Proceedings of the 2nd Workshop on Assessment of Contemporary Modularization Techniques (ACoM.08)

Phil Greenwood, Alessandro Garcia, Eduardo Figueiredo
Lancaster University, UK
{greenwop, garciaa, figueire}@comp.lancs.ac.uk

Yuanfang Cai
Drexel University, USA
yfcai@cs.drexel.edu

Kevin Sullivan
University of Virginia, USA
sullivan@cs.virginia.edu

Elisa Baniassad
The Chinese University of Hong Kong, China
elisa@cse.cuhk.edu.hk

Alan MacCormack
Harvard Business School, USA
amaccormack@hbs.edu

http://www.comp.lancs.ac.uk/computing/ACoM.08/

Nashville, TN, 19 October 2008, USA
Co-located with ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)
http://www.oopsla.org/oopsla2008/

Technical Report COMP-004-2008
October 2008
Computing Department
InfoLab21
South Drive
Lancaster University
Lancaster LA1 4WA UK

Workshop Organisation

Program Committee

- Mehmet Aksit, University of Twente, The Netherlands
- Sven Apel, University of Passau, Germany
- Paulo Borba, Federal University of Pernambuco, Brazil
- Yvonne Coady, University of Victoria, Canada
- Marc Eaddy, Columbia University, USA
- Eduardo Figueiredo, Lancaster University, UK
- Rachel Harrison, Stratton Edge Consulting, UK
- George Heineman, Worcester Polytechnic Institute, USA
- James D. Herbsleb, Carnegie Mellon University, USA
- Arno Jacobsen, University of Toronto, Canada
- Gregor Kiczales, University of British Columbia, Canada
- Christa Schwanninger, Siemens AG, Germany
- Peri Tarr, IBM Watson Research Center, USA
- Robert Walker, University of Calgary, Canada

Organising Committee

- Phil Greenwood, Lancaster University, UK
- Alessandro Garcia, Lancaster University, UK
- Elisa Baniassad, The Chinese University of Hong Kong, China
- Kevin Sullivan, University of Virginia, USA
- Yuanfang Cai, Drexel University, USA
- Alan MacCormack, Harvard Business School, USA
- Eduardo Figueiredo, Lancaster University, UK

Introduction

A number of new modularization techniques are emerging to cope with the challenges of contemporary software engineering, such as Aspect-Oriented Software Development (AOSD), Feature-Oriented Programming (FOP), and the like. The effective assessment of such technologies plays a pivotal role in (i) understanding of their costs and benefits when compared to conventional development techniques, and (ii) their effective transfer to mainstream software development.

Workshop Goals

The main goal of this workshop is to put together researchers and practitioners with different backgrounds in order to discuss open issues on the assessment of contemporary modularization techniques, such as:

1. How do new modularization techniques affect working practices and help with software development and evolution? What guidelines can be established from assessment results to improve working practices?
2. What is the impact of using conventional quantitative metrics to assess software modularity? Are they effective enough to assess contemporary modularity techniques? How can we validate assessment mechanisms?
3. To what extent does assessment depend on extensive experience in practice? How can observations of practitioners help in assessment of contemporary modularization techniques?
4. What are the potential paths leading to more effective modularization techniques?
5. How can we compare these modularization techniques, reconcile their seemingly different appearance, and synthesize their applications to design software more effectively?

Topics of Interest

The workshop is intended to cover a wide range of topics, from theoretical foundations to assessment frameworks and empirical studies involving contemporary software modularity techniques. Topics of interest include the following (but not limited to):

- Lessons learned from assessing new modularization techniques
- Empirical studies and industrial experiences
- Comparative studies between new modularization techniques and conventional ones
- Assessment frameworks
- Software metrics and estimation models
- Validation of assessment techniques and mechanisms
- Assessment techniques, methods and tools to different phases of the software lifecycle
- Development of predictive models of defect rates and reliability from real data
- Infrastructure issues, such as measurement theory, experimental design, and analysis approaches
- Improvement of modularization techniques based on assessment.

