: similar or, sometimes, identical call sites. For example, Exception wrapping concerns typically consist of catching a specific type of exception and re-throwing an exception of a different type [3]. Another example consists of calls that occur together, like the notification of changes in the previously discussed instance of the Observer pattern for Figure changes: typically, a method changing the state of a *Figure* object would call `willChange` before the modifications, and then the *changed* method after completing the modifications.
- "intentional" relations between callers, such as modifiers of Subject objects in the context of the Observer pattern. The relations between the callers are due to their participation in the pattern implementation.

3.3. Results after extending fan-in with a callers analysis

Table 3 shows the quality of the fan-in seeds after analyzing the callers based on some of the proposed properties.


Java - ZoomCommand.java - Eclipse SDK
Relations
Callers-BeforeAfterCallPositions Relations
Callers-BeforeAfterCallPositions Relations
Relations table
OK
Java
Fan-in Analysis View
CH.ifa.draw.test.contri
CH.ifa.draw.test.figure
CH.ifa.draw.test.frame

nt name	First call	Second c
.draw.contrib.CommandMenuItem.actionPerformed(QActionE	x	
.draw.contrib.CTXCommandMenu.actionPerformed(QActionE		
.draw.contrib.zoom.ZoomCommand.execute()	x	
.draw.figures.GroupCommand.execute()	x	
.draw.figures.InsertImageCommand.execute()	x	
.draw.figures.UngroupCommand.execute()	x	
.draw.standard.AlignCommand.execute()	x	
.draw.standard.BringToFrontCommand.execute()	x	
.draw.standard.ChangeAttributeCommand.execute()	x	
.draw.standard.CopyCommand.execute()	x	
.draw.standard.CutCommand.execute()	x	
.draw.standard.DeleteCommand.execute()	x	
.draw.standard.DuplicateCommand.execute()	x	
.draw.standard.PasteCommand.execute()	x	
.draw.standard.SelectAllCommand.execute()	x	
.draw.standard.SendToBackCommand.execute()	x	

```
/** Executes the command */
public void execute() {
    super.execute();
    zoomView().zoom(scale);
```

No. shown methods: 142 (filtered: 5,936). Creation time: 689 ms.

```
CH.ifa.draw.contrib.CommandMenuItem.actionPerformed(QA
CH.ifa.draw.contrib.CTXCommandMenu.actionPerformed(QA
CH.ifa.draw.contrib.zoom.ZoomCommand.execute()
CH.ifa.draw.figures.GroupCommand.execute()
CH.ifa.draw.figures.InsertImageCommand.execute()
CH.ifa.draw.figures.UngroupCommand.execute()
CH.ifa.draw.standard.AlignCommand.execute()
CH.ifa.draw.standard.BringToFrontCommand.execute()
CH.ifa.draw.standard.ChangeAttributeCommand.execute()
CH.ifa.draw.standard.CopyCommand.execute()
CH.ifa.draw.standard.CutCommand.execute()
CH.ifa.draw.standard.DeleteCommand.execute()
CH.ifa.draw.standard.DuplicateCommand.execute()
CH.ifa.draw.standard.PasteCommand.execute()
CH.ifa.draw.standard.SelectAllCommand.execute()
CH.ifa.draw.standard.SendToBackCommand.execute()
CH.ifa.draw.standard.StandardDrawingView$DrawingViewKe
CH.ifa.draw.standard.ToggleGridCommand.execute()
CH.ifa.draw.util.CommandButton.actionPerformed(QActionEv
CH.ifa.draw.util.CommandChoice.itemStateChanged(QItemE
CH.ifa.draw.util.CommandMenu.actionPerformed(QActionEve
CH.ifa.draw.util.RedoCommand.execute()
CH.ifa.draw.util.UndoableCommand.execute()
CH.ifa.draw.util.UndoCommand.execute()
```

Figure 1. FINT support for callers-analysis

The mined instances that are part of the Undo concern are representative for many (/most) of the concerns identified using fan-in analysis. In most cases, the quality of the seeds is improved by analyzing the proposed properties.

Although the original quality of the seeds for the Observer implementation is 100%, the analysis of the callers based on the proposed properties is meaningful due to the complex relation between the callers that has to be observed. Because the relation is intentional, it is expected that the investigated, mainly structural, properties to be irrelevant, offering little or no insight into the intentional relation. However, the structural properties can provide insight into the specific consistent behavior instances: Most of the callers (10 out of 11) of the registration method belong to classes that implement the listener role and that register themselves as listeners of a *Figure* object. The *common role* property groups and relates the 10 methods by this common, relevant concern.

A similar grouping and insight into the callers' relation can be obtained by considering the *call position* property for the callers of the notification method.

The tool support for fan-in analysis, FINT [2], allows for improving the quality of the results of the analysis. In the present version (v.0.5b), the user can display and investigate (1) relations between the callers of a method and their declaring types (i.e., common roles) or implementing classes, (2) relations between the callers and the position of the call, as well as (3) relations between the callers and all their callees.

Figure 1 shows a part of the tool support in FINT for the analysis of the callers. The user can investigate the relations between the callers of a method by their declaring interfaces: callers declared by the same interface are shown in a same color. Another analysis allows to look at the relations between the callers and the position of the call to the method with the high fan-in value. This position can be an absolute value or relative to the caller's body.

[2]http://swerl.tudelft.nl/bin/view/AMR/FINT

4. Discussion and conclusions

To be able to assess aspect mining techniques, we need objective measures. In this work, we proposed a quality measure aimed at assessing the relevance of the results of an aspect mining technique and the effort required to analyze these results. We showed how this measure applies to fan-in analysis, and identified a number of properties to improve the quality of the results of this mining technique.

The proposed analysis of the callers to reason about a fan-in candidate also shows the potential relations between the elements describing a candidate. Such relations are important for defining the context of the concern through a pointcut construct.

The situations described for the *Undo* concern, where the callers are related by structural relationships, are common to many of the concerns discovered by fan-in analysis. These include Consistent behavior and Contract enforcement instances in JHOTDRAW, as well as concerns in other analyzed case-studies, like PETSTORE and TOMCAT [3]. The intentional relationship, on the other hand, are typically harder to detect. The intention of a developer could be captured by annotations in the code, which to be analyzed by aspect mining techniques.

Acknowledgments The author would like to thank Arie van Deursen (TU Delft) for his reviews and feedback.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994.

[2] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, Boston, MA, 2002. ACM Press.

[3] M. Marin, A. van Deursen, and L. Moonen. Identifying Aspects using Fan-In Analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004).*, pages 132–141, Los Alamitos, CA, 2004. IEEE Computer Society Press.

[4] M. Marin, L.Moonen, and A. van Deursen. A classifi cation of crosscutting concerns. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society, 2005.

[5] M. Marin, L.Moonen, and A. van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *Int. Workshop on the Modeling and Analysis of Concerns in Software, ICSE*. Software Engineering Notes (volume 30, issue 4), 2005.

[6] S. Soares, E. Laureano, and P. Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proc. 17th Conf. on Object-oriented programming, systems, languages, and applications*. ACM Press, 2002.

On Using Metrics in the Evaluation of Aspect-Oriented Programs and Designs

Katharina Mehner[*]
Software Engineering Group
Technical University of Berlin
Germany
mehner@cs.tu-berlin.de

ABSTRACT

Metrics are an important technique in quantifying desirable software and software development characteristics of aspect-oriented software development (AOSD). Currently, metrics proposed for AOSD have rarely been validated. We give an overview on the necessary steps to validate definitions and applications of metrics. We also compare definitions for proposed metrics.

1. INTRODUCTION

In order to study the impact of aspect-oriented software development (AOSD) on evolution one has to study its impact on software characteristics such as evolvability, maintainability, understandability, and quality. To evaluate these characteristics not only qualitative but also quantitative techniques are of interest.

The key question is how we can quantify when applying AOSD is beneficial. *Metrics* are an important techique in quantifying software and software development characteristics. However, metrics have to be used knowledgeably and carefully. Theoretical and empirical validation of metrics and of their relation to software attributes is a cumbersome and long process. It is of paramount importance that we validate the *utility* of metrics we use in order to enable others to use them, too. Until now, the metrics used and proposed for AOSD are rarely validated. It is not sufficient to prove their definitions correct but also their usefulness to describe software characteristics has to be validated. In most cases, this can only be achieved through controlled experiments or through analysing large amounts of data, e.g., from case studies. In both cases, statistical evaluations are a key technique to examine hypotheses.

In the remainder of this position paper, we first make our problem statement more concrete. We give an overview on the underlying theory of metrics and their validation. We focus on product metrics but much of what will be said equally applies to process, resource, or project metrics. Here, we cannot provide a thorough theoretical discussion but we want to raise the level of awareness concerning the issues involved in using metrics. We then discuss and compare some concrete metric examples, which are often influenced by OO metrics. We conclude by calling for more empirical research in relation with metrics in AOSD.

[*]The author has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

2. PROBLEM STATEMENT

The benefit of AOSD is often studied in two ways. One is the comparison with an OO software, which is already existing or which sometimes is developed in paralled. The other one is the independent evaluation of AOSD, often by assessing evolution scenarios.

In both cases, the evaluation can employ design and program metrics to support and to quantify their observations. It has already noted by Chidamber and Kemerer that because of the abstractions introduced in OO, one can distinguish between design and code metrics [4]. The benefit of this distinction is that design metrics, although applicable also to code, can already be applied to the design when the coding has not yet begun. Apart from the difference, the validation requirements are the same. In the following, we discuss design and program metrics together. As already mentioned in the introduction, validation is twofold.

The *theoretical* or internal validation addresses the proper definition of a metric and guarantees that a metric fulfils generally accepted (axiomatic) properties (for an overview see [5]). Often, metrics are defined using the representational theory of measurement. In this theory, measurement is regarded as "the correlation of numbers with entities that are not numbers". For an empirical attribute, there should be an empirical observation of a relation in order to consider defining a metric. Then one can try to establish a mapping to a numerical presentation which preserves empirical relations. To ensure preservation of relations and proper definition a number of axioms or properties have to be fulfilled [5]. Using this method, it can be avoided that for a metric we desperately seek an empirical counterpart. There exist competing and sometimes contradictory sets of axioms or properties for validating metric definitions, for instance the axioms by Weyuker [12] or the validation framework by Kitchenham et al. [8].

The *empirical* or external validation of a metric addresses its utility in describing desirable properties or predicting desirable properties of software (for an overview see [5]). For a metric to be useful in assessing a software product or process quality, i.e., a characteristic, we need a validation that shows that this metric has a de facto influence on a characteristic or that it serves in predicting characteristics of a process. Such a cause-effect relation can not be validated by merely statistically proving a correlation, but a correlation can be an indicator as to what should be examined. Instead, an experiment with a falsifiable hypothesis has to be carried out, or data gathered from case studies or real projects have to be evaluated. Empirical validation thus can be based either on controlled experiments or on case studies which produce enough data or data collected from real projects. The use of statistical tests is helpful to discover correlations between variables of controlled experiment [8, 5, 3].

Empirical validation is also required when a *quality model* is used to define relationships between metrics and software characteristics. For instance, the Goal-Question-Metric (GQM) model is used to identify what must be measured in order to answer the question from which the goal of an empirical study can be determined [2]. This model helps to define and select appropriate metrics. However, especially the relation of the questions to the metrics has to be empirically validated for each new model derived.

Given these steps for validating metrics, the state-of-the-art in metrics for AOSD can be assessed. Metrics related to desirable design and program characteristics have been proposed amongst others by Lopes [9], Zhao [14], Garcia [10], and Ceccato [11]. Individual examples therof will be shortly discussed in Sect. 3. The theoretical validation has only been addressed in some of these publications. Empirical validation is even harder to achieve and has been rarely addressed. The research by [10] has included an empirical validation which indicates the usefulness of the proposed metrics through two case studies. This is a first step towards an empirical validation of the metrics. The authors acknowledge that this does not yet provide a complete validation. It would be desirable to accomplish their studies by planned, controlled experiments to validate their hypotheses. Note that controlled experiments in AOSD have been used to gather qualitative data as in [7, 1].

Some of these approaches use metrics in the comparison of AO and OO design. Here, the challenge is to validate that metrics which have been extended from OO to AO permit the direct comparison. Without empirical validation it is particularly dangerous to interpret values of metrics as better or worse than other values with respect to software characteristics. One has to be careful with taking such statements for granted.

3. METRICS FOR AOSD

The examples discussed in the following present by no means a complete overview. We have selected them to show that different ideas exist on how to define AOSD metrics.

For defining AOSD metrics it is obvious to build on existing general software metrics and especially on existing OO metrics [4]. Thus, it is a key issue, how to extend metrics and how to define corresponding metrics besides defining completely new metrics. Most existing metrics cannot be applied straightforwardly to aspect-oriented software, since AOSD introduces new abstractions and new composition techniques. Consequently, there are new kinds of coupling and cohesion. Not only new metrics but also extended metrics to cover a new programming language have to be validated.

Some characteristics and attributes have been widely acknowledged as playing a key role in the evolability of software, both for OO and AO, such as size, coupling, cohesion and separation of concern, and because they are obviously related to aspect-oriented modularisation of crosscutting concerns. We can observe two approaches with respect to defining metrics for these characteristics, *extending* metrics and *defining new* metrics. Here we shortly sketch the different ideas behind the two approaches.

Separation of concern metrics

Separation of concern metrics are new ones. In [9] the first separation of concern metrics were proposed. In [10, 6] these metrics have been refined. Until now, these metrics requires manually identifying concerns.

Coupling metrics

Coupling is interesting because it is treated both ways. Initially, the coupling-between-objects metric [4] is defined as counting all classes (once) to which a class is coupled. This is in turn defined by counting the classes of the objects on which a given object/class "acts upon". This refers to access or method calls on instance variables, local variables or formal parameters. Although the metric relies on different kinds of variables or parameter, essentially the same kind of access is counted, i.e., that an instance of another class is accessed via a reference and potentially via its attributes or methods. Also note that in the OO context, high numbers of coupling are considered as undesirable.

In [13] no new metrics are defined but the preservation of the existing definitions is assumed. They focus on the effects aspects typically have on these metrics, e.g., they discuss that OO coupling may be decreased. Thus, their work implies the necessity to be able to understand the OO part on its own, e.g. by preserving OO metrics.

In [10, 6], this original metric is extended to cover also coupling between aspects. This means that the metric still counts the same kinds of coupling as before, i.e., coupling to classes used in declarations of attributes, parameters and local variables, but now includes declarations that belong to aspects.

Other authors [11] have decided to provide separate metrics for the new kinds of couplings found in AO. The metric can be used as an indicator for different effects of the different kinds of coupling in a validation experiment. Although the different metrics mentioned so far, coupling for OO and coupling for AO, are all related to coupling, it should be avoided to compare them directly as they describe coupling related to two different paradigms.

Cohesion metrics

Cohesion has been addressed in both ways, too. [14] specifically looks at a new way for defining cohesion. Others extend existing metrics such as lack-of-cohesion in methods straightforwardly to advice [10].

Size metrics

Size metrics are often critized as being dependent on programming styles and being too simple. Size itself has an empirical counterpart in the physical length though there are numerous ways to define it, e.g., LOC. From this viewpoint it might be admissable to count a pointcut as a code line. In [10, 6], experience with an extended LOC metric has been gathered.

Size metrics can point to the fact that duplicated code is avoided. The question is, if LOC is a good indicator for maintainability and reusability when comparing AO with OO. Avoiding duplicate code might be achieved by writing difficult-to-maintain pointcuts.

4. CONCLUSION

This paper gave a short overview on the necessary steps for validating metrics that are to be used in an evaluation process. These steps are well-known in software engineering. The current state-of-the-art in AOSD is that one has started to work on the definition of apparently useful metrics. Now it is time to start with completing this research by providing empirical results. This will enable a larger to community to use AOSD metrics and more importantly, understand the benefits of AOSD. Thus, we have to strive for planning and carrying out the corresponding experiments. The results may also give hints as to for which purposes metrics extensions are useful and for which purposes separate metrics are useful.

5. REFERENCES

[1] E. L. A. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. In *1st International Conference on Aspect-Oriented Software Development (AOSD)*, pages pp. 120–126, 2002.

[2] Victor R. Basili. Software Modeling and Measurement: The Goal Question MetricParadigm. In *Computer Science Technical Report Series, CS-TR-2956(UMIACS-TR-92-96), University of Maryland*, 1992.

[3] Coral Calero, Mario Piattini, and Marcela Genero. Method for obtaining correct metrics. In *ICEIS (2)*, pages 779–784, 2001.

[4] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[5] Norman Fenton and Shari Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd edition).* 1997.

[6] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. v. Staa. Modularizing design patterns with aspects: A quantitative study. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages pp. 3–14, 2005.

[7] Mik Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *OOPSLA99*, pages 340–352, 1999.

[8] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman E. Fenton. Towards a framework for software measurement validation. *IEEE Trans. Software Eng.*, 21(12):929–943, 1995.

[9] Christa V. Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, Nov. 1997.

[10] Claudio Sant'Anna, Alessandro Garcia, Christina Chavez, Carlos Lucena, and Arndt von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *XVII Brazilian Symposium on Software Engineering*, 2003.

[11] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *WCRE*, pages 112–121. IEEE Computer Society, 2004.

[12] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Software Eng.*, 14(9):1357–1365, 1988.

[13] A. Zakaria and H. Hosny. Metrics for Aspect-Oriented Software Design. In *Workshop on Aspect-Oriented Modeling AO'03*, 2003.

[14] Jianjun Zhao and Baowen Xu. Measuring aspect cohesion. In Michel Wermelinger and Tiziana Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2004.

Aspect Mining using a Vector-Space Model Based Clustering Approach

Grigoreta Sofia Moldovan
Department of Computer Science
Babeş-Bolyai University
Str. Mihail Kogălniceanu, Nr. 1
Cluj-Napoca, Romania
grigo@cs.ubbcluj.ro

Gabriela Şerban
Department of Computer Science
Babeş-Bolyai University
Str. Mihail Kogălniceanu, Nr. 1
Cluj-Napoca, Romania
gabis@cs.ubbcluj.ro

ABSTRACT

This paper presents a new approach in aspect mining that uses clustering and proposes two techniques: a *k-means* based clustering technique and a *hierarchical agglomerative* based clustering technique. We are trying to identify the methods that have the code scattering symptom. For a method, we consider as indication of code scattering a big number of calling methods and, also, a big number of calling classes. In order to group the best methods (candidates), we use in our approach the vector-space model for defining the similarity between methods. For testing the efficiency of the proposed techniques, a number of Java applications are being used.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; I.5.3 [**Computing Methodologies**]: Pattern Recognition—*Clustering*

Keywords

Aspect Mining, Clustering.

1. INTRODUCTION

The Aspect Oriented Programming(AOP) is a new paradigm that is used to design and implement *crosscutting concerns* [11]. A *crosscutting concern* is a feature of a software system that is spread all over the system, and whose implementation is tangled with other features' implementation. Logging, persistence and connection pooling are well-known examples of crosscutting concerns. In order to design and implement a crosscutting concern, AOP introduces a new modularization unit called *aspect.* At compile time, the aspect is woven to generate the final system, using a special tool called *weaver.* Some of the benefits that the use of AOP to software engineering brings are: better modularization, higher productivity, software systems that are easier to maintain and evolve.

Aspect mining is a relatively new research direction that tries to identify crosscutting concerns in already developed software systems without using AOP. The goal is to identify them and then to refactor them to aspects, to achieve a system that can be easily understood, maintained and modified.

Crosscutting concerns in non AO systems have two symptoms: *code scattering* and *code tangling.* *Code scattering* means that the code that implements a crosscutting concern is spread across the system, and *code tangling* means that the code that implements some concern is mixed with code from other (crosscutting) concerns.

The paper is structured as follows: section 2 presents the main issues related to the clustering problem, section 3 explains our approach, and section 4 presents the applications we have used to test our approach and the results we have obtained. Section 5 presents our conclusions and some future research directions.

1.1 Related Work

Several approaches have been considered for aspect mining until now. One approach was to develop tools that would help the user to navigate and to analyze the source code in order to find crosscutting concerns. Some of them rely on lexical analysis, and some also include a type-based search([6], [8], [20]). Other approach uses clone detection techniques to identify duplicate code, that might indicate the presence of crosscutting concerns([16] [15], [2]). These are all static approaches that analyze the source code for crosscutting concerns. There are also two dynamic approaches: one that analyzes the event traces [1], and one that uses formal concept analysis to analyze the execution traces [18]. In [19] formal concept analysis is used again, but in a static manner. A comparison of three different approaches can be found in [3].

There is also a clustering approach that constructs the clusters based on the methods' names [17]. The user can then navigate among the clusters, visualize the source code of the methods and identify the crosscutting concerns.

2. CLUSTERING

Clustering, or unsupervised classification, is a data mining activity that aims to partition a given set of objects into groups (classes or clusters) such that objects within a cluster would have high similarity to each other and low similarity to objects in other clusters. The inferring process is carried out with respect to a set of relevant characteristics or attributes of the analyzed objects. Similarity and dissimilarity between objects are calculated using metric or semi-metric functions applied to the attribute values characterizing the objects.

Let $X = \{O_1, O_2, \ldots, O_n\}$ be the set of objects to be clustered. Using the vector-space model, each object is measured with respect to a set of m initial attributes $A_1, A_2, \ldots$ $\ldots, A_m$ (a set of relevant characteristics of the analyzed objects) and is therefore described by a m-dimensional vector $O_i = (O_{i1}, \ldots, O_{im}), O_{ik} \in \Re, 1 \leq i \leq n, 1 \leq k \leq m$. Usually, the attributes associated to objects are standardized in order to ensure an equal weight to all of them ([7]).

The measure used for discriminating objects can be any *metric* function d. We used the *Euclidian distance*:

$$d(O_i, O_j) = d_E(O_i, O_j) = \sqrt{\sum_{l=1}^{m} (O_{il} - O_{jl})^2}$$

The *similarity* between two objects O_i and O_j is defined as

$$sim(O_i, O_j) = \frac{1}{d(O_i, O_j)}$$

A large collection of clustering algorithms is available in the literature. [7], [9] and [10] contain comprehensive overviews of the existing techniques. Most clustering algorithms are based on two popular techniques known as *partitional* and *hierarchical* clustering.

In the following, an overview of both techniques is presented.

2.1 Partitioning Methods. The *k-means* Clustering Algorithm

A well-known class of clustering methods is the one of the partitioning methods, with representatives such as the *k-means* algorithm or the *k-medoids* algorithm. Essentially, given a set of n objects and a number $k, k \leq n$, such a method divides the object set into k distinct and non-empty clusters. The partitioning process is iterative and heuristic; it stops when a "good" partitioning is achieved.

Finding a "good" partitioning coincides with optimizing a criterion function defined either locally (on a subset of the objects) or globally (defined over all of the objects, as in *k-means*). These algorithms try to minimize certain criteria (a squared error function); the squared error criterion tends to work well with isolated and compact clusters ([10]).

Partitional clustering algorithms are generally iterative algorithms that converge to local optima.

The most widely used partitional algorithm is the iterative *k-means* approach. The objective function that the *k-means* optimizes is the squared sum error (SSE). The SSE of a partition $K = \{K_1, K_2, \ldots K_p\}$ is defined as:

$$SSE(K) = \sum_{j=1}^{p} \sum_{i=1}^{n_j} d^2(O_i^j, f_j) \qquad (1)$$

where the cluster K_j is a set of objects $\{O_1^j, O_2^j, \ldots, O_{n_j}^j\}$ and f_j is the centroid (mean) of K_j:

$$f_j = \left(\frac{\sum\limits_{k=1}^{n_j} O_{k1}^j}{n_j}, \ldots, \frac{\sum\limits_{k=1}^{n_j} O_{km}^j}{n_j} \right)$$

Hence, the *k-means* algorithm minimizes the intra-cluster distance. The algorithm starts with k initial centroids, then iteratively recalculates the clusters (each object is assigned to the closest cluster - centroid) and their centroids until convergence is achieved.

2.2 Hierarchical Methods. The Hierarchical Agglomerative Clustering Algorithm ($HACA$)

Hierarchical clustering methods represent a major class of clustering techniques. There are two styles of hierarchical clustering algorithms. Given a set of n objects, the agglomerative (bottom-up) methods begin with n singletons (sets with one element), merging them until a single cluster is obtained. At each step, the most similar two clusters are chosen for merging. The divisive (top-down) methods start from one cluster containing all n objects and split it until n clusters are obtained.

The agglomerative clustering algorithms that were proposed in the literature differ in the way the two most similar clusters are calculated and the linkage-metric used (single, complete or average).

3. CLUSTERING APPROACH IN ASPECT MINING

Our approach is to try to discover crosscutting concerns by finding measures of the two symptoms: *code scattering* and *code tangling*. The version presented here is just for *scattering*. Our goal is to group the methods by the number of calling methods (the *fan-in* metric) and also by the number of calling modules (in this case we have considered classes as modules). In [14] is presented an approach to aspect mining that also uses the *fan-in* metric, but in our opinion the number of calling classes is also important. A method might have a high fan-in value, but all the calling methods belong to the same class. This might show a high-coupling between the two classes and it might, even, indicate that some refactoring is needed [4].

In order to group methods, we use two clustering algorithms: the *k-means* algorithm and the Hierarchical Agglomerative Clustering Algorithm $HACA$ (section 2).

In our approach, the objects to be clustered are methods $X = \{M_1, M_2, \ldots M_n\}$. The methods belong to the application classes or are called from the application classes. We have considered two vector-space models:

- The vector associated with the method M is $\{FIV, CC\}$, where FIV is the fan-in value and CC is the number of calling classes. We denote this model by $\mathcal{M}_1$.
- The vector associated with the method M is $\{FIV, B_1, B_2, ...B_s\}$, where FIV is the fan-in value and B_i is the value of the attribute corresponding to the application class C_i $(1 \leq i \leq s)$. The value of B_i is 1, if the method M is called from a method belonging to C_i, and 0, otherwise. We denote this model by $\mathcal{M}_2$.

In the following, we will briefly describe the application of *k-means* and *HACA* in the context of aspect mining.

k-means

We applied a modified version of the *k-means* algorithm in order to optimally divide the set of methods into clusters. We define the "optimal" partition $K = \{K_1, K_2, ...K_p\}$ as the partition that minimizes $SSE(K)$ and we will refer to p as the "optimal" number of clusters $(p \leq k)$.

We also mention that in order to assure a "good" choice of the initial centroids, we choose as initial centroids the most dissimilar initial methods (objects).

We mention that, for simplicity, we will continue to refer this method as *k-means*.

HACA

With the optimal number of clusters p determined after applying *k-means*, we have applied a modified version of the traditional *HACA* algorithm in order to determine p clusters in data (the agglomerative algorithm stops when p clusters are reached).

We also mention that we have used complete-link as a linkage metric, because, in general, complete-link generates compact clusters [10] and is a better choice for our approach (single-link produces elongated clusters).

3.1 Identification steps

The approach consists in the following steps:

Step 1. Computation

Computation of the set of methods in the selected source code, and computation of the attribute set values, for each method in the set.

Step 2. Filtering

Methods belonging to some data structures classes like *ArrayList, Vector* are eliminated. We also eliminate the methods belonging to some built-in classes like *String, StringBuffer, StringBuilder*, etc.

Step 3. Grouping

The remaining set of methods is grouped into clusters using *k-means* or *HACA*. The clusters are sorted by the average distance from the point 0_m in descending order, where 0_m is the m dimensional vector with each component 0.

Step 4. Analysis

The clusters obtained are analyzed to discover which clusters contain methods belonging to crosscutting concerns. We analyze the clusters whose distance from 0_m point is greater than a threshold (eg. two).

The first three steps are done automatically, but the last one must be done manually.

3.2 Example

In the following, we present a small example that shows how methods are grouped in clusters by our *k-means* approach using the model $\mathcal{M}_1$. If we have the classes shown in Table 1, the values of the attribute set are presented in Table 2 and the clusters obtained are shown in Table 3:

```
public class A {
  private L l;
  public A(){l=new L(); methB();}
  public void methA(){ l.meth(); methB();}
  public void methB(){ l.meth();}
}
public class L {
  public L(){}
  public void meth(){}
}
public class B {
  public B(){}
  public void methC(L l){ l.meth();}
  public void methD(A a){a.methA();}
}
```

Table 1: Code example.

Method	FIV	CC
A.A	0	0
A.methA	1	1
A.methB	2	1
B.B	0	0
B.methC	0	0
B.methD	0	0
L.L	1	1
L.meth	3	2

Table 2: Attribute values when $\mathcal{M}_1$ is used.

Cluster	Methods
C1	{ L.meth }
C2	{A.methA, A.methB, L.L }
C3	{ A.A, B.B, B.methC, B.methD }

Table 3: The clusters obtained by *k-means*.

3.3 Characteristics of our clustering approaches

Our clustering approaches have some advantages, reducing the well known main disadvantages of *k-means* and *HACA*:

- we have adapted the traditional *k-means* approach in order to determine the "optimal" number of clusters (*k-means* is repeatedly applied with a different number of initial centroids until a partition with a minimum *SSE* is obtained);
- the *k-means* dependence on the initial centroids is reduced by a good selection of the initial centroids (the most dissimilar initial objects);
- the *HACA* based approach that we have used, instead of merging all the methods in a single cluster, determines a good enough partition into clusters;

- the $HACA$ based approach uses the complete-link as linkage metric, choice that is better for our approach (we are looking for compact clusters in data).

4. CASE STUDIES

For each case study, we have generated the vector-space models $\mathcal{M}_1$ and $\mathcal{M}_2$ as inputs for clustering, and we have applied the two clustering algorithms described in Section 3: *k-means* and $HACA$. For lack of space we will give only the results obtained by applying the *k-means* algorithm and a single vector space model.

4.1 Theatre

The first case study is a web application, called *Theatre*, developed by an undergraduate student as her graduation project. It allows searching for a show, reserving tickets for a show, canceling reserved tickets; it displays the configuration of a showroom and the occupied places. The application was developed using applets, servlets and databases. It has 27 classes (4 applets, 9 servlets, and 14 additional classes), and 336 methods.

The "optimal" number of clusters obtained by *k-means* for the model $\mathcal{M}_1$ is 9 and for the model $\mathcal{M}_2$ is 15. The distribution of methods inside each cluster is shown in Table 4. The first two clusters are identical for all the algorithms, independent of the vector-space model used.

Cluster	C1	C2	C3	C4	C5	C6	C7	C8	C9
Methods	1	1	2	2	11	14	37	206	62

Table 4: No. of methods inside each cluster when $\mathcal{M}_1$ and *k-means* are used.

The first cluster contains one method *PrintStream.println(String)* that was used inside the applet classes to print logging information. The second cluster also contains a method used for logging, *LogWriter.log(...)*, from the servlets classes. The application contains two distinct logging methods, because the applets are not implicitly allowed to write to files, so applets logging information is written to the Java console of the browser.

The next two clusters contain methods used for the construction of the user interface, and constructors for writing to files.

The application uses a connection pool implemented using the *Singleton* design patterns. In all the cases, all the methods belonging to the database connection were grouped into the same cluster; the difference is the index of the cluster they appear in. In one case, they are the only methods contained in the cluster (when $\mathcal{M}_2$ is used as the vector-space model and *k-means* as the clustering algorithm).

4.2 Carla Laffra - Dijkstra's Algorithm

We have also considered as a case study Carla Laffra's implementation of the Dijkstra algorithm [12]. The "optimal" number of clusters obtained by *k-means* for the model $\mathcal{M}_1$ is 7 and for the model $\mathcal{M}_2$ is 5. The distribution of methods in clusters is shown in Table 5.

Cluster	C1	C2	C3	C4	C5
Methods	1	2	20	40	90

Table 5: No. of methods inside each cluster when $\mathcal{M}_2$ and *k-means* are used.

In each case, the methods *Component.repaint()* and *DocText.showline(String)* appear in the first or the second cluster. The method *Component.repaint()* is used each time a new step is executed, or when the algorithm is finished or when a new execution is started with new input data. This method might be considered as part of the crosscutting concern that refreshes the user interface after the execution of different operations. The method *DocText.showline(String)* is used to display guiding information in the TextArea or to display error messages when some preconditions are not met. The last usage may be considered as a crosscutting concern.

In [18] two crosscutting concerns were discovered: locking and unlocking the user graphical interface each time a functionality was executed. The methods used to implement them were not found in the clusters we have analyzed (a better choice for the threshold will, certainly, influence the final results). Another explanation can be the fact that the approach used in [18] was trying to discover *tangled code*, and the current version of our approach is only trying to discover *scattered code*.

4.3 JHotDraw

Our last case study is JHotDraw, version 5.2 which contains 190 classes. The "optimal" number of clusters obtained by *k-means* for the model $\mathcal{M}_1$ is 20 and for the model $\mathcal{M}_2$ is 34. After analyzing the results we have observed that better results were reported by using the model $\mathcal{M}_1$ and $HACA$ algorithm. That is why we briefly present only these results.

Most of the crosscutting concerns discovered in [14] were also discovered by our approach and they were not eliminated during step 4. The first six clusters contain methods like *Point.Point(...)*, *FigureEnumeration.hasMoreElements()* or *Figure.displayBox()* which cannot be considered as crosscutting concern seeds.

The first occurrences of methods belonging to crosscutting concerns in the obtained clusters are as follows: *Observer* in cluster 7, *Policy Enforcement* in cluster 7, *Persistence* in cluster 10, *Composite* in cluster 11 and *Contract Enforcement* in cluster 17.

5. CONCLUSIONS AND FURTHER WORK

We have presented a new clustering approach in aspect mining based on vector-space models. It tries to identify the methods used to implement crosscutting concerns that have the *scattered* symptom. For that we compute the *fan-in* metric of each method that is called inside the application classes, and the number of classes that call this method. The obtained results are divided into clusters using two clustering algorithms: $HACA$ and *k-means*. Some of the obtained clusters are then manually analyzed to determine if they contain methods used to implement crosscutting concerns.

The case studies used to test our techniques have shown that the analyzed clusters contain almost the same methods independently of the clustering algorithm used. Most of the methods belonging to these clusters are used to implement crosscutting concerns. We also mention that the approach proposed in this paper can be used for large applications, but the complexity of the clustering algorithms grows with the number of methods.

Further work can be done in the following directions:

- To discover a set of attributes that can indicate the *tangling* symptom for methods. This set of attributes can be easily integrated into our approach just by modifying the vector-space model used.
- To apply other filtering steps, for example to eliminate the *get/set* type methods, as in [14].
- To use other vector-space models in the clustering approach, and to identify the models that will lead to better results.
- To apply other clustering techniques in the context of aspect mining.
- To use other approaches for clustering that were proposed in the literature (such as variable selection for hierarchical clustering [5], search based clustering [13]).
- To isolate conditions in order to decide the clustering methods and the metric that will lead to better results.
- To apply this approach for other case studies like PetStore and TomCat, as in [14].
- To identify and to explain the reasons for success and failure in our approach.

6. REFERENCES

[1] S. Breu and J. Krinke. Aspect Mining using Event Traces. In *Proceedings of International Conference on Automated Software Engineering*, pages 310–315, 2004.

[2] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *Proceedings International Conference on Software Maintenance(ICSM 2004)*. IEEE Computer Society, 2004.

[3] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. A Qualitative Comparison of Three Aspect Mining Techniques. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 13–22. IEEE Computer Society, 2005.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] E. B. Fowlkes, G. Gnanadesikan, and J. R. Kettering. *Design, Data, and Analysis: By Some Friends of Cuthbert Daniel*. Wiley, New York, NY, 1987.

[6] W. G. Griswold, Y. Kato, and J. J. Yuan. AspectBrowser: Tool Support for Managing Dispersed Aspects. Technical Report CS1999-0640, UCSD, 3 2000.

[7] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[8] J. Hannemann and G. Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Advanced Separation of Concerns Workshop,at the International Conference on Software Engineering (ICSE)*, May 2001.

[9] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, New Jersey, 1998.

[10] A. Jain, M. N. Murty, and P. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[12] C. Laffra. Dijkstra's Shortest Path Algorithm. http://carbon.cudenver.edu/ hgreenbe/courses/dijkstra/ DijkstraApplet.html.

[13] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59. IEEE Computer Society, 1999.

[14] M. Marin, A. van, Deursen, and L. Moonen. Identifying Aspects Using Fan-in Analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*, pages 132–141. IEEE Computer Society, 2004.

[15] O. A. M. Morales. Aspect Mining Using Clone Detection. Master's thesis, Delft University of Technology, The Netherlands, August 2004.

[16] D. Sheperd, , E. Gibson, and L. Pollock. Design and Evaluation of an Automated Apect Mining Tool. In *Proceedings of Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

[17] D. Shepherd and L. Pollock. Interfaces, Aspects, and Views. In *Proceedings of Linking Aspect Technology and Evolution Workshop(LATE 2005)*, March 2005.

[18] P. Tonella and M. Ceccato. Aspect Mining through the Formal Concept Analysis of Execution Traces. In *Proceedings of the IEEE Eleventh Working Conference on Reverse Engineering (WCRE 2004)*, pages 112–121, November 2004.

[19] T. Tourwé and K. Mens. Mining Aspectual Views using Formal Concept Analysis. In *Proc. IEEE International Workshop on Source Code Analysis and Manipulation*, 2004.

[20] C. Zhang, G. Gao, and H. Jacobsen. Multi Visualizer. http://www.eecg.utoronto.ca/ czhang/amtex/.

Using Design Patterns as Indicators of Refactoring Opportunities (to Aspects)

Miguel P. Monteiro
Escola Superior de Tecnologia – Instituto Politécnico de Castelo Branco
Avenida do Empresário 6000-767 Castelo Branco
Portugal
mmonteiro@di.uminho.pt

ABSTRACT

In this position paper, we argue that traditional object-oriented design patterns can be regarded as workarounds for limitations in current programming languages, including crosscutting concerns. Aspect-oriented programming (AOP) is able to modularise crosscutting concerns and overcomes many of the limitations. To illustrate, we mention a few examples. We describe several situations in which patterns are used to cope with the presence of crosscutting concerns. Such solutions are inferior to solutions make possible by AOP and on this basis we hypothesize that patterns can provide clues to improve existing systems by refactoring to aspects. We briefly outline an approach to derive a deeper understanding of how patterns can be used as indicators of refactoring opportunities.

Keywords

Aspect-Oriented Programming, Design Patterns, Refactoring.

1. INTRODUCTION

In this paper, we focus on the problem of identifying refactoring opportunities in object-oriented (OO) legacy systems, in the light of aspect-oriented programming (AOP). Our position is based on the premise that design patterns comprise an important technique used by developers of OO systems to cope with crosscutting. We argue that such efforts would benefit of a more systematic knowledge of the uses that developers make of design patterns in such circumstances. Such knowledge promises to yield useful catalogues of refactoring opportunities, i.e., descriptions of situations in OO code that can be improved using AOP's superior compositional capabilities. For the purposes of this paper, the concept of refactoring opportunity is akin to that of *code smell* as proposed in [6].

This paper is intended to be approached as a true "position paper": we do not present concrete results; rather, we limit ourselves to a general presentation of a hypothesis and propose an approach to assess whether it can be proven. The paper is structured as follows. Section 2 presents our position. To better support it, we present a few illustrative examples in section 3. Section 4 outlines the approach we propose to meet our goals. Section 5 concludes the paper.

2. PATTERNS AS INDICATORS OF REFACTORING OPPORTUNITIES

In the last decade, design patterns became increasingly popular as a way to express design solutions to recurring problems [7]. The 1990s witnessed a veritable industry of "pattern hunting" and as a result we now have a rich repository of object-oriented (OO) patterns. Patterns are currently regarded as an essential component of the skills of any programmer involved in developing, maintaining and evolving object-oriented systems. Patterns are often presented as solutions to attain greater flexibility in a system for a given requirement.

However, it is also possible to view patterns in a more negative light. Patterns are problem-solution pairs [14], meaning that whenever we use a pattern, there is problem that the pattern is supposed to solve, or at least circumvent. In a significant number of cases, the problem stems from limitations in the OO language used. When a feature or composition capability is not directly available in a language, the solution often lies in implementing some pattern that achieves the intended effect, usually at some cost in flexibility and added complexity. Often, patterns are used as workarounds for limitations that theoretically need not exist. In such a light, the existence of such a large variety of patterns seems to suggest that existing OO languages are rather limited.

Let us give some examples: (1) *Abstract Factory* (pages 87-96 of [7]) proposes a way to emulate co-variance, (2) *Factory Method* (pages 107-116 of [7]) describes a way to emulate polymorphic construction of objects (directly supported in languages such as Objective C), (3) *Prototype* (pages 117-126 of [7]) is a way to emulate the prototype-cloning effect found in prototype-based languages such as Self [13] (4) *Decorator* (pages 175-184 of [7]) describes a way to emulate *mixins* [3] and (5) *Visitor* (pages 331-344 of [7]) proposes a way to emulate multiple dispatch. Many other examples could be given, though an extensive list lies out of the scope of this paper.

Despite its limitations, OO is a rich paradigm that sometimes enables multiple variants to achieving a given effect. Many design problems can be addressed by a plethora of different solutions, each one providing its unique set of specific advantages and trade-offs. This richness makes it likely that different programmers working in different contexts may select different solutions to deal with the same problem.

It has been noted that some of the motivations for implementing patterns has its roots in crosscutting, such as those that can be effectively tackled using aspect-oriented languages. Some patterns are known to "disappear" when implemented using AOP, while other patterns witness a significant simplification in their implementations [8]. As with other design problems, we observe a rich variety of different OO design solutions in relation to crosscutting. As a consequence, symptoms of the presence of crosscutting concerns in OO legacy systems can take many different forms and patterns. In the next section, we mention a few testimonies that can be found in the literature. We believe that patterns may offer a rich set of clues of when to refactor well-formed OO systems to aspects.

Why do we focus on patterns? In the context of this paper, we're referring to well-formed OO code, developed by experienced and knowledgeable programmers. Such programmers are more likely to use patterns well (in an OO sense) than novice programmers and be aware that "duplication is evil" [6]. Therefore, they take great effort to remove such duplication, by keeping their code clean through refactoring. It seems reasonable to assume that many such programmers resort to patterns to deal with such issues, both when designing [7] and when refactoring [10].

On the other hand, the compositional capabilities of OO – currently the dominant programming paradigm – do not seem to be sufficient to cope with all demands of modern software, namely crosscutting. There seems to be a conflict between the stated aims of the test-driven and refactoring communities and what can be achieved with the programming paradigm that most people from those communities (currently) use. Most testimonies from these communities suggest that all forms of duplicated code can be eliminated from OO code. And yet, this claim is likely to raise eyebrows from among the AOP community, given the close link between crosscutting and duplication. Note that crosscutting is not generally mentioned in [7][10] as a reason to use patterns.

In our view, what explains this apparent conflict is the use of elaborate design structure, namely patterns, to *mask* the symptoms of duplication. Developers that "mercilessly refactor" a given OO code base until all manifestations of duplication are removed, are really trading one problem with another: they merely remove the *semblance* of duplication, replacing it with increased structural complexity and inflexibility. There is a risk that the refactored structure proves to be almost as hard to evolve and reason with as the original one. By contrast, AOP promises to provide developers with more acceptable trade-offs. That is the claim suggested by Isberg in [9], which analyses the structure of the JUnit framework [2]. Isberg discusses trade-offs of the current design decisions for JUnit and proposes re-implementations of some parts of the framework using pointcuts and advice, pointing out that most of what he proposes to re-implement is, to current thinking, well-modularised.

3. A FEW ILLUSTRATIVE EXAMPLES

In this section, we describe a few examples of the use of OO design patterns that are used to cope with crosscutting. We suggest hypothetical refactorings that yield better AOP alternatives. Throughout the descriptions, we assume the reader has a general knowledge of the Gang-of-Four patterns [7] and refrain from providing descriptions of the patterns.

3.1 Decorator

In [5], Feathers describes various techniques to deal with cases in which additional logic must be added to the core logic of a system. Feathers mentions as a typical case a situation in which the new logic to be added happens to execute at the same time as the one in a method, giving raise to temptation to place it in the same method. However, the new logic is otherwise unrelated and there is a chance that in future someone will want to use one without the other. Feathers mentions logging as an example of such an additional logic. People familiar to AOP recognise the favourite example of crosscutting (though in recent times it has been challenged by the *Observer* pattern). To address this problem, Feathers proposes a few techniques that include *Wrap Method* (pages 67-70 of [5]) and *Wrap Class* (pages 71-76 of [5]). There are a few variants to implementing *Wrap Method*, but it basically entails wrapping the method with the original logic with a new method that simply performs the additional logic (before or after, depending on the specific problem) and forwards it to the old method. *Wrap Class* is, by Feathers' own admission, really an instance of *Decorator*.

The two above techniques suggest two new AOP refactorings: *Replace Wrap Method with Pointcut and Advice* and *Replace Decorator with Aspect*. We envision *Replace Wrap Method with Pointcut and Advice* as creating a pointcut that captures the points in the execution of the program where the wrap method is called and adding an advice acting on those joinpoints that provides the logic formerly provided by the wrap method. Next, the wrap method can be removed. *Replace Decorator with Aspect* is about creating an aspect that captures joinpoints where the behaviour that the decorator decorates is called, and placing in the aspect an advice acting on those joinpoints that provides the logic of the decorator. The decorator class can probably be removed afterwards. In the simplest cases, using an AOP implementation of *Decorator* proposed in [8] may be sufficient.

3.2 Template Method

Template Method (pages 325-330 of [7]) looks very promising as a signal of refactoring opportunities. It comprises one of the design backbones of many OO frameworks and frequently features in APIs. Classes `java.applet.Applet` and `java.lang.Thread` from Java's API include widely-known examples of *Template Method*.