Table of Contents

Session I - Contemporary Metrics for Assessing Modularity and Session II - Assessment in Software Evolution	
Measuring Software Design Modularity **Yuanfang Cai**, **Sunny Huynh**	Page 1
Toward Probabilistic Assessment of Modularity **Kevin Hoffman**, **Patrick Eugster**	Page 3
Assessing Modularity of Feature Models with ACNs **Kanwarpreet Sethi, Sunny Huynh, Yuanfang Cai**	Page 5
Session II - Assessment in Software Evolution	
Measuring Design Volatility Against Design Rule Stability **Yuanfang Cai**, **Kanwarpreet Sethi**	Page 7
On the Assessment of Pointcut Design in Evolving Aspect-Oriented Software **Raffi Khatchadourian**, **Phil Greenwood, Awais Rashid**	Page 9
Assessing the Malleability of Modular Design **Giuseppe Valetto**	Page 11
Session III - Experience in Empirical Studies	
Using Metadata in Aspect-Oriented Frameworks **Eduardo M. Guerra**, **Jefferson O. Silva, Fábio F. Silveira, Clóvis T. Fernandes**	Page 13
Mining Software Repositories for Evaluating Software Engineering Properties of Language Designs **Hridesh Rajan**	Page 19
Evaluating the Efficacy of Concern-Driven Metrics: A Comparative Study **Claudio Sant'Anna**, **Alessandro Garcia, Carlos J. P. Lucena**	Page 25
Session IV - Taxonomies for Software Evaluation	
Assessing Contemporary Modularization Techniques for Middleware Specialization **Akshay Dabholkar**, **Aniruddha Gokhale**	Page 31
A Close Look at Composition Languages **Florian Heidenreich, Jendrik Johannes,** Steffen Zschaler**, Uwe Aßmann**	Page 38
Towards a Framework for Guiding Aspect-Oriented Software Maintenance Empirical Studies **Marcelo Moura, Sergio Soares**, **Fernando Castor Filho, Mario Monteiro, Phil Greenwood, Alessandro Garcia, Elliackin Figueiredo, Diego Araujo**	Page 44

Measuring Software Design Modularity

Yuanfang Cai, Sunny Huynh
Department of Computer Science
Drexel University
Philadelphia, PA, 19104
yfcai@cs.drexel.edu

Abstract

Common metrics of design modularity include coupling, cohesion, separation of concerns, etc. Large number of modularity assessment work has been done at the level of source code as retrospective empirical studies, but few at design level. In addition, these metrics are limited in terms of showing how tasks can be assigned to independent groups, and how the design enables parallel development. In fact, the term *module* used in traditional modularity assessment is not always consistent with Parnas's canonical definition: independent task assignment. In this paper, we propose to measure modularity in design in term of independent task modules. The idea is to use design structure matrices to represent a design, automatically cluster the DSM to reveal independent modules, and assess to what extent the design can be independently implemented or changed.

Keywords *Software Design, Modularity, Metrics*

1. Introduction

Software developers usually refer functions, classes, or components as modules, and large number of work has been done to measure the coupling and cohesion among modules defined this way [3][4]. Modern programming paradigms, such as OOD, AOP, and FOP extend the definition of modules to include new program constructs, such as aspects, and researchers have conducted numerous empirical studies to assess the impact of different programming paradigms on software modularity [2][3][4].

We observe two issues in prevailing definition of *module* and modularity assessment methods. First, Parnas proposed a widely recognized definition of a *module*, that is, an independent task assignment [5]. Due to their interdependencies, functions, classes, or aspects do not capture how a design can be decomposed into independent task assignments that can be accomplished in parallel. Second, prevailing metrics, such as coupling, cohesion, and separation of concerns, are usually used to assess source code, but not the design that precedes and dominates coding. Finally, how well a design can support independent and modularized implementation and evolution can not be measured effectively by these commonly-used metrics. For example, the prevailing separation of concern assessment measures how concerns are spread in code, but not to what extend the design can support parallel evolution.