It is possible to view *Template Method* as a crude technique to emulate pointcuts and advice. The template method performs a role that bears some similarities to pointcuts in that it serves to control the moments when some desired logic executes. The concrete classes that override and concretise the hooks exposed by the template method perform a role similar to that of advice: in both cases, the blocks of code execute reactively, or implicitly. This suggests a *Replace Template Method with Pontcut and Advice* refactoring.

3.3 Singleton

The *Singleton* pattern (pages 127-134 of [7]) is one of the patterns that attracted most criticisms. There are many testimonies of the excessive use of singletons, some of which can be found in the refactoring mailing list at Yahoo[1]. Overuse of singletons is troublesome, as it scatters multiple dependency points to the singleton throughout the system. Singletons also create specific problems when creating unit tests for classes that depend on them [12][5]. The pattern is considered prone to misuse, often by programmers that have yet to fully absorb the fundamental principles of OO and that lean on singletons to write "procedural-style OO code". Such programmers tend to create too many singletons that are really procedural-style global variables.

[1] http://groups.yahoo.com/group/refactoring/

Such bad uses of *Singleton* can be addressed in a number of ways and a few of them are suggested in [10][5]. However, in cases, turning the singleton into an aspect may be the appropriate solution. Aspects can have global access to the remaining elements of the system, but can also compose behaviour in a controlled way. Aspects can compose the behaviour equivalent to that provided by the singleton in an *implicit* way, thus avoiding the kind of dependencies that result from scattered calls to singleton logic. This suggests a *Replace Singleton with Aspect* refactoring. Such a refactoring entails moving to an aspect the singleton logic that is called from multiple places and ensuring that the aspect is able to capture all those points. The logic provided by the singleton is next moved to advices within the aspect that act on those joinpoints.

4. PROPOSED APPROACH

In order to build a catalogue of refactoring opportunities such as the ones mentioned above, we propose an approach similar to the one we took in [11]: use existing object-oriented systems as case studies to derive insights. We plan to analyse the source code of selected systems to assess whether there is a link between crosscutting concerns and the use of design patterns. Refactoring experiments should also be performed, such as those suggested in [9]. Ideal case studies are pattern-rich code bases, e.g. frameworks. Open-source OO systems such as JUnit [2] and JHotDraw [1] comprise good candidates and have the advantage of some previous work being already available [9][4].

Any refactorings derived from such a study must address various issues, including the one that follow:

- What are the preconditions for the refactorings? Are there any special situations that prevent its use or do not make it advisable to apply it? For instance, it is likely that some instances of *Singleton* remain desirable: documentation should be derived that clearly states when the refactoring is applicable.
- What are the detailed mechanics of the refactorings? Do the refactorings require the use of other, preparatory, refactorings?
- What impact do the refactorings have on the remaining code base? Can some designs be simplified? Can the extraction of a crosscutting concern remove the motivation to use a pattern in some situations?

5. CONCLUSION

In this paper, we argue that in order to identify refactoring opportunities to aspects, we need to go beyond the more superficial manifestations of crosscutting, such as duplicated (and scattered) code. The efforts of well-meaning and experienced programmers and designers may mask such superficial manifestations behind less obvious ones such as elaborate structures and use of design patterns. We argue that patterns comprise a primary candidate to build a deeper base of knowledge for identifying opportunities to evolve legacy systems using AOP. To illustrate, we describe uses of a few patterns whose motivation stems from either the presence of crosscutting effects or limitations of OO relative to AOP. We suggest a few refactorings that address the same problems more effectively.

6. ACKNOWLEDGMENTS

Miguel P. Monteiro is partially supported by project PPC-VM (POSI/CHS/47158/2002) and by FCT under project SOFTAS (POSI/EIA/60189/ 2004).

7. REFERENCES

[1] JHotDraw home page. http://www.jhotdraw.org/

[2] JUnit home page. http://www.junit.org/

[3] Bracha G., Cook W., *Mixin-Based Inheritance*, ECOOP/OOPSLA 1990, Ottawa, Canada, October 1990.

[4] van Deursen, A., Marin, M., Moonen, L., *AJHotDraw: A showcase for refactoring to aspects*. LATE 2006 workshop at AOSD 2005, Chicago, USA, March 2005.

[5] Feathers, M., Working Effectively with Legacy Code, Prentice Hall 2005.

[6] Fowler, M. (with contributions by K. Beck, W. Opdyke and D. Roberts), *Refactoring – Improving the Design of Existing Code*, Addison Wesley 2000.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[8] Hannemann, J., Kiczales, G., *Design Pattern Implementation in Java and AspectJ*, OOPSLA 2002, Seattle, USA, November 2002.

[9] Isberg, W., *Design with pointcuts to avoid pattern density*, online article at Developerworks (AOP@Work series), June 2005. www-128.ibm.com/developerworks/java/library/ j-aopwork7/index.html

[10] Kerievsky, J., *Refactoring to Patterns*, Addison-Wesley, 2004.

[11] Monteiro, M. P., Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts. Ph.D. thesis, Universidade do Minho, Portugal, March 2005.

[12] Rainsberger, J., *Use your Singletons Wisely*, online article at Developerworks, July 2001. www-128.ibm.com/ developerworks/webservices/library/co-single.html

[13] Ungar D., Smith, R., *Self:the power of simplicity*, OOPSLA'87, Orlando, USA, October 1987.

[14] Venners, Patterns and Practice – A Conversation with Erich Gamma, Part IV, Artima developer, June 2005. www.artima.com/lejava/articles/patterns_practice.html

Concern Highlight: A Tool for Concern Exploration and Visualization

Eugen C. Nistor [*] André van der Hoek

Department of Informatics
School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
E-mail: {enistor,andre}@ics.uci.edu

Abstract

Separation of concerns is a powerful principle that can be used to manage the inherent complexity of software. One of the benefits of separation of concerns is an increased understanding of how an application works, which helps during evolution. This benefit comes from the fact that the code belonging to a concern can be seen and reasoned about in isolation from the other concerns with which it is tangled together. In this paper, we present our tool that gives the developer the freedom to annotate existing code based on different concerns. These concerns can be later analyzed or visualized as highlighted text. Although our experiences were based on the special case of high-performance computing programs, we believe that the observations are pertinent to general programming as well.

1 Introduction

A concern denotes anything of importance while developing software. Naturally, every piece of software will have different concerns tangled together, and this complexity makes not only software construction difficult, but makes long term evolution and maintenance difficult as well.

Separation of concerns is an important principle in software engineering. The idea behind separation of concerns is that one should be able to reason about a single concern separately from the other concerns tangled in the final program. Techniques such as Aspect Oriented Programming (AOP) [1] offer the mechanisms through which concerns can be written in separate modules as aspects, and then weaved into an existing application at specified join points. However, the benefits of being able to identify concerns in code, and to use these concerns to explore the code should not be limited to programs following AOP. In some cases it might not always be possible to identify and write concerns separately from the start, like in the case of exploring projects for which source code cannot be modified. The same is true for situation where identification of concerns will lead to a future refactoring where some of the concerns will be written as aspects. Furthermore, even in programs written using AOP, different concerns might crosscut the code inside one particular aspect, and a tool like the one presented here might be useful in visualizing overlapping concerns.

In this paper we present the Concern Highlight, our tool for concern exploration and visualization. The tool is built as an extension to the Concern Manipulation Environment (CME), an aspect-oriented software development environment [2], and adds support for automated and free selection of source code snippets that belong to different concerns. These concerns can then be visualized as source code highlights in an editor in Eclipse [3].

The development of the tool came from our experiences of trying to apply concern-based development and evolution to the specific domain of high-performance programs written using the Message Passing Interface (MPI) library [4]. This is an interesting field because it exhibits the particularities described above that make traditional aspect-oriented approaches difficult or impossible to apply. In the rest of this paper, we discuss the specifics of MPI programming related to aspect-orientation, and present our Concern Highlight tool. Although our work is exploratory, we believe that our observations can be suc-

[*]The work described in this paper was done while at the IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA

cessfully used to help concern-based development and evolution in regular programming as well.

2 High-performance Computing with MPI

The initial motivation for the project was to bring the benefits of identifying concerns and using the CME to programs written in high-performance computing. Scientific programming in general – and high-performance programs written for parallel processor environments in particular – have a number of particularities: they are in their majority written in a procedural language such as C and Fortran, they are difficult to change because of their complexity, and the information content tends to be very dense because of the algorithms embedded into code.

Message Passing Interface (MPI) is a standard for high-performance computing libraries that provides a common interface for programming in a multiprocessor environment. The power of MPI comes from the fact that a complex middleware functionality is hidden from the programmer behind a set of well defined method calls, and the parts of the program that are executed in parallel coexist with the code that will be executed only on one processor. Specifying MPI as a standard assures separation from specific hardware implementations and interoperability across different MPI implementation libraries.

Although the complete MPI specification contains a significant number of methods, a few of them are very important and used extensively. However, while using these functions, the developer needs to be aware of special requirements and assumptions – which we can consider under the realm of MPI domain knowledge – that if not respected, can lead to errors. For example, the code that will be executed in parallel is separated by two special method calls, *MPI_Init* and *MPI_Finalize*, that delineate the start and the end of the parallel code, and all the other MPI calls have to occur in between these two. Methods *MPI_Send* and *MPI_Recv* are responsible for sending and receiving data needed for communication between processes. Their parameters determine where the data is sent to or coming from, and the process where these methods are executed is blocked until the methods calls are complete. Other methods include support for asynchronous communication, synchronization, I/O, timing and logging [4].

3 The Concern Highlight Tool

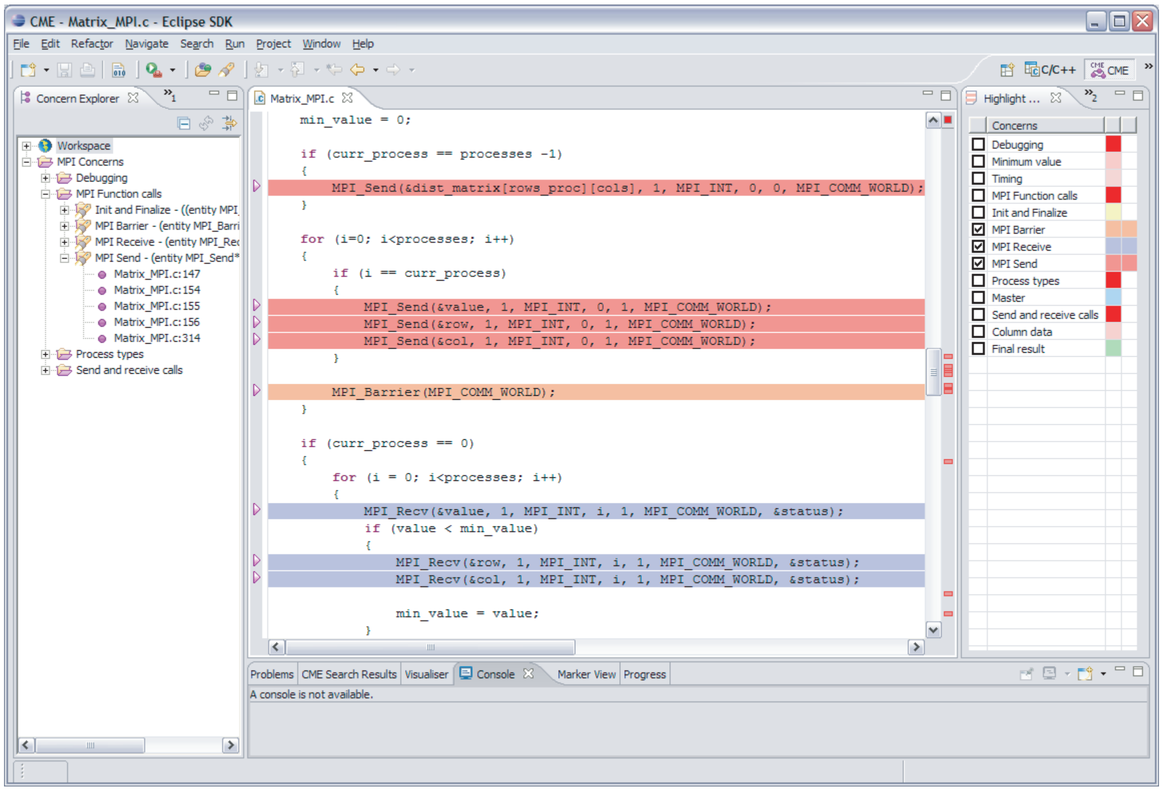
CME contains a number of tools that support concern-based software exploration, implemented as Eclipse plug-ins [5, 3]. One of such tools is the Concern Explorer, which presents a hierarchical model of different concerns identified in the software. These concerns are either defined by the developer, or loaded from specialized concern loaders from source code and other development artifacts, such as build files or design documents. A powerful query evaluator can be used in the Query View to evaluate queries over the concern model. These queries not only help browsing the concern model, but can also be saved in the concern model as new concerns.

The Concern Highlight tool is an Eclipse Plug-in that can be used to mark up source code ranges belonging to different concerns. The highlight tool functions as a complex code annotation system, but its integration with CME allows these concerns to be browsed or queried afterwards in the CME Query Analyzer. Figure 1 shows the Concern Highlight tool in a typical usage in Eclipse. On the left-side, CME's Concern Explorer displays the concern model. On the right-side, the Highlight List contains a list of concerns that the user is interested in exploring, populated with concerns selected from the Concern Explorer. The editor is shown in the middle, with the code belonging to the selected concerns highlighted in the text.

The user can associate source code snippets with concerns in the highlight list in two ways: either automatically, by using the CME's API, or manually by marking up source code in the current editor opened in Eclipse, through a context menu option. Automatic detection of source code related to a concern is desirable, but at the same time is problematic since it is highly domain dependent and depends a lot on the type of concern sought.

The format of MPI programs makes a number of concerns to be suitable for being marked up automatically in code, but in general we found three distinct types of concerns:

- Concerns that can easily be identified automatically. A simple example is marking the beginning and ending of MPI-related code, which corresponds with the *MPI_Init* and *MPI_Finalize* function calls.
- Concerns where automatic identification is possible, but manual marking would be much simpler. A good example is to try to find out usual MPI code patterns like the one shown in Figure 2. The code shown is a common MPI pattern, where one of the processes, commonly called the *master process*, has extra responsibilities than the other processes, usually related to input and output or distribution and gathering of data. Automatic detection of this pattern is fairly complex, while the developer could have marked the code as belonging to the concern while writing the code.


CME - Matrix_MPI.c - Eclipse SDK
File Edit Refactor Navigate Search Run Project Window Help
C/C++ CME
Concern Explorer
Matrix_MPI.c
Highlight ...
Problems CME Search Results Visualiser Console Marker View Progress

```
Workspace
MPI Concerns
  Debugging
  MPI Function calls
    Init and Finalize - ((entity MPI_
    MPI Barrier - (entity MPI_Barri
    MPI Receive - (entity MPI_Rec
    MPI Send - (entity MPI_Send*
      Matrix_MPI.c:147
      Matrix_MPI.c:154
      Matrix_MPI.c:155
      Matrix_MPI.c:156
      Matrix_MPI.c:314
  Process types
  Send and receive calls
```

```
min_value = 0;

if (curr_process == processes -1)
{
    MPI_Send(&dist_matrix[rows_proc][cols], 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

for (i=0; i<processes; i++)
{
    if (i == curr_process)
    {
        MPI_Send(&value, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        MPI_Send(&row, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        MPI_Send(&col, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }

    MPI_Barrier(MPI_COMM_WORLD);
}

if (curr_process == 0)
{
    for (i = 0; i<processes; i++)
    {
        MPI_Recv(&value, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
        if (value < min_value)
        {
            MPI_Recv(&row, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
            MPI_Recv(&col, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);

            min_value = value;
        }
```

	Concerns		
☐	Debugging	🟥	
☐	Minimum value	🟥	
☐	Timing	🟥	
☐	MPI Function calls	🟥	
☐	Init and Finalize	🟨	
☑	MPI Barrier	🟧	🟧
☑	MPI Receive	🟦	🟦
☑	MPI Send	🟥	🟥
☐	Process types	🟥	
☐	Master	🟦	
☐	Send and receive calls	🟥	
☐	Column data	🟥	
☐	Final result	🟩	

A console is not available.

Figure 1. A screenshot of the Concern Highlight tool in Eclipse.

```
int myrank;
...
MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
...
if ( myrank == 0 ){

      /*code that is only executed on the
        first process */

}
```

Figure 2. A typical MPI pattern.

- Some concerns, such as the name of the user that performed a certain change in code, or trying to determine the code that was changed when transforming a program from a non-parallel to a parallel version, might be either impossible to detect automatically, would have to rely on comments written consistently throughout the code, or be imported from some other artifacts such as configuration management logs. In this case, human identification of concerns seems like the best solution.

Manual marking of the source code is more suitable for scenarios where the user wants to understand an existing application's implementation. Many times the source code for the libraries one uses in a project forms the most complete documentation at hand. The task of finding out how the existing application works is driven by a type of concern, which is the problem that needs to be solved. Marking up this concern will prove beneficial in documenting the code in such a way that if somebody else will be interested in the same concern to be able to focus on only those parts of the code that are relevant. For example, somebody might be interested in only looking at the code pertinent to the *master process*. In order to do so, they would select the code that looks like the pattern in Figure 2, either manually or with the use of a tool, and mark them as being a part of the *master process* concern. The integration with CME adds the benefit of persistence, such that if somebody new to a project wants to understand the code based on concerns, can use the Highlight View later to select this concern and to quickly see where that source code is located.

One of the important features of our tool is the visualization of concerns by highlighting the text of the source code ranges associated with it in the current editor in Eclipse. The concerns in the highlight list have an associated color, and the user can choose, by selecting a check-box near each concern, which ones should be highlighted. Since the concerns have a hierarchical nature, selecting a parent concern will automatically select all of the other concerns that are its sub-concerns. In this case, all the source code ranges associated with the sub-concerns will be highlighted using the color of the parent concern, unless they are explicitly selected themselves in which case they will be displayed with their own color. Figure 1 shows three different concerns selected and the corresponding source code snippets highlighted in the editor. The highlight feature is implemented on top of Eclipse's support for annotations, and currently most text-based editors from Eclipse are supported.

4 Observations

The main purpose of our tool was to enhance software understanding through recording code exploration traces and associating them with concerns. The main benefits would be seen during software evolution stages, since the code can be browsed on a concern-basis rather than the file or class-based organization. Moreover, if some part of the code needs to be changed, other parts of the code associated with it through a concern will probably hint the developer where to be careful about the effects of the change.

Besides helping understanding and existing program, we believe that concern-based development can potentially avoid common errors. MPI programming includes a number of domain rules that are either documented in books [6, 7], or are learned with experience. Some of the typical errors can be found with the help of either a static or dynamic analyzer. However, we believe that using concern modeling can help avoid some of these errors and offer significant advantages during software evolution.

Probably one of the most compelling arguments in this regard is based on a simple observation related to a typical development scenario that disregards concerns. While writing the software, the developer's intentions, which are naturally related to a concern, are translated into programming. This information is encoded in the specifics of the programming language used, in the libraries used, and possibly in comments. In this way, the concern information is lost as an explicit description. However, after the program is written, special analyzers will try to determine the presence of errors. While errors related to the syntax of the program can be easily found by a compiler, the more complex errors are based on domain knowledge. Typically, these errors can be found by trying to infer the concerns back from the code, and to figure out if a domain-based pattern was broken. Having concerns marked out and preserved together with the program being developed has the potential of avoiding some of these errors.

In the current state, our tool can help the developer in detecting errors by perusing the code related to a concern. As an example related to the scenario above, a simple rule in MPI is that *MPI_Send* calls have to be matched by corresponding *MPI_Receive* calls. When developers write the code, they know exactly which *MPI_Send* is matched by which *MPI_Receive*. However, this information is then lost as explicit information, and only encoded in the parameters given to these method calls. Later, static and dynamic analyzers will try to detect errors due to wrong values for parameters, and in order to do so they will try to guess back these correspondences. Depending on how the program was written, it might not be possible to achieve this statically since the values of the parameters have to be evaluated. Figure 1 shows such an example, with the editor showing a piece of code with four *MPI_Send* calls and three *MPI_Receive* calls. Only the three *MPI_Send* calls grouped together correspond to the *MPI_Receive* calls. This is a type of information that can save important resources consumed by a dynamic analyzer that can induce an important overhead [8]. Moreover, just by looking at the code, the fact that the *MPI_Send* calls are grouped together, and the corresponding *MPI_Receive* calls are not, can give the developer a hint: possibly not all messages that are sent are also received. A quick look at the code reveals that indeed, some of the messages are not received unless a condition is fulfilled, and this can lead to memory leaks and failure.

Although the initial use of the tool was on procedural programs, we believe that its benefits will be maintained in both regular object-oriented programs as well as programs written using aspects. Aspect-oriented programming offers the mechanisms to write the code that belongs to a concern in a separate module. However, concerns can be cross-cutting each other in such a way that the same line of code belongs to multiple concerns. An example from our experience with MPI is having the the same *MPI_Send* call belonging to a master program, being linked to some other *MPI_Receive*, and dealing with some special variables important in the algorithm. Isolating such lines of code in separate aspect modules just based on one of those concerns, using AOP for instance, still leaves the problem of having the other concerns overlapping. Therefore, even in programs where some aspects are written in separate modules, our tool would still be useful in visualizing how the different concerns overlap in the source code.

5 Related Work

Both CME and the AspectJ Development Tools projects [2, 9] include a source code visualizer, originating from SeeSoft [10]. Our tool is similar to them in the fact that it shows the position of the occurrence of each concern in text, and identifies each concern with a different color. However, our tool displays this information in the editor for each individual file, with the user selecting which concerns to be displayed, while the other visualizers show a global view of the code more suitable for a statistical overview of an entire project.

FEAT [11] is a similar tool in that it is offering a mapping between source code text and different features (or concerns). However, FEAT purposely moves away from free source-code mark-up text and stores relationships between different language artifacts, such as method calls traces, as concern graphs [12]. We preferred a free annotation solution, where the user is selecting any part of source code that they think is necessary. FEAT is also implemented as an Eclipse plug-in, but we could not use it in our projects since it works on Java implementations, while our MPI programs were implemented in C or C++.

The Aspect Mining Tool (AMT) [13] presents a discussion of text-based and type-based analysis as tools for discovering hidden concerns in code, with a visualization tool also similar to SeeSoft. We found that since MPI programming is much more rigorous than general programming, text-based and type-based mining is easier in finding MPI-related concerns. However, the AMT tool could provide useful insights for evaluating the possible use of the Concern Highlighter in general object-oriented programming.

6 Conclusion

In this paper we presented our experiences with developing a tool for concern-based exploration. Although initially developed for MPI programming, for which we have discussed its potential benefits and possible uses, we believe that the Concern Highlight tool would prove helpful for identification and visualization of concerns in any type of programs, even the ones written using AOP. The identification of concerns through our tool, together with the integration with CME, can increase software understanding and ultimately help with software evolution.

Our work was exploratory, and a better evaluation of its usefulness can only come from its use in real-life projects. In its current form, our tool can be used to determine what kinds of concerns are more likely to be identified, and what are the possibilities of automation in concern detection, for both MPI programs and in general programming.

The Concern Highlight tool is integrated within CME and is available for download from http://www.eclipse.org/cme.

7 Acknowledgements

The authors would like to thank Harold Ossher, Stan Sutton and William Chung from the IBM T.J. Watson Research Lab for their valuable input and guidance throughout the project.

This research was supported in part by the Defense Advanced Research Projects Agency under grant NBCHC020056.

References

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming. (1997)

[2] Concern Manipulation Environment Eclipse Technology Project. (http://www.eclipse.org/cme)

[3] Eclipse. (http://www.eclipse.org)

[4] The Message Passing Interface (MPI) Standard. (http://www-unix.mcs.anl.gov/mpi/index.htm)

[5] Harrison, W., Ossher, H., S.M. Sutton, J., Tarr, P.: Concern modeling in the concern manipulation environment, IBM Research Report RC23344 (September 2004)

[6] Snir, M., Otto, S.W., Walker, D.W., Dongarra, J., Huss-Lederman, S.: MPI: The Complete Reference. MIT Press (1995)

[7] Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press (1999)

[8] Vetter, J.S., de Supinski, B.R.: Dynamic software testing of mpi applications with umpire. In: Supercomputing '00, Washington, DC, USA, IEEE Computer Society (2000) 51

[9] AspectJ Development Tools Eclipse Technology Project. (http://www.eclipse.org/ajdt)

[10] Eick, S., Steffen, J., E.E. Sumner, J.: Seesoft - a tool for visualizing line oriented software statistics. In: IEEE Transactions on Software Engineering. (1992) 957–968

[11] Robillard, M.P., Murphy, G.C.: Feat: a tool for locating, describing, and analyzing concerns in source code. In: Proceedings of the Proc. Int. Conf. on Software Engineering (ICSE). (2003)

[12] Robillard, M.P., Murphy, G.C.: Concern graphs: Finding and describing concerns. In: Proceedings of the Proc. Int. Conf. on Software Engineering (ICSE). (2002)

[13] Hannemann, J., Kiczales, G.: Overcoming the prevalent decomposition in legacy code. In: Workshop on Advanced Separation of Concerns (ICSE). (1991)

VEJAL: An Aspect Language for Versioned Type Evolution in Object Databases

Awais Rashid, Nicholas Leidenfrost
Computing Department
Infolab21, Lancaster University
Lancaster LA1 4WA
+44-1524-510316
awais@comp.lancs.ac.uk

ABSTRACT

In this paper, we present our aspect language Vejal (Versioned Java Language) which superimposes support for versioned types and AOP onto Java. Vejal has been developed to service the needs of AspOEv[1], an evolution framework that supports dynamic adaptation of evolution strategies in object databases [8, 9]. In this paper, we describe how the framework benefits from the advanced type semantics and the capacity for dynamic manipulation of Java offered by Vejal and its interpreter. We discuss how these capabilities allow programmers to alter types dynamically and undertake reflective analysis and consequent reflective action to preserve structural and behavioural consistency. We also highlight the dynamic AOP capabilities of Vejal and its command-line environment (CLE) which allows programmers to define expressive, single-use evolution primitives.

Keywords

Aspect-oriented software development, aspect-oriented programming, object database evolution, schema evolution, instance adaptation, versioned types, aspect-oriented databases.

1. INTRODUCTION

Determining all requirements for the future use of a software system may be impossible to do during the design and implementation stages of the software life cycle. Therefore, despite modern software engineering practices and attempts to create extensible, reusable, and flexible software, maintenance continues to be the largest phase in the life of any application. As databases serve multiple applications, the extent of evolution in databases, as demonstrated in [10], is beyond that of any single system. Object databases in particular are subject to advanced evolution needs due to the inherent structure of their data. True to the OO paradigm, persistent objects in an object database define their own behaviour, in contrast to a relational entity, which merely provides raw data for external applications to interpret. Therefore evolutionary changes can have complex implications for the behaviour of both existing instances and applications. Furthermore, databases commonly serve as a central storage point for many distributed applications, each of which may evolve independently and have different requirements for the shared data. The use of persistent data in such a heterogeneous environment further compounds the effects of changes. Newer applications may wish to add functionality to a type[2], or may deem functionality which is used by older applications as unnecessary. Compatibility issues arise as applications must operate on persistent data created with diverging definitions of a type.

Furthermore, compound aggregation in object data means that types may be interdependent, and therefore, a change in one type may make another potentially unstable. Altering a type can, therefore, have non-localised effects. Hence evolution in object databases must account for the effect of a change on the entire organisation of types (the database *schema*) and their instances within the database.

The process of evolution in an object database raises a number of major concerns, two of the most prominent being *schema evolution* and *instance adaptation*. Schema evolution pertains to the management of active types within the database environment and their respective evolution, while instance adaptation pertains to the meaningful conversion of existing data from one version of a type to another version.

Object databases usually provide an integrated (fixed) approach to address schema evolution and instance adaptation concerns. This supplied approach, however, may not be the best fit for the application programmer, or the application base that will end up using the database. In addition, it may be impossible to preordain schema evolution and instance adaptation needs when choosing which object database system a project will use.

Our solution to this problem is AspOEv, an aspect-oriented framework which supports dynamic adaptability of schema evolution and instance adaptation strategies in object databases. In such a flexible environment, which permits the simultaneous existence of multiple versions of a type, applications must have the ability to distinguish between versions and allow interaction between instances of different versions. This requires special language features on two levels: first, a language must support the syntactic expression of an explicit version wherever a type is referenced, i.e., allowing clear indication of which version of the type is intended, and second, the type system governing the language must permit, to some extent, the interoperability of instances of different versions of the same type.

Our aspect language, Vejal, and its interpreter have been specifically developed to service the above needs – the framework derives two key benefits from them. Firstly, the advanced type semantics and the capacity for the dynamic manipulation of Java facilitate dynamic evolution of types and adaptation of evolution

[1] The framework can be downloaded from: http://www.comp.lancs.ac.uk/computing/aose/AspOEv.php

[2] In an object database altering the representation of a complex data entity equates to altering the data's *type definition* (henceforth referred to as *type*).

strategies within the framework. Secondly, because Vejal applications are interpreted, the framework is able to capture evolution related events that occur within their execution, including detecting and handling behavioural inconsistencies. Inconsistencies in the execution of Vejal applications arise within the domain of the interpreter, enabling the framework to handle them in a uniform fashion.

The rest of this paper is structured as follows. Section 2 introduces the requirements that the Vejal type system has to cater for. Section 3 introduces the reflective features of Vejal and their use to maintain structural and behaviour consistency upon evolution. Section 4 discusses how join points and advice are handled in Vejal and describes some of its dynamic AOP capabilities. Section 5 discusses how Vejal and its CLE facilitate specification of specialised, one-off ad hoc evolution primitives by the programmers. Section 6 concludes the paper.

2. VERSIONED TYPE REQUIREMENTS FOR THE VEJAL TYPE SYSTEM

Vejal yields many advantages and features that aid AspOEv's flexibility in supporting adaptable evolution strategies. Applications written in Vejal have the ability to specify an explicit version of a type. The grammar for Vejal attempts to emulate that of Java, however, Vejal allows the additional expression of a '<' / '>' delineated version number where Java would normally only expect a class name:

```
Class Student<1.0> extends Person<1.3> {...}
```

Vejal allows the explicit declaration of a version wherever a type may be referenced, including type definitions, variable declarations, method return types, and method parameter types:

```
Person<1.3> frank = new Person<1.3>(...);
```

Applications also have the option to omit the explicit declaration of a version in type declarations:

```
Person frank = new Person(...);
```

If the application does not specify which version a variable should be treated as, the interpreter binds the variable to the most recent version. Object database schema evolution strategies in AspOEv can uniformly advise various nodes within the interpreter's abstract syntax tree (e.g., variable declaration, constructor invocation) to override this functionality, if desired

```
Person <3.0> {                 Person <4.0> {
    String firstname;              String firstname;
    String lastname;               String lastname;
    double mass;                   double mass;
}                                  String mobileNo;
                               }
```

```
Person<4.0> bob = new Person<4.0> ( … );
. . .
Person<3.0> manager = bob;
```

Fig. 1: Assignment of an additive version

In addition to the mere expression of versioned-type semantics, the Vejal type system also allows versions of the same type to be assigned interchangeably, hence providing a database evolution environment that is version polymorphic [9]. Static type safety guarantees are traded for flexibility as most type checking is deferred to runtime, allowing custom strategies to more explicitly define the semantics of type equality. For example, an approach may allow the unconditional assignment of two different versions of the same type if the type of the right-hand value is additive with respect to the type of the left-hand variable. Such an assignment could occur in a class versioning scenario, as shown in Fig. 1, or could happen as the result of a query on the database returning instances of various versions. Regardless of the cause, however, because the properties and operations defined by the right-hand type are a superset of those of the left-hand type, it can be safely assumed that the declared variable will not break behavioural consistency at some later stage.

3. VEJAL META-DATA

Meta data refers generally to objects which represent elements of the executing program itself, allowing the program to take action to alter its behaviour dynamically. Meta objects are the first class representations of program structures in OOP – meta classes model class definitions, meta fields and meta methods model the declarations of properties and operations respectively within meta classes, etc. Meta classes typically maintain inheritance information, as well as any meta fields and meta methods defined by the represented class. Likewise, meta fields typically store the modifiers, type, and name of the field, and meta methods the method signature, including modifiers, return type, name, and parameter types.

Like many schema evolution approaches able to achieve evolution dynamically, e.g., [1, 7], AspOEv relies heavily on a meta-layer to represent the structure of the persistent data. The relationships between versions of a type adds an extra dimension to Vejal meta classes – in addition to storing a type's inheritance graph (which is directed, acyclic), meta classes also maintain links to derived and base versions. This is important as instance adaptation mechanisms often use the derivation path between two versions to determine how conversion should occur upon database evolution. To ensure stability among changing type definitions, the AspOEv framework stores meta-data persistently. The Vejal interpreter loads all type definitions from the database at start-up, preventing duplicate definitions of a type. This ensures that instances cannot have their original representations changed; once a type definition is finalised, it cannot change.

In addition to providing a cohesive view of the types in the schema, Vejal's meta-classes and their relationships allow database programmers to alter types (and hence, the schema) dynamically. Additionally, execution-level meta-data, i.e., a meta representation of procedural (method-body) code, enables the reflective analysis of the impact of a change, and consequent reflective action to preserve consistency.

As any change to a type will likely affect all of its subtypes, (with the exception of the alteration of a member which the subtype overrides) the derivation of a new version of a type must also implicitly create new versions of every type which inherits, directly or indirectly, from the type. Bidirectional inheritance relationships between a type and its super-type enable Vejal meta-classes to automatically create these implicit versions.

Meta-classes can determine type dependencies by analysing the types of instance and local variable declarations (including method parameters) and the return types of method invocations. Moreover, meta-classes can analyse the use of variables, i.e., the fields referenced and methods invoked on variables, to determine compatibility with versions of other types.

The representation of Vejal programs enables the framework to query the parsed code for program structures which are directly affected by evolution. For example, an evolutionary change which removes an instance variable from a type invalidates all references, both internal and external, to that instance variable. The meta-representation of Vejal operations allows querying the code for relevant references. Evolution primitives which operate on Vejal meta-data perform such consistency searches, and draw attention to circumstances resulting from the enacted change. This enables evolution approaches to take appropriate action. An evolution approach could implement a set of generative handlers to take autonomous action, or could simply notify the database programmer that further evolution need occur. Take, for example, the case of renaming an instance variable, as shown in Fig. 2.

```
Person <1.0> {
    String firstname;
    String surname;
    double weight;
}
```

```
Person <2.0> {
    String firstname;
    String lastname;
    double weight;
}
```

Fig. 2: A renamed field

Any operations defined by the type, or by dependent types, with references to the previous field handle *surname* must also evolve to use the new field name, *lastname*. As this evolutionary change does not have any subtle semantic implications, it is a perfect example of behavioural inconsistencies which can be detected and handled automatically.

Execution-level meta-data in the evolved type (Person<2.0>) are searched for field references to the deprecated field name and altered to refer instead to the new field name. External references to the altered field are detected by searching dependent types and applications. Matching field references are first queried by field name. Subsequently, the types of field references are verified by using the execution-level meta-data in a manner similar to a compiler to determine the static type of their target variables.

Note that although such an action presents a significant overhead, it only occurs once when a type is evolved, and is considerably less work than manually revising code.

3.1 Consistency using Reflective Handlers

To demonstrate the consistency management features of Vejal, we use the restructuring scenario from [2] and [4] which affects the direction of an aggregate relationship between two types, Supplier and Part. Initially, a Part contains a set of Suppliers. The proposed reorganisation of the data, however, reverses the nature of the relationship between the two entities – a part loses the knowledge of which suppliers carry it and a supplier gains a set of parts.

The nature of this evolution removes an instance variable, *suppliers* from a type, *Part*. This removal creates inconsistencies in any code which references the removed entity, which [2] proposes to handle via reflectively altering dependent code. With Vejal's ability to detect invalidated references in loaded types and applications, inconsistencies can be handled by reflectively introducing and initialising a local variable with the appropriate value prior to use.

Fig. 3(a) shows a simple method, defined within the type *Part,* which uses the removed field *suppliers*. Note that the removal of *suppliers* leaves the method (as well as any other dependent entities) in an inconsistent state as it contains a reference to a removed entity. Fig. 3(b) illustrates how such an inconsistency could be uniformly handled by using reflective generators [2]. The handler defined in Fig. 3(b) responds to a Removed Member Referenced Exception, which occurs as the result of consistency checking within evolution primitives. The exception carries information relevant to the inconsistency, e.g., the Vejal method in which it occurs, enabling handlers to take necessary action. The handler shown in Fig. 3(b) updates the inconsistent method by adding code which declares and initialises a local variable, *suppliers*, to the appropriate value. Line 8 of Fig. 3(b) retrieves the inconsistent Vejal method from the exception. Subsequently, lines 14, 15, and 16 declare the necessary Vejal handling code, parse it, and merge it with the inconsistent method, respectively. The Vejal method resulting from the application of the handler in Fig. 3(b) to the Vejal method in Fig. 3(a) is listed in Fig. 3(c).

```
void listSuppliers () {
  Iterator iter = suppliers.iterator();
  while (iter.hasNext())
    print(iter.next());
}
```

(a)

```
1  public class RemoveHandler extends ExceptionHandler {
2    public RemoveHandler (InterpreterException exception) {
3      super(exception);
4    }
5
6    public void handleException () {
7      RemovedMemberReferencedException rmre =
       (RemovedMemberReferencedException)exception;
8      MetaMethod method = rmre.getReferer();
9      MetaMember removed = rmre.getRemovedMember();
10     Type declaredIn = method.getDeclarer();
11     Type part = new Type("Part");
12     if (declaredIn.equals(part)) { // Handling for Part
13       if (removed.getName().equals("suppliers")) {
14         String handlerCode = "QueryEnumeration result =
Database.where([Supplier], \"hasPart(\" + partNo + \")\");  Set suppliers
= new HashSet(result.toCollection());";
15         Statement parsedHandlerCode =
              VejalClassLoader.parseStatement(handlerCode);
16           Block methodBody = Method.getMethodBody();
17           MethodBody.prependStatement(parsedHandlerCode);
18       }
19     }
20   }
21 }
```

(b)

```
void listSuppliers () {
  QueryEnumeration result= Database.where([Supplier], "hasPart(" +
            partNo +")");
  Set suppliers = new HashSet(result.toCollection());
  Iterator iter = suppliers.iterator();
  while (iter.hasNext())
    print(iter.next());
}
```

(c)

Fig. 3. (a) Code invalidated by the removal of instance variable *suppliers* **(b)** Handler declaration **(c)** Inconsistency correctly handled by introducing and initializing a local variable

Unfortunately, at present there is no way to declare handlers in an ad hoc manner, and all handlers as seen in Fig. 3(b) must be written and compiled in Java, and then associated with the schema evolution strategy prior to the correlated evolution. Notice the use of bracket ('[', ']') delineated type names in Figs. 3(b) and 3(c). This is a provision of the Vejal language which allows referencing types as first class entities. Note also that Figs. 3(a) and 3(c) are code listings in Vejal, whereas Fig. 3(b) is Java.

4. THE META-JOINPOINT/ADVICE MODEL

AspOEv utilises integrated aspects to achieve modularisation of evolution concerns: both join points and advice are first class objects within the framework. Points of interest are captured with meta join points (similar to AspectWerkz), known as *Bindings*, that are woven into the core OO portion of the framework using

AspectJ. At the application level, a Binding is simply an object representation of a join point and its associated advice. In some respects, a Binding is similar to an Event-Condition-Action (ECA) rule within active databases [3]. It must be observed, however, that Bindings and their associated advice exist outside of the database and are not rule-driven.

Fig. 4 illustrates how a Binding operates over a join point. Program control passes to the Binding when the join point is met, the Binding then executes *before* and *pre-proceed around* advice, then the join point itself (if not circumvented), followed by its *post-proceed* and *after* advice, and finally returns control and execution resumes. At runtime, a Binding can be viewed simply as a collection of *before*, *after*, and *around* advice surrounding a join point.

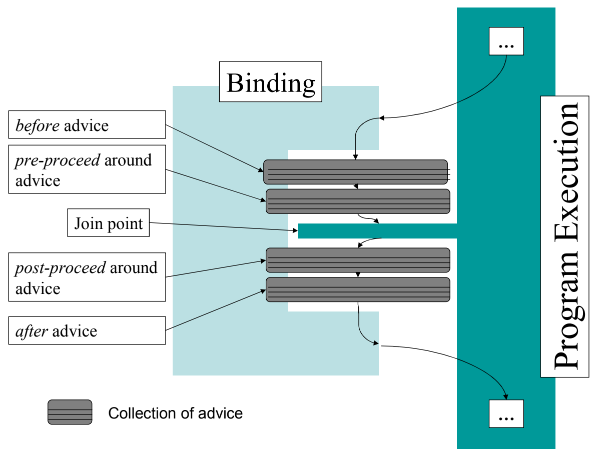


Fig. 4: Activation of a binding

When AspOEv is built, the framework uses declared Bindings to generate AspectJ code and create a seamless interface between AspectJ advice and the executing Vejal environment. Declaration of Bindings in AspOEv is nearly identical to the specification of activation of join points in [6]. Bindings are declared and added to a 'Binding Manager.' The Binding Manager creates an AspectJ aspect, to which each binding adds:

- a runtime declaration of the Binding itself;
- a declaration of the Binding's arguments;
- an AspectJ pointcut targeting the Binding's specified join point;
- an AspectJ *around* advice creating an interface with the executing Vejal environment.

AspectJ *around* advice generated for a binding is summarised in Fig. 5. Via manipulation of Vejal scopes, the AspectJ advice prepares the Vejal environment in such a way as to be both supportive of Vejal advice and safe for the executing Vejal code to which the Vejal advice applies. The AspectJ advice can then execute any Vejal advice associated with the binding.

Our *binding model* yields a number of advantages. Firstly, Bindings are accessible dynamically, allowing management of applied advice at runtime. Moreover, because all advice is written in Vejal, and Vejal is interpreted, advice can be added and removed dynamically, as well as making advice subject to the flexibility and safety of the interpreter.

The creation of a seamless interface between AspectJ advice and the Vejal environment allows the framework to apply interpreted advice to aspects, permitting the dynamic addition and removal of advice from specified binding points. The framework is thereby able to take a unified approach to evolution, viewing all evolution events – from changes incited by primitives, to widespread, long lasting evolution strategies – as aspects applied to Vejal meta-data and instances.

```
Object around ( arguments ) : bindingPointcut( arguments ) {
  - Push Vejal environment scope
  - Declare arguments as variables in Vejal environment
  - execute binding before advice
  - execute binding pre-proceed advice
  if ('proceed ( )' has been called in pre-proceed advice) {
     - pop and save Vejal environment scope
     Object result = proceed( arguments );
     - restore Vejal environment scope
     - store result of proceed call in Vejal 'proceed' variable
  }
  - execute binding post-proceed advice
  - execute binding after advice
  - Pop Vejal Environment Scope
  - return appropriate value
}
```

Fig. 5: Pseudo code of Binding-generated AspectJ advice

4.1 Dynamic AOP in Vejal

The ability to dynamically add and remove Vejal advice from bindings provides dynamic AOP capability to Vejal. However, currently, advice defined within types can only utilise variables which are declared statically within the execution environment, or associated with the Binding to which they subscribe – instance variables within the type are inaccessible as the advice is essentially static, and therefore, not associated with any individual instance of the type. Fig. 6(a) shows the declaration of a piece of advice within a class declaration.

```
class BindMonitor {
  before (bind) {
    bindCount = bindCount + 1;
  }

  before (unbind) {
    bindCount = bindCount - 1;
  }
}
```

(a)

```
after (someBinding) {
  // Some functionality
  if (someCondition) {
    thisBinding.removeRule(thisAdvice);
  }
}
```

(b)

Fig. 6. (a) Simple AO functionality within Vejal **(b)** Removal of advice from a binding

The Vejal class shown in Fig. 6(a), *BindMonitor*, merely keeps track of the number of objects bound within the database. Bindings *bind* and *unbind* refer to methods on the database manager of the same name. Note that the variable referenced within the advices to *bind* and *unbind*, *bindCount*, must be statically declared (not shown in Fig. 6(a)) before it can be used. While meta advice declared within a type does store a reference to the type in which it was declared, currently there is no mechanism for resolving which instance of the type advice code refers to – i.e. as in AspectJ's deployment model via *percflow*, *perinstance*, etc. – thus only statically declared variables are available for manipulation within advice.

Fig. 6(b) shows dynamic AOP functionality in Vejal by demonstrating how advice can dynamically unsubscribe itself from a Binding. The variable *thisBinding* refers to the currently executing Binding, while *thisAdvice* to the currently executing piece of advice.

5. THE EVOLUTION PRIMITIVE MODEL

Vejal allows AspOEv to provide database programmers the option of using expressive, single-use primitives. Many object database systems supply the database programmer with a special language for executing predefined evolution primitives, e.g., Odberg's *Change Specification Language* [5]. These languages are focused, however, only on accepting input concerning the evolution primitive desired, as well as any parameters required by the primitive to carry out its particular change. This is problematic for a number of reasons:

- The programmer must learn another language.
- Often these languages are created for the sole purpose of expressing primitives which are predefined by the object database system, and thereby lack the capability to express custom evolution semantics.
- Extending the language (if possible) could potentially require modifying the language's parser. Due to the simplicity of most primitive expression languages, the parser is apt to be ad hoc.

As opposed to introducing yet another language into the framework, we allow database programmers to manipulate the schema through Vejal's Command Line Environment (CLE). The CLE allows schema changes to be expressed and interpreted in a manner consistent with evolution rules defined within the AspOEv framework. Vejal primitives can also utilise predefined framework primitives, and apply to multiple types, or in fact, the entire schema. As opposed to creating and managing potentially large collections of Java objects that represent obscure primitives that might only be used once, ad-hoc use of Vejal code provides the immediacy, convenience, and expressive ability to create powerful and diverse primitives.

Fig. 7 shows a primitive which adds the type *Object<1.0>* as a base type to any type which does not have any base types, cf. lines 7 and 8. Additionally, in lines 3 – 6, the primitive circumvents any collisions, which might occur as a result of the introduced base type, by renaming the field *uniqueID* in any type which defines it.

```
class Object<1.0> {
  double uniqueID;
}
```

```
1  edit (*) {
2    List supertypes = Type.getSuperTypes();
3    if (Type.declaresField("uniqueID")) {
4      String newName =
         Type.getTypeName() + "_uniqueID";
5      RenameFieldCommand(Type, "uniqueID", newName);
6    }
7    if (supertypes.cardinality() == 0) {
8      Type.addSuperType([Object<1.0>]);
9    }
10 }
```

Fig. 7: Specialised, single-use, ad hoc primitive

Note that the primitive in Fig. 7 is very specialised and would likely only need to be used once. Therefore, ad hoc declaration of the primitive in Vejal is preferable to creating a dedicated Java object – complex and specialised primitives written in Java would have to be written in a separate text editor, compiled, reflectively fetched and loaded, and then executed. Vejal code, on the other hand, can be more readily integrated, as the CLE provides AspOEv with a means of accepting and executing valid Vejal code.

By clarifying the semantics of a change, detailed evolution primitives can lessen the burden on the database programmer to specify how that change affects existing data during instance conversion. Moreover, the ability to apply primitives to multiple types facilitates the evolution of the schema as a whole.

6. CONCLUSION

In this paper, we have presented, Vejal, an aspect language with a versioned type system. The language underpins the AspOEv evolution framework supporting adaptation of object database evolution strategies. A major advantage of object databases lies in transparent persistence, i.e. to supply OO developers with the ability to express persistence concerns through techniques that they already know and understand, without the need for specialised syntax. Vejal extends this idea to the AO level by integrating aspect capabilities within its versioned type system, hence providing a means of expressing AO concerns in user-defined persistent types that can transparently act upon persistent data. In addition, it facilitates aspects to operate uniformly on multiple levels of execution - at both the Java and Vejal level. As aspects bind to elements within the Vejal Abstract Syntax Tree (AST), their causality relationships with evolution strategies in fact originate from the Vejal execution the AST node expresses.

Vejal also facilitates specification of both static (i.e. on types) and dynamic (i.e. on instances) action to associate with a change. An evolution primitive can, therefore, be viewed as an aspect to be applied to a type and its instances. The aspect-primitive, written in Vejal, gives the database programmer greater expressive ability than a single-purpose primitive specification language. Furthermore, structures provided for declaring primitive execution permit schema-wide changes by allowing conditional evaluation and application of the primitive to multiple types.

Acknowledgement. This work is supported by UK Engineering and Physical Sciences Research Council Grant GR/R08612.

7. REFERENCES

[1] J. Banerjee, et al., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", ACM SIGMOD Conference, 1987, ACM, SIGMOD Record, 16(3), pp. 311-322.

[2] R. C. H. Conner, et al., "Using Persistence Technology to Control Schema Evolution", ACM SIGAP Conference, 1994.

[3] K. R. Dittrich, et al., "The Active Database Management System Manifesto: A Rulebase of ADBMS Features", 2nd Workshop on Rules in Databases, 1995, LNCS 985, pp. 3-20.

[4] G. N. C. Kirby, et al., "Using Reflection to Support Type-Safe Evolution in Persistent Systems", University of St. Andrews, UK, Technical Report No. CS/96/10 1996.

[5] E. Odberg, "MultiPerspectives: The Classification Dimension of Schema Modification Management for Object-Oriented Databases", Proc. TOOLS-USA, 1994, IEEE.

[6] A. Popovici, et al., "Just-In-Time Aspects: Efficient Dynamic Weaving for Java", Proc. AOSD 2003, ACM, pp. 100-109.

[7] A. Rashid, "A Database Evolution Approach for Object-Oriented Databases": PhD Thesis, Computing Department, Lancaster University, UK, 2000.

[8] A. Rashid, "Aspect-Oriented Programming for Database Systems", in *Aspect-Oriented Software Development*, Addison-Wesley, 2004, pp. 657-680.

[9] A. Rashid, N. Leidenfrost, "Supporting Flexible Object Database Evolution with Aspects", Proc. GPCE 2004, LNCS 3286, pp. 75-94.

[10] D. Sjoberg, "Quantifying Schema Evolution", *Information and Software Technology*, 35(1), pp. 35-44, 1993.

Tracking and Assessing the Evolution of Scattered Concerns

Martin P. Robillard

School of Computer Science
McGill University
Montreal, QC, Canada
martin@cs.mcgill.ca

ABSTRACT

In this position paper, we describe how we document the implementation of scattered concerns by combining intensional descriptions of relations between program elements and their corresponding extensions for a specific version of a program. We show that this strategy allows us to automatically track the source code implementing a concern as it evolves and to assess the stability of a concern's implementation. We illustrate these benefits with results obtained from ongoing empirical studies of the evolution of scattered concerns.

1. INTRODUCTION

Software modifications often address concerns, or features, whose implementation is scattered across a number of modules. In such cases, developers often have to spend a significant amount of effort investigating a system to identify all the code locations which may be associated with the change. When repeated changes address a same scattered concern, the continual re-investigation of the code can directly translate into inefficiencies of the software development process.

One way to address this problem is to explicitly link the description of a concern with the code implementing the concern. With this approach, first proposed by Soloway et al. in 1988 [16], annotations or artifacts document how specific parts of the source code relate to different scattered concerns. Different forms of tool support can help developers view and navigate this information to ease software development tasks. One universal challenge with this approach, however, is that every time a system is modified, the concern documentation is at risk of becoming invalid, and must be constantly maintained.

In our current work on concern representation, we are developing and evaluating ways to model concerns that can withstand the destructive effects of source code evolution. In one of our approach, *concern graphs* [12, 13], we represent the implementation of a concern using both an intensional specification (e.g., "all the callers of method `m1()`") and the corresponding extension on a specific version of a code base (e.g., `m2()` and `m3()`). This way, it is possible to automatically detect when the code evolves to a point where the projection of an intension on a code base does not correspond to the extension (what we call a concern graph inconsistency).

Past [15] and ongoing legacy [18] studies of concern evolution have provided encouraging evidence that concern graphs can be used to track and assess the evolution of scattered concerns. In the rest of this paper, we describe several observations we made about the value of combining intensional and extensional specifications to describe the implementation of concerns. Among others, simple heuristics applied to concern graph inconsistencies enable the automatic detection of new concern methods, of changes to method signatures, of class and method moves, of moves to code blocks, etc. In addition, we found that studying the history of concern graph inconsistencies between different versions of a system allowed us to assess the relative stability of a concern's implementation for the purpose of refactoring the concern's code to a different object- or aspect-oriented modularity.

2. BACKGROUND

A concern graph [13] is an artifact representing the implementation of a concern in source code by documenting the relations between the different program elements involved in the concern's implementation (fields, methods, etc.). In this paper, we present a simplified version of concern graphs that includes only the concepts necessary to understand the paper. The complete description of the most recent version of the concern graph framework can be found in a separate report [12].

In the concern graph framework, a *concern* is a named collection of *fragments*. A fragment represents a basic relation between program elements that are relevant to a concern's implementation. The definition of a fragment includes an *intension*[1] and its corresponding extension representing the actual range of the relation for a specific version of a program.

For example, if a developer decides that all the accessors of a field `f` in a class `C` are associated with the implementation of a concern, then the following intension is recorded:

```
C.f accessed by ALL.
```

When this intension is recorded, a tool analyzes the current version of the code of the program and determines the extension (e.g., methods `C.m1()` and `C.m2()`). The complete fragment recorded in the concern graph then consists of the intension `C.f accessed by ALL` *and* its extension {`C.m1()`,`C.m2()`}.

By combining an intension and its extension in a fragment, whenever the program evolves, the intension can be *projected* onto new versions of the program to determine if the *generated* extension still corresponds to the *stored* extension. Inconsistencies between the generated and the stored extensions indicate modifications that invalidate the concern graph (a concern graph inconsistency).

[1]We use the term "intension" in the sense of Eden and Kazman, to denote a structure that can "range over an unbounded domain" [4, p. 150]

In practice, concern graphs are created and used with an Eclipse[2] plug-in called FEAT [14]. FEAT augments Eclipse with a number of search facilities for program investigation (e.g., to obtain all the accessors of a field) that allow a user to add the entire result of a search as a fragment in a concern (the intension is the query and its extension is the query result). Every time source code in an Eclipse project associated with a concern is modified (or when a concern graph is loaded), FEAT re-projects the intension of each fragment and checks the resulting extension for inconsistencies with the stored extension. FEAT reports inconsistencies in a specialized Eclipse view that describes which fragments are inconsistent and why they are inconsistent. For example, Figure 1 shows the FEAT Inconsistencies View displaying the details of the inconsistencies for a fragment with the intension

`Marker.createPosition() called by ALL`.

The adornments on the elements in the detailed view (bottom panel) show that the stored extension is missing a call from `parseBufferLocalProperties()` but includes a call from a method that no longer exists, `parseBufferLocalProperties(String)`.

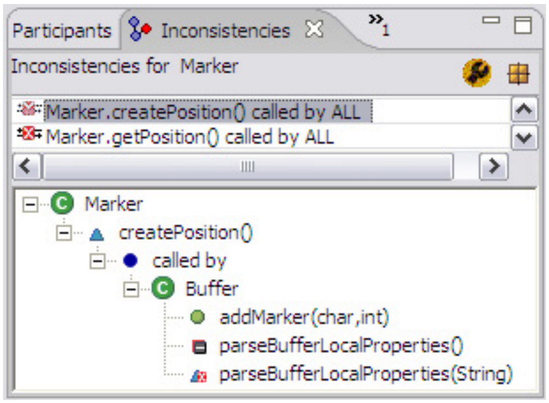

Participants
Inconsistencies
Inconsistencies for Marker
Marker.createPosition() called by ALL
Marker.getPosition() called by ALL
Marker
createPosition()
called by
Buffer
addMarker(char,int)
parseBufferLocalProperties()
parseBufferLocalProperties(String)

Figure 1: The FEAT Inconsistencies View

3. CONCERN TRACKING

Concern graphs are models that abstract the implementation of a concern as a relation between program elements. As for any model, concern graphs are intended to represent the essential properties of a phenomenon while abstracting away the details. In the case of concern graphs, the essential properties of a concern's implementation are the inter-procedural relations between program elements (e.g., fields, methods), and the details include the syntax of statements, variable names, local interactions, and comments.

3.1 Minor Code Changes

Details of the code and local interactions have no impact on the information stored in a concern graph. As an example, let us consider the following Java source code

```
class A {
   void m(boolean p) {
      // p is true in the morning
      if(p) x();
      else y();
   }
}
```

[2]www.eclipse.org

If a developer decides that all of the methods called by `m` are involved in the implementation of a concern, this fact can be specified with the intension:

`A.m(boolean) calls ALL`.

In a concern graph this intension would be stored in a fragment that would include the extension {`x()`,`y()`}.

In this case, any change that would not involve either removing or adding a method call in `m` would have no effect on the concern graph. For example in the revised version

```
class A {
   void m(boolean p) {
      // p is true at night
      if(!p) x();
      else y();
   }
}
```

the extension corresponding to `m(boolean) calls ALL` is preserved (and the concern graph is likely to remain valid) despite the changes to the comment and branching predicate.

This basic tolerance to changes in source code is an important property of concern graphs that facilitates the tracking of concern code. Since concern graphs are not intended to describe the *behavior* of a concern's implementation but rather its *location*, the risk of transparently invalidating a concern graph by changing the local implementation of a method is very low. In our case, this would only occur in the case where changes to the source code invalidate the intension while preserving the extension. In other words, we would have to change `x()` or `y()` in a way that renders the method irrelevant to the concern, while preserving all existing relations encoded in fragments. We have not yet encountered such a case in practice.

3.2 Major Code Changes

In addition to minor modifications, a software system will also undergo more important changes that will affect the validity of the concern graph. For example, if we now change our example code fragment to:

```
class A {
   void m(boolean p) {
      // p is true at night
      if(!p) x();
      // No else branch
   }
}
```

the extension corresponding to `m(boolean) calls ALL` is now {`x()`}, which is inconsistent with the previous extension {`x()`,`y()`}. Experience and studies of concern graph evolution have demonstrated that analyzing the details of concern graph inconsistencies can give us a wealth of information about the nature of the source code changes.

We are currently studying how concern graph inconsistencies can be translated into high-level information about concern changes that can be used to automatically adapt the concern graph to reflect the code changes. We present a number of our results here in the form of heuristics. In the following descriptions, we use the expression *synchronizing a fragment* to indicate updating a fragment's stored extension with a generated version.

HEURISTIC 1 (SIGNATURE CHANGE). *If a generated extension is missing a method and contains an extraneous method of the same name defined in the same class but with a different parameter list, the missing and extraneous elements probably represent a change in the method signature that does not otherwise impact the concern graph. The concern graph can automatically be repaired by synchronizing the fragment.*

Figure 1 shows a case where this heuristic applies.

HEURISTIC 2 (ELEMENT MOVE). *If a generated extension is missing an element in class C and contains an invalid element with the same name in a class C', the element was probably moved from C to C'. The concern graph can be automatically repaired by synchronizing the fragment.*

HEURISTIC 3 (NEW CONCERN ELEMENT). *If a generated extension is missing an element that cannot be associated with any other heuristic, the missing element is probably a new element introduced in the concern. The new element should be inspected by a developer.*

HEURISTIC 4 (ELEMENT RENAME). *If an element is missing from a number of extensions and, for each extension, there is a corresponding invalid element, the invalid element was probably renamed. The concern graph can be automatically repaired by synchronizing the fragment.*

HEURISTIC 5 (CODE BLOCK MOVE). *If an element is missing from a number of extensions and, for each case, there is a corresponding extraneous but valid element, a concern-related code block may have been moved. Confidence that this heuristic applies increases if a number of fragments present the same inconsistencies.*

Our studies also showed cases where we could refine the Element Move heuristic into the more specific Pull Down Method.

HEURISTIC 6 (PULL DOWN METHOD). *If a generated extension is missing an element e in class C and contains invalid elements with the same name in a class C', and C' is a subclass of C, we can say that e was probably pulled down from C to C'.*

We are currently designing a way to automatically encode, detect, and execute these heuristics in the FEAT tool, to increase the level of automation with witch we can track concern code in evolving software.

4. CONCERN CHANGE ASSESSMENT

In addition to facilitating the tracking of concern-related code throughout the evolution of a system, analyzing concern graph inconsistencies can help assess the relative stability of different parts of the code relating to a concern. This is important when trying to decide whether and how a concern can be refactored, either using traditional object-oriented refactoring [6] or aspect-oriented techniques[5].

One way to assess the (in)stability of a concern's implementation is to count the number of times a given fragment became inconsistent as the result of the evolution of a system. In our empirical studies, analysis of this factor allowed us to make two simple but important observations. First, a very stable class can be used by very unstable code. In one case, a class that was part of the core implementation of a concern went through only 6 revisions throughout the 4-year history of the system. However, the code referring to this class was in a constant flux that resulted in many concern graph inconsistencies. This example shows that we cannot use naive metrics such as the number of file revisions to assess the stability of a concern's implementation.

Our second observation is that AspectJ [9] programmers should be careful when specifying pointcuts intensionally. We now have a collection of cases showing that, in practice, the extension corresponding to an pointcut-like intension may be changing over time. For example, an AspectJ pointcut may have many different sets of shadows in the history of the system without programmers necessarily being aware of the situation. Although in ideal AOP programs the differences should not matter, in reality they might. Storing the extensions corresponding to intensional specifications and analyzing the inconsistencies between extensions in different versions of a system enables the automatic detection and notification of such cases.

5. RELATED WORK

A number of approaches have been proposed that allow developers to specify a subset of the source code of a program using intensional specifications.

For example, the Stellation [1, 2] software configuration management system supports virtual source files using a typed aggregation mechanism that can intensionally collect different program elements and other artifacts in a single modular unit. The Aspect Browser is a tool proposed to help developers find concerns using lexical searches of the program text [7]. In AspectBrowser, a concern description is intensionally defined as a set of regular expressions. As another example, Intentional Views [10, 11] allow developers to specify different views of a system that reflect some form of commonality, which can include relevance to the implementation of a concern. With Intentional Views, developers can manage the evolution of concerns by providing alternative intensions for the same concern, whose extensions can then be checked for consistency as the software evolves. Finally, The Concern Manipulation Environment (CME) [8] includes a Concern Explorer tool that can be used to describe concerns using queries (i.e., intensions) in a way that is similar to FEAT, but without the support for inconsistency analysis between different extensions corresponding to a single intension.

We see the number and variety of approaches for intensional specification of concern code as an exiting indicator that such an idea is feasible and practical. The main difference between concern graphs and previous approaches is that concern graphs store the extensions that correspond to programmer-defined intensions. Only with this property can the analyses described in the paper be possible.

A number of approaches have also been proposed for inferring past refactorings from as system's change history. For example, Demeyer et al. [3] analyze changes in object-oriented metrics (e.g., the number of messages sent) to infer potential past refactorings. Xing and Stroulia [17] developed an approach to recognize past refactorings based on changes in a class hierarchy as documented in an object-oriented design model. In contrast, our inferences are done using concern graphs, an abstraction that is closer to the source code than both metrics and class diagrams.

6. CONCLUSIONS

Artifacts that describe the source code implementing scattered concerns can be helpful to developers performing program evolution tasks. However, an artifact referring to source code typically becomes inconsistent as the code evolves, reducing its effectiveness. In this paper, we described how we can efficiently evolve descriptions of concerns in source code together with with the evolution of a system. The key to enabling the inexpensive evolution of concern descriptions is to combine intensional specifications of the concern code with their corresponding extensions. Analysis of the inconsistencies between extensions stored in a concern description and extensions generated on a program by projecting the corresponding intensions can help us determine how to adapt the concern descriptions, and provides us with valuable insights on the stability of a concern's implementation as the code evolve.

Acknowledgments

The author is grateful to Gail Murphy and to the LATEr 2006 referees for useful comments on this paper.

7. REFERENCES

[1] Mark Chu-Caroll, James Wright, and David Shields. Supporting aggregation in fine grained software configuration management. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 99–108, 2002.

[2] Mark C. Chu-Carroll and Sara Spenkle. Coven: Brewing better collaboration through software configuration management. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering*, pages 88–97, 2000.

[3] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the Conference on Object-Oriented Programming, Systems, and Applications*, pages 166–177, 2000.

[4] Ammon H. Eden and Rick Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software engineering*, pages 149–159, 2003.

[5] Tzilla elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, 2001.

[6] Martin Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technologies Series. Addison-Wesley, 2000.

[7] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274, 2001.

[8] William Harrison, Harold Ossher, Stanley Sutton Jr., and Peri Tarr. Concern modeling in the concern manipulation environment. Technical Report RC23344, IBM Research, 2004.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):51–57, 2001.

[10] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 289–296, 2002.

[11] Kim Mens, Bernard Poll, and Sebastián González. Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 169–178, 2003.

[12] Martin P. Robillard. *Representing Concerns in Source Code*. PhD thesis, Department of Computer Science, University of British Columbia, Canada, 2003.

[13] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002.

[14] Martin P. Robillard and Gail C. Murphy. FEAT: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, pages 822–823, 2003.

[15] Martin P. Robillard and Gail C. Murphy. Evolving descriptions of scattered concerns. Technical Report SOCS-TR-2005.1, School of Computer Science, McGill University, 2005.

[16] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

[17] Zhenchang Xing and Eleni Stroulia. Recognizing refactoring from change tree. In *Proceedings of the First International Workshop on Refactoring: Achievements, Challenges, and Effects*, pages 41–44, 2003.

[18] Martin V. Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *IEEE Computer*, 31(5):23, 1998.

Empirical Study for Evaluating Evolvability Requirements

Christa Schwanninger, Iris Groher, Regine Meunier, Uwe Hohenstein
Siemens AG, CT SE 2, Otto-Hahn-Ring 6,81739 Munich, Germany
{christa.schwanninger, regine.meunier, uwe.hohenstein} @siemens.com, Iris.Groher@students.jku.at

ABSTRACT

To convince the industry to use aspect-oriented programming techniques we have to defeat the prejudice against this paradigm. One mean to achieve this are studies proving the benefits of AO and disproving the retentions against it. We describe a small comparative study comparing one OO with three AO languages. The study results and the experience we gained help to plan and conduct further studies to show the value of AO.

1. INTRODUCTION

1.1 Evolvability

Aspect-oriented (AO) languages improve the modularity of systems. The claim that increased modularity improves understandability and thus the evolvability of systems is as old as modular and object-oriented (OO) programming. AO proponents see improved modularity as the main advantage of this paradigm [1, 2], critics argue that developers loose the overview when crosscutting concerns get modularized. A modularized concern still crosscuts the application. Understanding its effect when separated could be harder than if it was tangled with the code. The modules that are influenced by the crosscutting concern miss part of their context, the aspect contains pointcut declarations that are brittle and often hard to understand and handle without tool support. If the first position is wishful thinking or the second is a prejudice needs to be proven first before many practitioners in industry are willing to adopt AOP. We suggest giving evidence to practitioners that AO really improves the evolvability of software. One means to do so is to conduct empirical studies that prove that AO programs are easier to understand, easier to extend and thus easier to evolve than procedural or OO programs.

1.2 Background

Our industrial research group evaluates AO languages, tools and methods (among other paradigms) to find out, which AO technologies are helpful and mature enough to be used in industry. We recommend mature concepts to internal partners in product development, prepare teaching material and provide support.

To get evidence on the maturity and applicability of different AO languages in comparison to OO, we implemented a demonstrator that simulates a couple of use cases of an industrial application (size is 57 classes in Java) in three different AO languages, AspectJ [3], Hyper/J [4] and CaesarJ [5], and in Java using OO design patterns [6]. We compared the implementations according to

- Quantifiable criteria such as size, coupling, cohesion, complexity, inheritance hierarchy depths, and runtime
- Qualitative criteria such as understandability and extensibility

This paper presents the results of a small comparative study falling into the second category.

We hope to get feedback from the workshop participants, hear about similar experiences and trigger more studies to

- Tackle the prejudice against AOP, like programs are less easy to understand, to debug and to extend.
- Get material to convince management and development teams to use AO languages when appropriate.

Our main intention for this first comparative study was to get a first sense on how different the language ratings are and also for how much preparation such a study requires. Our sample is small, but the resulting data allows creating hypotheses for further investigations. We asked Siemens employed diploma and PhD students and two interns to participate.

A second source for comparing the languages qualitatively is professionals. Considering the time constraints of professionals we plan a briefer version of our study, where we introduce all languages to a group of professionals and explain the designs for all our solutions. We will then ask them to rate how understandable, mature and applicable they find these languages for the development tasks they participated in. This will be the next step in our qualitative language comparison.

In the next section we explain the applications we wrote for our comparisons and how the study was prepared and conducted. Section 3 presents the study results, Section 4 gives a summary of the hypotheses we derived from the results and the lessons learned on conducting comparative studies. In 5 we briefly state our further plans and 6 talks about related work.

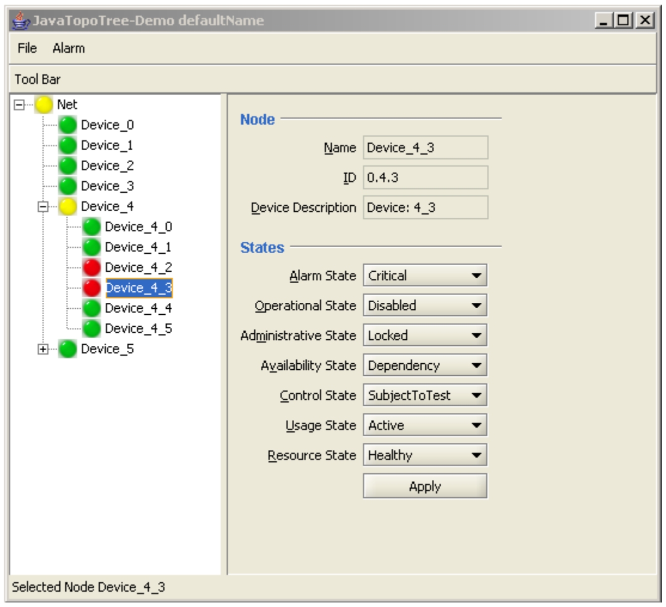
2. STUDY DESCRIPION

2.1 Application

We implemented a central and important part of a telecommunication network operation and maintenance application. The kernel entity of such an application is a topological tree (TopoTree) that visualizes the state of hardware elements in a telecommunication network, reflecting the topology of the network.

Topological trees are common in various application domains, e.g. for managing high bay warehouses, power distribution systems or traffic control systems. The main difference between the application domains is the kind of monitored elements, which can be goods, hardware elements or vehicles. For our example the topological tree represents a network infrastructure to be controlled by telecom network operators. Controlling here means that hardware network entities have to be monitored to take action in case of a failure. Relevant data has to be presented to the operator in an easy to grasp way in a tree view. Color encoding is used to indicate the state of the network. Leaf nodes are hardware entities such as CPU boards, which are then aggregated to higher entities like racks containing the boards, and so on. The whole structure forms an acyclic tree. One server application manages the model of the tree; it writes the recent state into a database and calculates state changes for tree elements. Clients have a GUI and present a view on the tree to operators.

In our implementation clients can change the state of nodes and push these state changes to the server, which is then responsible for calculating resulting state changes and updating all other clients. To make it easy for the operator to spot problems in the network, nodes are colored. The color code is: GREEN indicating normal operation, YELLOW for a warning, ORANGE for a major problem, RED for an error and WHITE for unknown state. These colors not only show the working state for each concrete leaf network element, the states of leaf nodes get aggregated and determine the color of the father node. This helps the operator to detect problems in a big network where most nodes representing higher level entities in the visualization tree are collapsed (e.g. node Device_5 in Figure1). Depending on the specific requirements, different state propagation strategies exist. If for example more than half of the children of a node are in a critical state, the father also shall show the coloring for critical state to make the operator aware of a major problem in a collapsed part of his visualization tree. Figure 1 shows a screenshot of such a client.



- Net
  - Device_0
  - Device_1
  - Device_2
  - Device_3
  - Device_4
    - Device_4_0
    - Device_4_1
    - Device_4_2
    - Device_4_3
    - Device_4_4
    - Device_4_5
  - Device_5

Node

Name	Device_4_3
ID	0.4.3
Device Description	Device: 4_3

States

Alarm State	Critical
Operational State	Disabled
Administrative State	Locked
Availability State	Dependency
Control State	SubjectToTest
Usage State	Active
Resource State	Healthy

Selected Node Device_4_3

Figure 1: TopoTree Client

2.2 Implementation Variants

We implemented the same requirements in four different languages, three of them aspect-oriented extensions to Java – AspectJ, Hyper/J and CaesarJ – and one in pure Java. The purpose of this is to have a basis for comparing the AO languages with pure OO regarding quality criteria like efficiency – runtime and code size – and development requirements like understandability and extensibility.

For all versions together we upfront decided on five concerns that were implemented with either the aspect modularization mechanism of the respective AO language or with design patterns in the OO case. We implemented these concerns such that they are easily exchangeable in each application, for four of the five concerns even several implementation variants exist. Beyond these concerns the implementers of the four versions were free to choose other concerns to modularize in aspects as they see fit best in the specific language. For the five predefined concerns we were interested in how good these concerns can be encapsulated in each language without corrupting the understandability of the whole application. We found out that all four languages were well suited for encapsulating these concerns. But as we stated in the beginning encapsulation is not a goal in itself but should serve the goals of understandability and extensibility. For this purpose we conducted an extensibility study to help us judge how well the different languages support understandability, extensibility and thus evolution. We did not yet check for properties like testability and independent reusability of modules or aspects, which are also very important for evolvability.

2.3 Concerns

The concerns we chose up front were:

- Persistency: The TopoTree application supports three different variants for retrieving the data: 1. reading the whole tree from the database as soon as the first client requests a node (eager persistence), 2. reading only the requested nodes, further nodes are read when a node gets expanded for the first time by a client (lazy persistence) or 3. not reading the database but generating the nodes only in main memory (no persistence).
- Propagation: Whenever an alarm state is raised on a node, a propagation strategy determines how its ancestors are affected. Two strategies are implemented in each version of the TopoTree.
- Remoting: The TopoTree either runs as a stand-alone application or in a client/server mode, where the server administers the model and arbitrary many clients show views on the model and feed back changes to the operator. For the second variant nodes should be remote objects.
- Client-Update: The client update concern cares for updating all client views when a change in the model occurs. This concern can be switched on or off. When it is missing, the application does not fulfill all its requirements, since operators should always see the current state of the network.
- Tracing: Different tracing levels should be possible.

2.4 Concern Implementation

Table 1 shows how the different concerns were implemented in the 4 versions of the TopoTree.

2.5 Study Set-Up

We chose the Client Update concern to be implemented by our test persons. We omitted the aspect in the AO examples, for the OO version we threw out updating code. For each of the AO languages two students implemented the concern independently, for the Java version we had three students, two of them did not know Java before. We were interested in how hard it is to learn an AO language in comparison to learning the first OO language.

	Persistence	Propagation	Remoting	Client Update	Tracing
Java	Strategy pattern; strategy set in a property file and implementing class instantiated at startup time	Visitor pattern; propagation strategy set in a property file and implementing class instantiated at startup time	Proxy pattern; when no remoting necessary direct communication of view with model, otherwise through proxy object	Observer pattern, not unpluggable	Tracing class with choice of tracing level at runtime
AspectJ	Abstract aspect for common retrieving/storing functionality and specific aspects for each persistence strategy	One abstract aspect with common functionality and one concrete aspect for each propagation strategy	Aspect for exception softening, for remote reference creation and retrieval	Observer pattern with inter type declarations adding functionality to the base classes	Several tracing aspects
HyperJ	One hyperslice per persistence strategy	one hyperslice per propagation strategy	One hyperslice for remote reference creation/retrieval and BCEL adding of Remote interface and RemoteException	One hyperslice for implementing the Observer functionality	Several tracing aspect hyperslices
CaesarJ	On cclass (aspect)implementation for each concern and one concern selection cclass	One abstract aspect for propagation and concrete aspects for the different implementations.	Making use of CaesarJ's remoting feature	One tree update aspect conforming to the Observer pattern	Several tracing aspects

Table 1: Concern Implementation

Three of the students are master students in computer science; they never participated in product development, the programs they write for their diploma theses are their first reasonably sized projects ever. Four students are PhD students in computer science; two of them have considerable experience with projects of about 80 to 100 classes each. The third one did mainly database programming and had nearly no experience with object-oriented languages and never used Java before. The last two are interns, one of them studying computer science and having a lot of development experience in Java and one bio-engineer that had only developed software in assembler and C so far. They all had heard about AOP, but none of them had investigated the paradigm more closely or ever used an AO language. To be able to connect the student education with his/her feedback, we give them names assembled from the language they worked in and a number. So Java1 is the PhD student who mainly developed database applications and has nearly no OO experience, Java2 is a diploma student who has OO and Java experience, Java3 is an intern who never used OO before. AspectJ1 is a PhD student with considerable Java knowledge, AspectJ2 a diploma student with some experience, comparable to student HyperJ1, HyperJ2 is a PhD student with some Java experience again. CaesarJ1 is an intern with a lot of programming experience in research projects and CaesarJ2 a PhD student with similar experience. To make participating in the study affordable for the students we

- taught mainly the language features that were necessary to implement the extension
- gave quite concrete hints where the application should be extended; thus the students did not have to understand all parts of the application thoroughly before being able to make the extension. (Future studies require more emphasis on the understandability of bigger portions of code.)
- supported the students if they had non-AO specific problems. We e.g. readily helped them solve RMI related problems asking them not to add the time for this to the final effort.

The students all together got a 2.5 hour's introduction to the application and to AO in general with a brief summary of the three aspect languages. Every students group that implemented in an AO language then got a one to two hour's crash course in their AO language concentrating on the general concept and on the specific concepts they needed for their solution. For all groups including the three Java extension developers we walked through the relevant parts of their specific application and gave rather concrete hints what was missing where.

Every student got documentation on their language, which was

- popular Java books for the Java developers
- AspectJ in Action [6]] and the AspectJ Quick Reference for the AspectJ developers
- The Hyper/J User and Installation Manual [4] for the Hyper/J developers
- Tutorial slides from the CaesarJ language development team [5] for the CaesarJ developers

During implementation, we immediately helped when we saw that the students had problems with non-AO related tasks or when we thought we didn't introduce a language feature well enough.

All students worked with Eclipse 3.1.0. The AspectJ version was 1.5.0 for compiler and runtime and 1.2.0 for AJDT, the Hyper/J version was the latest downloadable, and the CaesarJ version 0.5.3.

Our questions fell into four categories. The first one was on the effort for understanding and implementing the extension, the second on language understandability, then on development and documentation support and the last block on how the student judges the usefulness and applicability of the language for development projects.

3. STUDY RESULTS

3.1 Effort Comparison

In general all students managed to solve the implementation task in nearly a day including understanding the application, understanding the language basics, designing, implementing and testing the solution. Figure 2 shows the effort the students spent on understanding the base application, designing their specific AO or non-AO solution, implementing and testing it. The number of participants is too low to draw strong conclusions, but we can derive some trends. These hypotheses need to be proven or invalidated by studies done with a bigger sample.

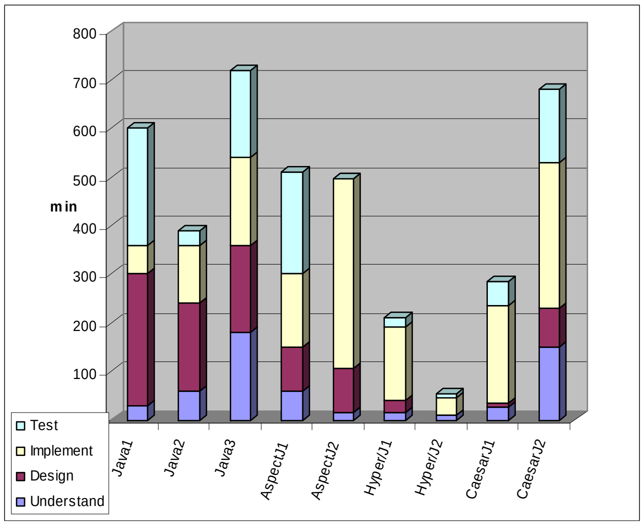


Figure 2: Development Effort

In general the students Java1 and Java3 who were new to Java and had to implement the Java extension had more problems dealing with the language than students who knew Java but were new to AO and had to implement one of the AO parts. Java3 never had used an OO language before; she was the one who had most difficulties in understanding the application and implementing the extension. The results for AspectJ and CaesarJ are similar, although in advance we were rather concerned that CaesarJ would be too different from the way how average OO developers think to get the task done without getting a full CaesarJ tutorial beforehand. However the students who volunteered to do the CaesarJ extension were the ones with the most programming experience; they already were familiar with generative programming and model driven development concepts and had more development experience than the other students.

We were surprised that the experienced Java developers who had to extend the HyperJ version were faster than the AspectJ developers. They felt that understanding the composition language was quite easy and developing a hyperslice requires mainly just Java knowledge. On the other hand they also said that it would have been really hard for them to identify all hyperslices in the first place, while the AspectJ developers found the concerns that were implemented as aspects quite obvious and would also have identified those as aspects.

We also found out that having the test persons measure the time for certain tasks themselves can temper the results. While some include every five minutes they thought about the problem during the coffee break, others tend to count only the time they fully concentrated on the solution.

3.2 Language Evaluation

In the next block of questions we wanted to know how easy to understand the students considered the languages and specific application. Table 2 “Language Evaluation Results” presents the results. 1 is the best grade and 6 the worst.

In average it seems to be harder for somebody who never used an OO language to understand Java than for a Java programmer to switch to an AO language based on Java.

For CaesarJ the documentation is not very rich, yet, this was one reason for the CaesarJ1 candidate to rate the language concept understandability very low. In general language concept understandability is better for AspectJ and Hyper/J, however the two CaesarJ ratings for how well the problem solving is supported are even slightly better than the AspectJ ratings. We had guessed AspectJ would be easiest to understand beforehand while we considered CaesarJ as the most different from the OO paradigm and thus hardest to understand. But this notion was not confirmed by the study results.

The fact that CaesarJ was considered more mature than AspectJ was also not what we expected. This definitely needs more validation in a study where each participant gets to know and rate all three AO languages.

3.3 Tool Support and Debugging

The next cluster of questions deals with the IDE and debugging support, see Table 3 “Tool Support and Debugging”. Again ratings were from 1 (very good), to 6 (not provided or very bad). Since all students used Eclipse as IDE with the plugins provided by the AO languages, there was a very good Java support and reasonably good AspectJ support, but no real Hyper/J and CaesarJ support, when it comes to language parts that are not Java.

For Hyper/J and CaesarJ (in the version we used) the IDE support beyond Java support is missing. This is also reflected by the answers to the questions concerning tool support. Debugging support is integrated for Java and AspectJ, some of the students rather used console printouts. Most of the students on the other hand really missed better browsing support especially for Hyper/J and CaesarJ. They said that for a bigger example they would need better visualization of the relationships between aspects and base code. For getting a good evaluation of the IDE integration the extension task obviously was too small. A study with a bigger extension task and a group of professional software developers would definitely lead to stronger results.

3.4 General Valuation

Finally, we wanted to get more general statements on whether the students can think of families of tasks or challenges they would consider the tested languages especially useful for, whether they would like to use and recommend the languages for their daily work and if they think the languages were mature. Table 4 “General Valuation” summarizes the results.

The whole Java group thinks that the language scales for industry projects, which is no surprise since everybody knows it does. From the lack of examples that could easier be solved with one of the languages we conclude that the students did not have enough experience to see a paradigm suitable for solving specific problems or that the time they spent with the language was too short to see its value. Again, a study with professionals has to be done to get better data on these questions.

One interesting fact is that one AspectJ implementer first solved his task with OO and then pulled the code out into an aspect. This developer also had major concerns that AspectJ could result in a bad design or easily be misused for patching a system. He therefore would not consider it for industrial use.

AspectJ and Hyper/J results are rather similar in the evaluation. CaesarJ is considered to be more a research language and needs a lot more tool support to be applied in real development projects.

4. SUMMARY AND LESSONS LEARNED

We conducted a small comparative study to evaluate how easy it is to extend an application written in 3 AO languages and one OO language. The study sample was small but sufficient to derive some hypotheses that have to be proven in studies with bigger samples. We encourage the AO community to conduct such studies to give evidence to the industry that AO is mature enough to be adopted and that quantifiable and qualitative attributes improve when applying AO technologies.

Some of our hypotheses that need further investigation are

- It is easier for an experienced OO developer to switch to an AO language that extends an OO language than it is for a developer who is only familiar with procedural programming to switch to an OO language.
- New language constructs are harder to learn than an external connector file; in this case AspectJ and CaesarJ were slightly more challenging than Hyper/J.
- The better a language supports modularization the more IDE support is needed to understand the result as a whole.

This was our first comparison study on this topic. The lessons we learned for further studies are:

- To compensate the variation in education and experience of test persons it is necessary to have the same test persons implement all 4 extensions if the sample is small or have a by far bigger sample (e.g. 25 per group) to get statistically valid data.
- To get comparable time measurements it is better to have a facilitator who watches the test persons and notes the time.
- For a more realistic assessment of how a language supports evolution several extension tasks with varying effect on the architecture should be performed for one application.
- Questions concerning IDE support need bigger examples and professional software developers as test persons.
- Questions falling into our general validation block need test persons with more experience than we had.

5. FUTURE WORK

In the near future we want to do a study with professionals whom we present all four solutions and ask for their evaluation of the languages. We also need measures on testability independent reusability.

6. RELATED WORK

There exists a number of studies that compare AO languages [8,9], AO with OO [10,11,12] or evaluate the suitability of AO for certain programming domains [13,14,15]. Due to space limitations we can't discuss them in this position paper.

7. ACKNOWLEDGMENTS

We thank all students who participated in our study.

8. REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, J. Irwin, “Aspect-oriented Programming”, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997

[2] P.L. Tarr, H. Ossher, W.H. Harrison, S.M. Sutton Jr., “Multi-Dimensional Separation of Concerns in Hyperspace”, In Proceedings of the International Conference on Software Engineering (ICSE), pages 107-119, 1999.

[3] AspectJ, Aspect-oriented Programming in Java, http://www.aspectj.org.

[4] Hyper/J web site: www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

[5] CaesarJ Website, http://www.caesarj.org

[6] Ramnivas Laddad. AspectJ in Action: Practical Aspect-Oriented Programming. Manning, 2003

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[8] Christina Chavez, Alessandro Garcia, and Carlos Lucena. Some insights on the use ofAspectJ and Hyper/J. In Rashid [1181].

[9] Mik Kirsten. Aop@work: Aop tools comparison, part 1: Language mechanisms. Technical report, IBM Developer Works, February 2005.

[10] Robert J.Walker, Elisa L.A. Baniassad and Gail C. Murphy, An Initial Assessment of Aspect-oriented Programming, In Proceedings of the 21st International Conference on Software Engineering,1999

[11] Alessandro Garcia, Cl´audio Sant’Anna, Eduardo Figueiredo, Uir´a Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantitative study. In Tarr [1361], pages 3–14.

[12] Shiu Lun Tsang, Siobhán Clarke, Elisa Baniassad, Object Metrics for Aspect Systems: Limiting Empirical Inference Based on Modularity, Trinity College Dublin technical report

[13] Kersten, A. and Murphy, G. Atlas: A Case Study in Building a Webbased Learning Environment Using Aspect-Oriented programming. Proceedings of OOPSLA’99, November 1999.

[14] Lippert, M. and Lopes, C.V. A Study on Exception Detection and Handling Using Aspect-Oriented Programming in Proc. ICSE 2000. Limerick Ireland. 2000.

[15] Shiu Lun Tsang, Siobhan Clarke, Elisa L. A. Baniassad, An Evaluation of Aspect-Oriented Programming for Java-based Real-Time Systems Development, ISORC 2004

	How easy was it to understand the language concept? (1=easy, 6=difficult)	How good was the documentation? (1=very helpful, 6=no help)	How hard was it to solve a problem in this paradigm? (1=very easy, 6=very hard)	Do you think this language is mature? (1=fully developed, 6=immature)
Java1	4	3	4	1
Java2	1	2	2	2
Java3	4	2	4	1
AspectJ1	2	2	4	3
AspectJ2	2	4	5	4
Hyper/J1	2	3	3	3
Hyper/J2	2	6	2	4
CaesarJ1	6	6	3	2
CaesarJ2	3	4	2	3

Table 2: Language Evaluation Results

	IDE integration	editing and browsing support	missing editing and browsing support	debugging integration
Java1	2	2	browsing support, which class calls a method	did not use
Java2	2	1	missed none	3
Java3	2	2	yes	3
AspectJ1	3 (buggy)	2	Refactoring (finish pure Java and refactor into aspects); see woven code	not tested
Aspect2J	2	2	code completion, context help (search for references)	didn't use
Hyper/J1	6	2	no, am happy with command line tool, too	didn't use
Hyper/J2	5	2 for Java, 6 for Hyper/J	support to write HyperJ files; management of slices and relationships	for java yes, for HyperJ no
CaesarJ1	4	5	code completion, browsing support, context help, refactoring support	not integrated
CaesarJ2	4	3	context sensitive help, code completion, optimized views (e.g. layered architecture view, CI/Implementation/Binding hierarchy/relationship view)	no support

Table 3: Tool Support and Debugging

	challenges/tasks to be solved more easily	want to use and recommend	scalability for industry
Java1	for developing protocol interfaces	yes	yes
Java2	no	1	1
Java3	no	don't know	can't judge
AspectJ1	could result in bad design, patching	3 (depends on relationship between effort to learn / number of potential aspects in project)	5 (results in bad design?)
AspectJ2	no	4; nice idea, but I prefer to have whole code together	2; good extensibility for existing code (extra features ..), different versions of features
Hyper/J1	for introducing new parts without knowing all existing methods	3	2
Hyper/J2	yes, e.g. logging	5	5
CaesarJ1	No	5	6
CaesarJ2	yes, use cases with layered architecture and AO requirements	5; not suitable for the industry in the current state, 2 in research area	4 not in current state (challenging developer skills and critical tool support)

Table 4: General Valuation

Modeling the Evolution of Aspect Configurations using Model Transformations

Uwe Zdun, Mark Strembeck
Institute of Information Systems, New Media Lab
Vienna University of Economics, Austria
{uwe.zdun|mark.strembeck}@wu-wien.ac.at

ABSTRACT

In this paper we introduce an approach to address the evolution of aspect configurations with model transformations. We use model transformation diagrams (MTDs) to define valid behavioral model states of a system as well as valid transitions between those states. MTD transformations can be used to define evolutionary changes in the weaving process of an aspect-oriented system. To allow for a straightforward incorporation of aspects in UML models, we extend UML2 activity diagrams with joinpoint start and end nodes. In this paper, each model state in an MTD refers to an extended UML2 activity diagram.

1. INTRODUCTION

In recent years a number of approaches for UML-based modeling of aspects have been proposed. Some approaches are extending the UML using a UML profile (see e.g. [6, 2]), others perform a meta-model extension, i.e. they extend the UML familiy of languages with new language elements (see e.g. [10, 4]). So far these approaches focus on mapping the elements of aspect-oriented environments (mainly the concepts are based on AspectJ [7]) to UML modeling elements. That is, the focus is on representing aspects in UML models.

The effects of applying aspects – i.e. how a model evolves if an aspect is woven – have only been marginally in focus of aspect-oriented modeling approaches so far. This concern, however, is important to be considered for a number of situations:

- In the early stages of system design we need to translate requirements into classes and aspects. In particular, we require some approach to show the evolution from a non-aspect-orineted model to an aspect-oriented model, as well as the interactions between the aspect-oriented and non-aspect-oriented parts of the system.

- Often a number of different aspect configurations can be woven for one and the same system. That is, the aspects woven into the system can be changed either at compile-time, load-time, or runtime – depending on the used aspect weaving mechanism. For example, consider a logging aspect, which is woven into the debugging environment only, but not into the productive system. Here, the evolution options resulting from the weaving time for the aspect configurations and their corresponding effects should be modeled as well.

- Often aspects have interdependencies or interactions among each other, a concern which of course should be modeled. For instance, consider a persistence aspect is allowed to be woven, but only if a storage device aspect is woven as well.

To address these problems, this paper proposes an approach to model the behavioral evolution of aspect configurations in software systems using model transformation diagrams. In other words, we use a model transformation to represent the aspect weaving step. The model transformation diagrams are an extension to UML 2.0. In particular, they model the aspect weaving dependencies via model transformations between different UML Activity Diagrams. Here, the Activity Diagrams show the behavior in the system with different aspect configurations. To enable the modeling of aspect-related behavior in Activity Diagrams we introduce a simple extension to Activity Diagrams for representing the start and end of the joinpoints of an aspect in the control flow.

2. THE APPROACH

In this section, we explain our model transformation diagrams, and our extension to Activity Diagrams for representing the start and end of the joinpoints of aspects.

2.1 Model Transformation Diagrams

We have defined the Model Transformation Diagrams (MTD) as a meta-model extension to the UML 2.0 standard (see Figure 1[1]). To define MTDs formally, we specify the new package *ModelTransformations*. The graphical notation of our model transformation diagrams is similar to UML2 interaction overview diagrams, however, the MTD semantics differ significantly. The UML2 interaction overview diagrams are a variant of activity diagrams and describe the flow of control between different nodes (see [9]). In contrast, our MTDs are a variant of state machines. Model transformation diagrams describe changes of specification of a software system. These changes are modeled through transitions between different diagrams. In this paper, we use only UML2 activity diagrams in the MTDs, to model transformations of the *behavioral model state*. (Please note that in our full meta-model definition, there are also structural model states, but these are not used in this paper.)

The main transition type used in MTDs are *transform transitions*. Transform transitions express that the source model state of the transition is transformed to the target model state of the transition. A transition from one behavioral model state to another means that the behavior of a certain system aspect is transformed, so that after the transition, the system behavior conforms to the state specified by the transition's target. For instance, the example transitions in Figure 2 show two model transformations between two activity diagrams: one adds a condition between the two activities, and the reverse transformations removes the condition. Figure 2 also contains

[1]Due to the page limit we do not include the full formal definition including OCL constraints of the meta-model extension here, but provide only the corresponding meta-model as an overview.

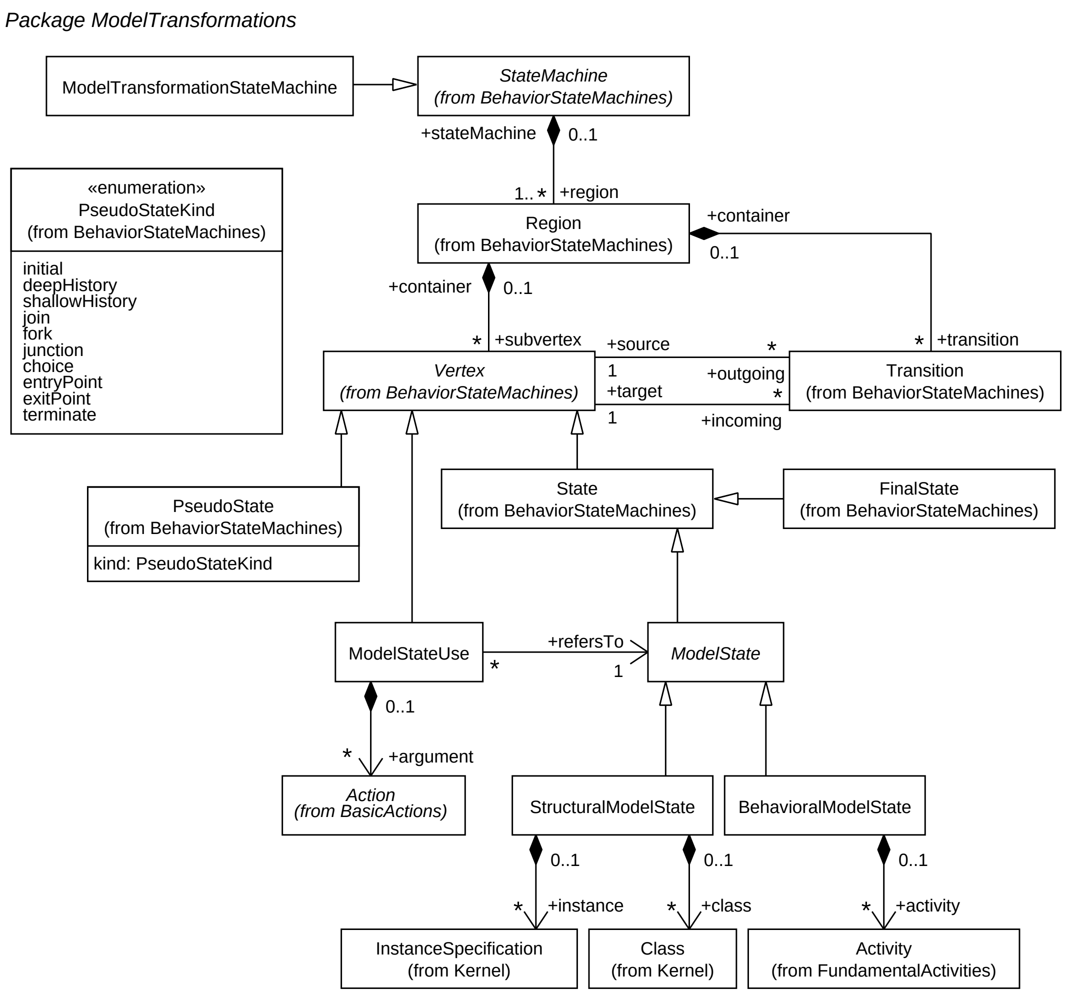

Package ModelTransformations
ModelTransformationStateMachine
StateMachine (from BehaviorStateMachines)
+stateMachine
0..1
1..*
+region
Region (from BehaviorStateMachines)
+container
0..1
«enumeration» PseudoStateKind (from BehaviorStateMachines)
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
+container
0..1
*
+subvertex
+source
*
+transition
Vertex (from BehaviorStateMachines)
1
+target
+outgoing
*
1
+incoming
Transition (from BehaviorStateMachines)
PseudoState (from BehaviorStateMachines)
kind: PseudoStateKind
State (from BehaviorStateMachines)
FinalState (from BehaviorStateMachines)
ModelStateUse
+refersTo
*
1
ModelState
0..1
*
+argument
Action (from BasicActions)
StructuralModelState
BehavioralModelState
0..1
*
+instance
0..1
*
+class
0..1
*
+activity
InstanceSpecification (from Kernel)
Class (from Kernel)
Activity (from FundamentalActivities)

Figure 1: Meta-model for Message Transformation Diagrams (MTD)

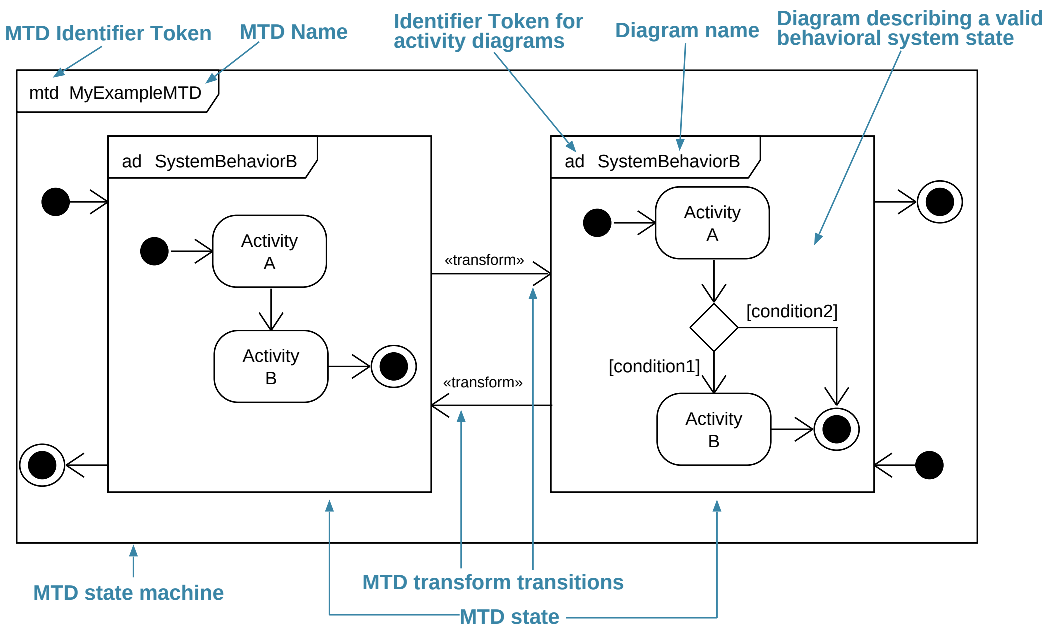


Figure 2: Informal overview for the elements of MTDs

informal explanations for our notations. A formal UML meta-model extension for MTDs can be found in [11].

In the first place, MTDs are a means to depict possible model transformations. The idea, presented in this paper, is to apply the transform transitions in the MTDs to model aspect weaving relationships. This way different behavioral model states show models of the behavior of the system in different aspect configurations. The transform transitions then show the possible ("legal") weaving steps between these model states.

2.2 Extending Activity Diagrams with Joinpoint Start and End Activities

In our approach, we model the behavior of aspects as part of the activity diagrams describing the system's behavior. That is, we show scenarios of the aspect in action. However, it is necessary to distinguish the aspect-oriented and non-aspect-oriented parts of the activity diagram. Moreover, in case more then one aspect is used, we need to distinguish different the aspects modeled in the same activity diagram.. Otherwise we would not be able to properly model aspect interactions.

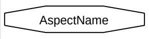
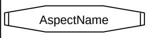
NODE TYPE	NOTATION	Explanation & Reference
JoinpointStart		JoinpointStart is an Activity that can be used in an Activity Diagram to indicate that the aspect "AspectName" has intercepted the control flow at this point. All subsequent steps in the Activity Diagram until a JoinpointEnd Activity with "AspectName" is reached are handled by the aspect "AspectName". Optionally, a Joinpoint Start node can have a tagged value "pointcut" that indicates the name of a pointcut designating this joinpoint. See Activity from FundamentalActivities.
JoinpointEnd		JoinpointEnd is an Activity that can be used in an Activity Diagram to indicate that the interception of the control flow by the aspect "AspectName" has ended. See Activity from FundamentalActivities.

Figure 3: Definition of two Activities for start and end of joinpoints in Activity Diagrams

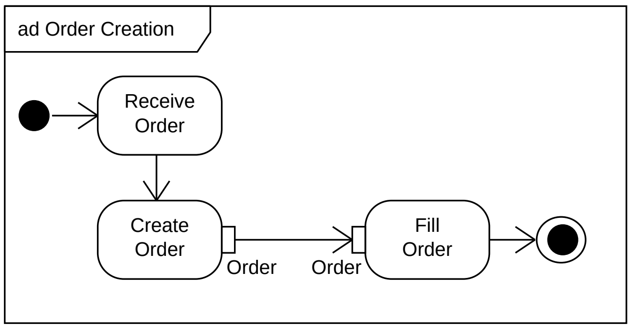


Figure 4: Activity Diagram for order creation

To address this problem, we introduce two new Activities as subclasses of the UML2 Activity meta-class (from FundamentalActivities, see [9]). JoinpointStart is an Activity that can be used in an Activity Diagram to indicate that the aspect referred to via "AspectName" has intercepted the control flow at this point. All steps in an Activity Diagram between a JoinpointStart and the corresponding JoinpointEnd Activity (referred to via the same "AspectName") are handled by the respective "AspectName" aspect. In addition it is possible for another aspect to intercept the control flow in between. In other words: JoinpointEnd is an Activity that can be used in an Activity Diagram to indicate that the interception of the control flow by the aspect "AspectName" has ended. Optionally, JoinpointStart Activities can have a tagged value "pointcut" that indicates the name of a pointcut designating this joinpoint. Figure 3 summarizes the definitions.

3. EXAMPLE: ORDER HANDLING

In this section, we consider an example from the early stages of designing an order handling system. In a first step, we design a simple activity for order creation according to the following short scenario description: when an order is received, an order object needs to be created and then the order object is filled with values. This simple control flow is shown in the activity diagram "Order Creation" in Figure 4.

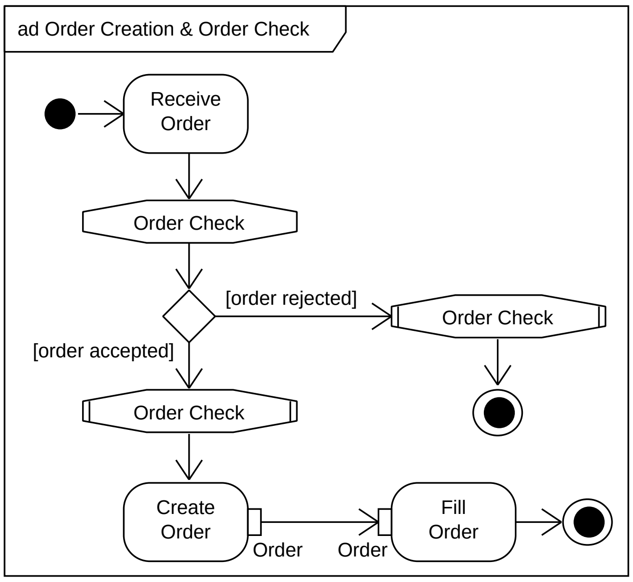


Figure 5: Activity Diagram for combining order creation with order checking

Next, we design other fundamental activities of order handling. During the ongoing design work, we realize that in some customer systems which should be used with the order handling system, a check is required, whether the order can be accepted or not. This check is not only relevant for order creation, but it must also be performed before an order is changed or re-submitted. Thus "Order Check" is a cross-cutting concern in our system and should be modeled as an aspect. To do so, we need to intercept the control flow between the Receive Order and Create Order activities. Similarly, we need to extend other activity diagrams that have joinpoints belonging to this aspect. The pointcuts for the corresponding aspect can be derived in later design stages by looking at all occurrences of the aspect's joinpoints and by defining proper (cross-cutting) designations for these points in the control flow. The woven aspect is shown in the Activity Diagram "Order Creation & Order Check" in Figure 5.

A second aspect that cross-cuts many order handling activities is "Order Persistence". This aspect needs to intercept the control flow after the order is filled in, and must call the Make Persistent Activity. The woven aspect is shown in the Activity Diagram "Order Creation & Order Persistence" in Figure 6.

For this aspect we need to consider one special case, though. If the aspect "Order Check" is configured, all rejected orders should be

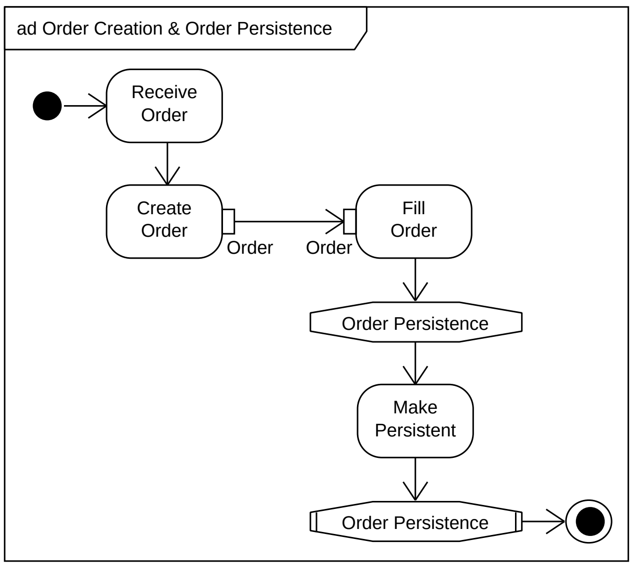


Figure 6: Activity Diagram for combining order creation with persistence

logged in the persistence store. That is, the two aspects have an interdependency among each other. Because both aspects are optional extensions, we need to model this interaction in a separate Activity Diagram "Order Creation & Order Check & Order Persistence" in Figure 7. Here, we can see that the "Order Persistence" aspect is cross-cutting the activities in this diagram. If the aspect is used, a rejected order log entry object is created, and the Make Persistent Activity is called.

Finally, we need to model the possible weaving-time aspect evolutions for this system. We use an MTD to show the possible weaving configurations for the two optional aspects described above. The diagram in Figure 8 shows that in any case the basic "Order Creation" diagram is the starting point for weaving. The aspect weaver can either weave order persistence, order checking, or no aspect. If one of the two aspects is chosen, the other aspect can optionally be woven as well. In this case, the behavioral state of the system is transformed to the Activity Diagram "Order Creation & Order Check & Order Persistence", so that the aspect interaction is modeled as well.

Please note that in this example we have shown the aspect weaving process independently of the concrete weaving time. Our approach is capable to model aspect weaving at compile-time, load-time, or runtime. Though, the MTD needs to be changed slightly if runtime weaving is supported. Runtime weaving would mean that we could turn off the aspects again. That is, we would introduce backward transformations between the model state nodes (the "mrefs" in the figure) to model runtime weaving properly.

4. RELATED WORK

Aldawud et al. [1] present a number of steps they apply to model aspect-oriented systems. In particular, they model the static system structure via class diagrams. System behavior, including aspects and crosscutting, is modeled with UML statecharts. Their approach, however, is not able to depict evolutionary changes resulting from (static or dynamic) weaving of aspects which is one of the main benefits of MTDs.

Gray et al. [3] describe an elaborated approach to support aspect-oriented domain modeling which has partially similar objectives

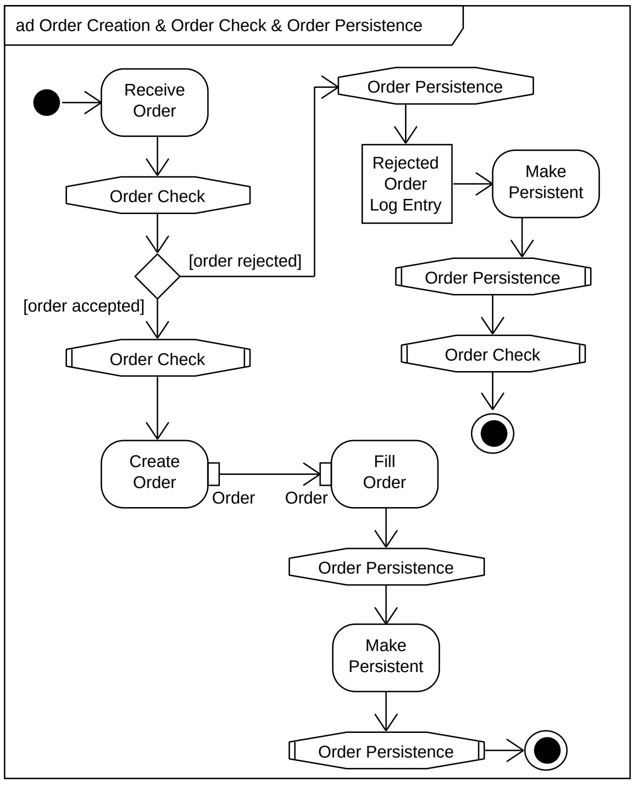


Figure 7: Activity Diagram for combining order creation with persistence and order checking

to our approach. For each modeling domain they define domain-specific weavers which operate on the abstraction layer of models (not source code). To specify these weavers they defined the embedded constraint language (ECL) as an extension to the OMG object constraint language (OCL). The ECL is used to specify transformations between models and to specify strategies that define how a concern is applied in a certain model context. ECL operates on XML files which are used to store the corresponding models and Gray et al. implemented a tool to generate C++ source code from ECL specifications.

Barros and Gomes [2] use UML2 activity diagrams to model crosscutting in aspect-oriented development. They define a new composition operation they call "activity addition" via an UML profile. Activity additions are used for weaving a crosscutting concern in a model. In particular, they define two stereotypes to mark certain nodes in activity diagrams that define the so called interface nodes which are then used to merge two or more activity diagrams, and the so called subtraction nodes which define what nodes need to be removed from a given activity diagram.

Jezequel et al. [5] represent crosscutting behavior using contract and aspect models in UML. They model contracts using UML stereotypes, and represents aspects using parameterized collaborations equipped with transformation rules expressed with OCL constraints. OCL is used in the transformations for navigating instances of the UML meta-model.

Han et al. [4] present an approach to support modeling of AspectJ language features to narrow the gap between implementations based on AspectJ and the corresponding models. Mahoney and Elrad [8] describe a way to use statecharts and virtual finite state machines to model platform specific behavior as crosscutting concerns. They

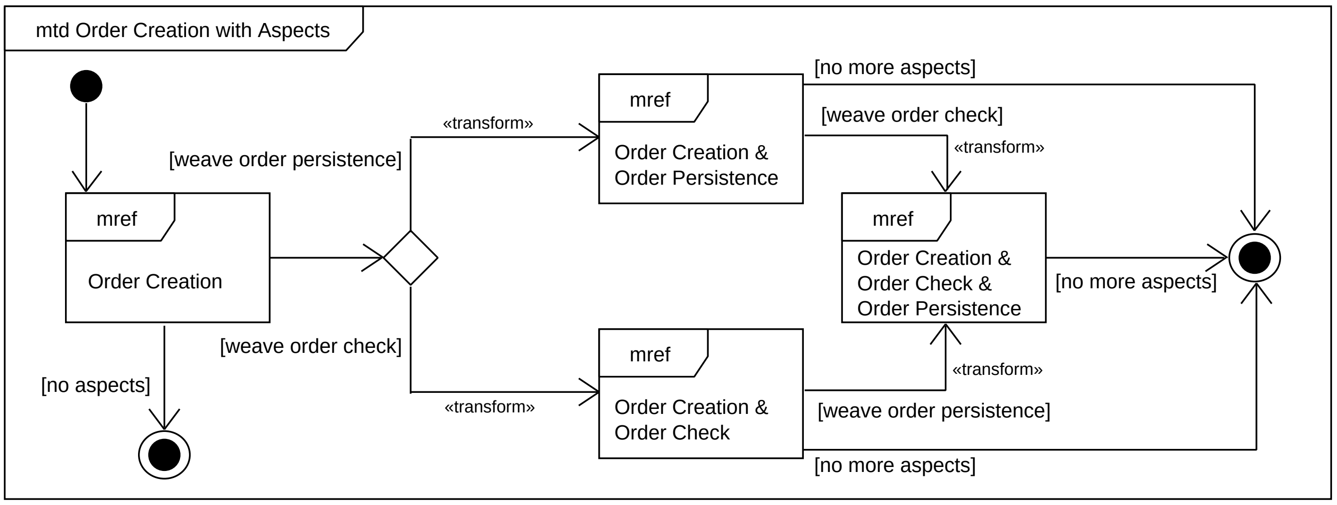

mtd Order Creation with Aspects
mref
Order Creation
[no aspects]
[weave order persistence]
[weave order check]
«transform»
«transform»
mref
Order Creation & Order Persistence
mref
Order Creation & Order Check
[no more aspects]
[weave order check]
«transform»
mref
Order Creation & Order Check & Order Persistence
[no more aspects]
«transform»
[weave order persistence]
[no more aspects]

Figure 8: MTD for order creation with its aspects

especially plan to evaluate the effectiveness of their approach in a model driven development context. Tkatchenko and Kiczales [10] present an approach to model crosscutting concerns. They extend the UML with a joint point model, advice and inter-type declarations, and role bindings. Moreover, they provide a weaver to process the corresponding extensions.

5. CONCLUSION

In this paper, we briefly presented an approach to model the evolution of aspect configurations via model transformations. In particular, we defined model transformation diagrams (MTDs) as an UML2 extension. In essence, MTDs are state machines which are applied to model the evolution of software systems. Each state in an MTD refers to a model that defines a valid structural or behavioral specification of the corresponding system. Transitions between those states describe valid transformations between those models. In this paper, however, we focused on the specification of behavioral system facets to model the evolution of aspect configurations. Therefore, we additionally introduced Joinpoint start and end activities that allow for a clear separation of the aspect-oriented and non-aspect-oriented parts of a system specification, as well as the modeling of crosscutting aspects. In our future work, we will provide tool support for MTDs both on the modeling level and source code level. In addition to behavioral states, we also use structural model states in MTDs to model the evolution of structural aspect models.

6. REFERENCES

[1] O. Aldawud, A. Bader, and T. Elrad. Weaving with Statecharts. In *Proc. of the Workshop on Aspect Oriented Modeling with UML*, April 2002.

[2] J. Barros and L. Gomes. Towards the Support for Crosscutting Concerns in Activity Diagrams: a Graphical Approach. In *Proc. of the AOSD Modeling with UML Workshop*, October 2003.

[3] J. Gray, T. Bapty, S. Neema, D. Schmidt, A. Gokhale, and B. Natarajan. An Approach for Supporting Aspect-Oriented Domain Modeling. In *Proc. of the 2nd International Conference on Generative Programming and Component Engineering (GPCE),*, September 2003.

[4] Y. Han, G. Kniesel, and A. Cremers. Towards Visual AspectJ by a Meta Model and Modeling Notation. In *Proc. of the International Workshop on Aspect-Oriented Modeling*, March 2005.

[5] J. Jezequel, N. Plouzeau, T. Weis, and K. Geihs. From contracts to aspects in uml designs. In O. Aldawud, G. Booch, S. Clarke, T. Elrad, W. Harrison, M. Kande, and A. Strohmeier, editors, *Aspect-Oriented Modeling with UML*, Enschede, The Netherlands, April 2002. http://lglwww.epfl.ch/workshops/aosd-uml/index.html.

[6] M. M. Kande, J. Kienzle, and A. Strohmeier. From AOP to UML – A Bottom-Up Approach. In *Proc. of the Workshop on Aspect Oriented Modeling with UML*, April 2002.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct 2001.

[8] M. Mahoney and T. Elrad. Modeling Platform Specific Attributes of a System as Crosscutting Concerns using Aspect-Oriented Statecharts and Virtual Finite State Machines . In *Proc. of the International Workshop on Aspect-Oriented Modeling*, March 2005.

[9] The Object Management Group. Unified Modeling Language: Superstructure. http://www.omg.org/technology/documents/formal/uml.htm, August 2005. Version 2.0, formal/05-07-04, Object Management Group.

[10] M. Tkatchenko and G. Kiczales. Uniform Support for Modeling Crosscutting Structure. In *Proc. of the International Workshop on Aspect-Oriented Modeling*, March 2005.

[11] U. Zdun and M. Strembeck. Modeling Composition in Dynamic Programming Environments with Model Transformations. In *Proc. of the 5th International Symposium on Software Composition*, Vienna, Austria, March 2006. LNCS, Springer-Verlag.

Towards Tool-supported Update of Pointcuts in AO Refactoring

Jan Wloka[*]
Fraunhofer FIRST
Berlin, Germany
jan.wloka@first.fraunhofer.de

Abstract

Aspect-oriented programming (AOP) is often introduced as an extension to a programming language. The new modularization mechanisms are provided by new language constructs, such as the pointcut and the advice. Pointcuts specify where and when an advice is executed and thereby refer to other program elements and structures to express the execution conditions. During the evolution of a program these referenced structures might be changed and hence the advice is not invoked as intended.
In this paper we present an approach for assessing the impact of source code changes on pointcuts and a program analysis that supports the identification of broken pointcuts. We elaborate how a refactoring tool can determine reasons and how an equivalent pointcut update can be calculated.

1 Introduction

Refactoring was found to improve the design of an existing software system in a safe and reliable way. It allows to modify a program's structure while it preserves its observable behavior. Especially tool-supported refactoring has become very important in software engineering for controlled software evolution. While refactoring tools for object-oriented programs are commonly used, it is not fully explored yet how the effects of even local changes on the behavior of aspect-oriented programs can be determined.

AOP introduces new modularization mechanisms to encapsulate implementations that would otherwise be spread over multiple modules. An *aspect*, as new implementation module, allows for example to adapt the behavior of other modules. Therefore, it provides new language constructs, mainly pointcut and advice, to specify where and how the program behavior should be adapted. A *pointcut* binds an advice to well-defined points in the program execution. These so called *joinpoints* are selected by specifying their properties. E.g., method call joinpoints can be selected by specifying their method's signature. Pointcuts refer to various information of a program to express a joinpoint property. An *advice* implements the additional behavior and is executed before, after or around a selected joinpoint.
If a refactoring tool should apply a structural improvement in a behavior preserving way, it must be able to determine the change impact on all pointcuts in an AO program. Otherwise, the program behavior cannot be preserved.

In this paper we present a program analysis technique for determining the impact of source code changes on pointcuts in aspect-oriented programs. We illustrate how pointcut matches can be represented to determine the change impact and discuss several factors that influence the analysis results. Based on this analysis we describe different situations for updating a pointcut and illustrate with an example how the proposed program analysis works.
The remainder of the paper is structured as follows. In section 2 the term joinpoint property is defined as the atomic part of a pointcut and examples are given. Section 3 presents some example code that is used in the following sections and applies one refactoring on it. Section 4 illustrates our approach for assessing the change impact on pointcuts, followed by a brief discussion in section 5 when and how an equivalent

[*]This work has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

update for a pointcut can be calculated. In section 6 this papers ends with some implementation details and a preliminary conclusion.

2 Specifying Joinpoint Properties

A pointcut can refer to very different kinds of program information. In general one distinguishes between static and dynamic pointcuts. A *static pointcut* uses only information that can be obtained from the program code, whereas a *dynamic pointcut* also employs runtime information.
In both cases, a pointcut specifies structural properties of a program representation that is either built from the program code (e.g., an abstract syntax tree, AST), or from runtime information during its execution (such as a stack trace). Therefore we define a joinpoint property as follows:

> *DEFINITION:* A ***joinpoint property*** is a structural property of a program representation that can be built from static or dynamic program information.

Pointcuts specify different kinds of joinpoint properties and thus refer to different information of a program. In recent publications [6, 1, 10, 2] various joinpoint properties, like naming, containment, inheritance relationships, method execution order and instance reachability in object-graphs, have been proposed for joinpoint selection in pointcuts.
From a software evolution point of view all these properties can be altered by different changes in a program's source. Two limitations need to be considered if the change impact on every property should be determined: how specific a property is defined within a pointcut and how good it can be approximated during evaluation.
Name patterns which are provided by most AOP approaches, allow a very fuzzy specification. Method executions of certain methods can be selected using AspectJ syntax[11] by expressing only some parts of the signature, like `execution(* *.foo(..))`.
In this case only the method name can be used to decide whether additional joinpoints are intended.
More recent approaches, e.g., stateful aspects in JAsCo[10] or Tracematches[1], allow to specify more "semantic" joinpoint properties[1]. They allow to specify a certain execution sequence as joinpoints, i.e., whenever this sequence is executed an associated advice will be invoked. The use of an arbitrary execution sequence causes another kind of problems. Situations in which a certain sequence appears can be approximated only in some cases and the reasons for an altered situations can rarely be determined from the static structure of a program.

[1]The term *semantic* is used here in the context of program execution, to indicate that a joinpoint property associated with the behavior that the program exhibits.

3 An Example *"Push Down Method"*

In the following, we will be using the source code example from Listing 1, which is implemented in AspectJ[3].

Listing 1: Code Example

```
package p1;
public aspect A {
    pointcut posChanged(): set(int *);
    before(): posChanged() {
        System.out.println("Changing␣
                    position");
    }
}

package p1;
public class B {
    int pos;
    static void main(String[] args) {
        C c = new C();
        c.setPos(1);
        c.update();
    }
    void setPos(int pos) {
        this.pos = pos;
    }
//  will be moved during the refactoring
    void update() {
        pos = pos + 10;
    }
}

package p1;
class C extends B {
}
```

The `main` method in `class B` creates a new instance of `class C`, sets the value of field `pos` to 1 by invoking `C.setPos(int)` and calls the method `C.update()` to increase the field `pos` by 10.
The `aspect A` intercepts the executions of any assignment to field `pos` and prints some status information to the console before the field `pos` is modified.

In a small experiment we now modify the code using the standard Java *"Push Down Method"* refac-

toring pattern (see [5] page 328). The refactoring is applied to method `B.update()` in order to move it to the subclass `C`. By moving the `update()` method also the contained joinpoint shadows are transfered to a new place.
In the following we present how the standard Java refactoring pattern influences the pointcut of `aspect A`, how this can be determined by a refactoring tool, and how a tool can calculate whether the pointcut needs to be updated.

4 Assessing the Change Impact on Pointcuts

A pointcut binds a set of joinpoints to an advice in order to have the advice executed at every bound joinpoint. Stoerzer and Graf have classified code changes that influence the program behavior defined by such advice bindings. In [9] they distinguish between three different kinds of changes:

- change of joinpoint property specifications in pointcuts
- change of implementation of the bound advice
- change of the base code which exhibits the addressed joinpoints

For brevity and because its the most difficult one, we focus in this paper on base code changes.

Representing base code changes. Very different changes can be applied to a software system. They differ in extent and complexity, and can range from a local text edit with no further impact, to huge adaptations that affects multiple implementation modules. Similar to Ryder et al. in [8, 7], we have developed an abstract change representation that consists of atomic changes.
These ***atomic changes*** represent modifications to program elements at any level of detail, such as type, field and expression. Smaller changes either can be ignored, or are represented as a property of a specific program element. More complex base code changes are then composed of atomic changes. This leads to several advantages and simplifies the change analysis. If we consider the *Push Down Method Refactoring* in our example code the change representation would look like the change tree shown in Figure 1. The tree contains every changed program element as well as its parents. Each element is associated to an atomic

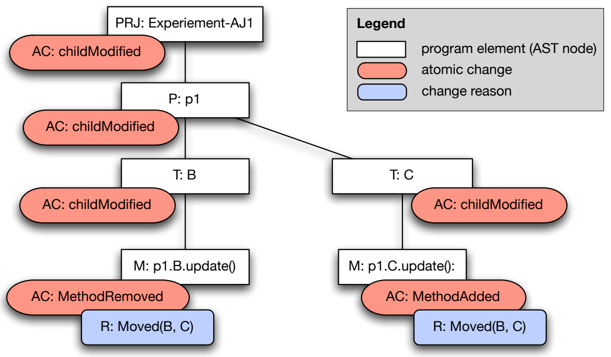


Figure 1: Atomic Change Tree representing the change for *Push Down Method*

change that stores the change reason. For the example, the tree in the figure shows the removal of method `update()` in `class B` and the addition of the method in `class C`. The associated reason indicates that both changes are caused by this method move.

Calculate a pointcut's selection. A pointcut specifies joinpoint properties to bind an advice to a set of joinpoints, i.e., an associated advice is bound to every *pointcut match*. A pointcut, however, may also refer to other program elements that are used to express a certain joinpoint property. For example, the pointcut `call(public void p1.B.update())` would refer to the fully qualified name of `class B`, even if the actual joinpoint may be located somewhere else. For the specification of every joinpoint property a pointcut selects other program elements. In the following, we distinguish between *partial pointcut matches* (only a subset of the joinpoint properties) and *complete pointcut matches* (all joinpoint properties). Based on this distinction we define a pointcut selection as follows:

> *DEFINITION:* A ***pointcut selection*** contains the mapping of partial or complete matches to the matched program elements as well as all program elements referenced by the pointcut.

In other words, a pointcut selection holds every information of a certain program that is needed to determine the pointcut's matches, including every partial match.
Figure 2 shows the pointcut selection for `set(int *)` of our example program. It holds all program elements that are directly referenced by the pointcut,

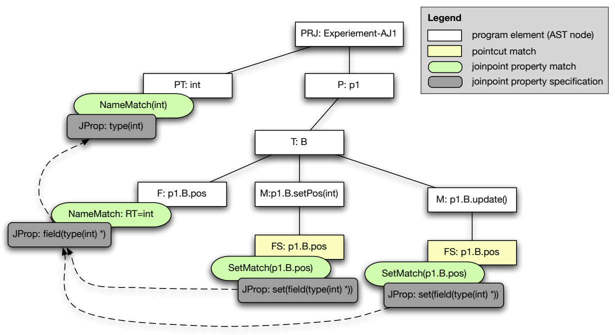


Figure 2: Pointcut Selection for `pointcut set(int *)` before the change

including their parent nodes, before the refactoring is applied.

In our example the pointcut would match at the field assignments in `setPos(int)` and `update()` of `class B`, as shown in the selection tree. Additionally both `SetMatch` matches refer to the field `pos` with one `NameMatch` match which in turn references the primitive type `int`.

Determine the change impact. The *Push Down Method Refactoring* is virtually applied to get the change information. For the modified program a second pointcut selection is calculated for every pointcut in the system. Both versions are compared and a ***pointcut selection delta*** is produced. This delta contains all partial and complete pointcut matches that aren't equal in both versions. In this way the impact of a source code change can be represented in terms of a pointcut.
For our code example the pointcut selection delta is shown in Figure 3. It contains two matches (one added and one removed) as well as all enclosing and referenced elements. The `SetMatch` matches at both field sets `FS` of `pos` are indicated as modified, because the matches are identified by a fully qualified name[2].

5 Propose a Pointcut Update

The overall goal in a refactoring process is to keep the set of selected joinpoints semantically equal. In particular, this means a pointcut is updated in a way that it matches the same program elements as be-

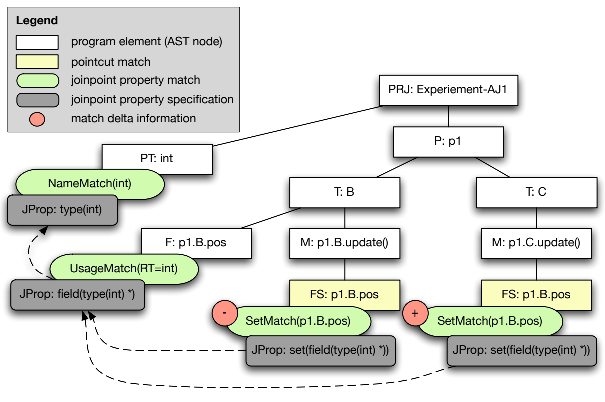


Figure 3: Pointcut Selection Delta for `pointcut set(int *)`

fore[3]. In the following we briefly discuss different update situations.

No update required. We've identified three situations in which a pointcut might be affected but doesn't need an update.

- *Unaffected pointcut:* The pointcut selection delta is empty, i.e., all program elements referenced by this pointcut are not effected by the change.
- *Unaffected pointcut matches:* The pointcut selection delta is not empty, but no complete pointcut match can be found in the delta. I.e., all matches in the delta are partial and thus not relevant for the advice execution. In such cases the pointcut refers to affected program elements but the resulting set of pointcut matches is not altered.
- *No affected joinpoint property specification:* The delta contains added and removed (complete) pointcut matches. However, for every removed pointcut match an equivalent added match can be found. This seems often the case when some program elements were moved around, but its actual location isn't specified in the pointcut.

Update required. If some complete pointcut matches have been added or removed, an update of the actual pointcut is necessary in order to ensure behavior preservation. Updating a pointcut means

[2] Statement level pointcut matches may have an ambiguous fully qualified name. For a unique identification their number of appearance in the enclosing block is added to the name.

[3]Since pointcut matching is done on program representations, a pointcut can match either at a program element itself (static) or at the execution of the program element (dynamic).

updating every specification of an added or removed joinpoint property match. For every altered pointcut match, the referenced joinpoint property matches are analyzed. If an equivalent joinpoint property specification can be calculated, an update is proposed. Otherwise the update is unresolvable. Both these cases as well as relevant causes are briefly discussed in the following:

- *Resolvable Update:* If the affected joinpoint property specification can be directly mapped to a base code change, then the refactoring together with this change are sufficient to derive an updated joinpoint specification. Interestingly, this often seems to be the case when program elements are renamed, extracted, inlined or moved.

- *Unresolvable Update:* Until now we have some possible reasons identified in which our approach cannot provide sufficient impact information.
  Joinpoint property removal: A source code change removes a program element or a structure that is directly referenced by a pointcut. A new joinpoint property needs to be found to select the joinpoint.
  Fuzzy joinpoint property meaning: Some pointcuts specify only very few concrete information, e.g., together with name patterns. A name pattern can partially specify the name of a program element, such as `call(* * get*(..))`. If an additional match with such a name is detected a tool has only the three letters 'set' to decide whether the new match is valid.
  Insufficient approximation of dynamic pointcut matches: Our approach is based on static program analysis. Therefore, all dynamic joinpoint properties needs to be approximated, i.e., actually the employed program representation. For example, used of runtime values such as in `if()` pointcuts cannot be approximated at all.
  Modified aspect interaction: If the proposed pointcut update would refer to a joinpoint that is also selected by another pointcut, a tool couldn't decide in which order the advices should be performed.
  In such situations the developer at least should be provided with sufficiently detailed information about identified conflicts or limitations.

Pointcut update calculation. Figure 4 presents the complete change impact tree for the refactoring in our example. In this case all affected pointcut matches can be directly mapped to a base code change of an enclosing element, i.e., no pointcut match (not even a partial match) is affected by the refactoring. In this case the pointcut would be left unchanged, because no program information referenced by a pointcut is affected.

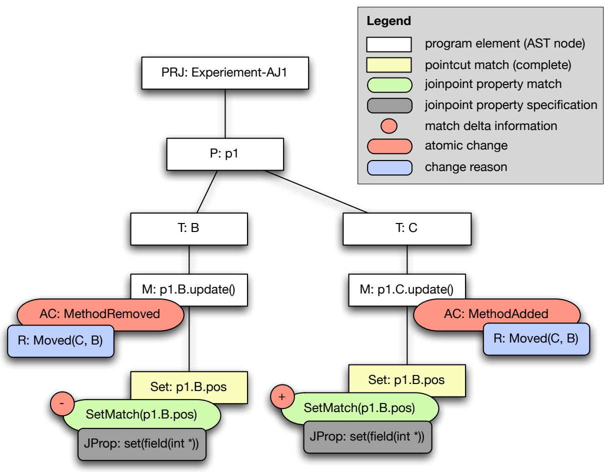


Figure 4: Change impact representation for `pointcut set(int *))`

6 Conclusions and Future Work

We have shown how static program analysis can be used to determine the change impact on pointcuts and how the impact can be represented in order to calculate an adequate update proposal within a refactoring process. We have briefly discussed different factors that influence the calculation of a pointcut update and illustrated our approach with an example.
The program analysis approach presented in this paper has been implemented by a small analysis framework called "Soothsayer". The implementation is part of the ObjectTeams/Java IDE[4] and based on Eclipse data structures. In previous work some Java refactoring patterns have been adapted for ObjectTeams/Java that were not concerned about pointcuts. The Soothsayer framework aims to extend this implementation in order to cope with pointcuts during a refactoring's application. Currently only of few kinds of joinpoint properties are supported, like name patterns, containment and inheritance relationships.

First experiences have shown that our analysis sufficiently represents the impact information, so in several situations an adequate pointcut update can be proposed. In the near future we are going to test our program analysis with more kinds of joinpoint properties and improve the calculation of pointcut updates, to consider more situations for resolvable updates. Soothsayer isn't currently really integrated with the refactoring capability of Eclipse, so the analysis results cannot be used to apply the actual pointcut updates. We are also going to do that.

Acknowledgements

Thanks to Thomas Dudziak for providing valuable comments on an earlier draft of the paper.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhotk, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching to AspectJ. Technical Report abc-2005-1, Programming Tools Group, University of Oxford University, UK; BRICS, Group of Aarhus, Denmark; Sable Research, McGill University, Montreal, Canada, 2005.

[2] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, Lecture Notes in Computer Science, pages 366–382, Taipei, Taiwan, November 2004. Springer-Verlag Heidelberg.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[4] Object Teams home page. http://www.ObjectTeams.org.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[6] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Olaf Spinczyk, Andreas Gal, and Michael Schoettner, editors, *ECOOP Workshop on Programming Languages and Operating Systems*, 2004.

[7] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 664–665, New York, NY, USA, 2005. ACM Press.

[8] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM Press.

[9] Maximilian Störzer and Jürgen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM*, pages 653–656. IEEE Computer Society, 2005.

[10] System and Software Engineering lab (SSEL) at the Department of (Applied) Computer Science (Faculty of Sciences) at Vrije Universiteit Brussel (VUB). *JAsCo Language Reference.* available from `http://ssel.vub.ac.be/jasco/documentation:main`.

[11] Xerox Corporation. *AspectJ Programming Guide.* available from `http://eclipse.org/aspectj`.