We propose to develop a new metric, based on *design structure matrix* (DSM), to measure the extent to which a software design can support independent and parallel implementation and evolution. We use Parnas's definition of modules, independent task assignments, and show how a module can be captured in a DSM. We also propose to develop a method to automatically cluster a DSM so that independent modules can be manifested automatically. We will introduce the definition of DSM, modules, and the new modularity metric in the next section.

Parnas's definition of module can be generally applied regardless of programming paradigms in use, and can be used in both design and implementation. Similarly, the modularity metric we are proposing can also be generally applied in various programming paradigms, traditional or contemporary. The DSM modeling spans the boundary of design and implementation because both can be modeled simultaneously in the same DSM model. As a result, the derived modularity metric can be used to assess how well a design can support independent task assignment, how its implementation are decomposed, how the design tasks and implementations tasks can be assigned together, etc. We use a small example to illustrate the definitions and the idea.

2. An Example

This section introduces how independent modules can be captured by a DSM, and the new modularity metric. Figure 1 shows a DSM modeling a maze game design using an Abstract Factory Pattern. A DSM is a square matrix in which the columns and rows are labeled with design variables and a marked cell models that the design decision on the row depends on the decision on the column.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
MapSite_interface	1	.																							
Maze_interface	2		.																						
Door_interface	3	x		.																					
DoorNeedingSpell_interface	4	x		x	.																				
MapSite_impl	5	x				.																			
Room_interface	6	x					.																		
EnchantedRoom_interface	7	x					x	.																	
RoomWithABomb_interface	8	x					x		.																
Room_impl	9	x				x	x			.															
Wall_interface	10	x									.														
MazeFactory_interface	11											.													
MazeFactory_impl	12	x	x	x			x				x	x	.												
BombedWall_interface	13	x									x			.											
EnchantedRoom_impl	14	x				x	x	x		x					.										
BombedWall_impl	15	x				x					x			x		.	x								
Wall_impl	16	x				x					x						.								
EnchantedMazeFactory_impl	17	x	x	x	x		x	x			x	x	x					.	x						
EnchantedMazeFactory_interfa	18											x							.						
RoomWithABomb_impl	19	x				x	x		x	x										.					
BombedMazeFactory_interface	20											x									.				
BombedMazeFactory_impl	21	x	x	x			x		x		x	x	x	x							x	.			
DoorNeedingSpell_impl	22	x		x	x	x	x																.	x	
Door_impl	23	x		x		x	x																	.	
Maze_impl	24	x	x				x																		.

Figure 1: Maze Game DSM

Previous work has shown that an important concept in software design is the notion of *design rules*[1][6]. A design rule is a stable design decision that serves as an interface that decouples two previously coupled decisions. In a DSM, design rules can be modeled as variables that precede sublimating decisions, and there are no dependencies from design rules to subordinating decisions. In Figure 1, the gray area is the impact area of design rules. That is, variables 1 to 13 are interfaces that are visible to more than one subordinating design decisions.

Given the definition of DRs, we can define a module as an independent block along the diagonal in a DSM. By "an independent block", we mean a set of design variables that only depend on design rules, but not on other independent blocks. In Figure 1, the modules are shown in rows/columns 14-24. The dark boxes indicate modules. For example, variables 17 (EnchantedMazeFactory_impl) and 18 (EnchantedMazeFactory_interface) are aggregated into one module, which means that the design and implementation of this module can be done independently as long as the design rules are stable.

The above definitions allow us to define a new modularity metric: the level of independence (LI). We define LI as the following: LI = Independent Module Size / Overall Size. The rationale is that the more variables in independent modules, the larger part of the system can be evolved independently. The LI for the maze game design shown in Figure 1 is 0.46. We have used this metric to compare the two Key Word in Context designs Parnas presented in his seminal paper [5]. The LI for the information hiding design is 60%, and the LI for the sequential design is 44%, showing that the information hiding design is better modularized.

3. Conclusions and Discussions

This paper presents a new modularity metric, the Level of Independence (LI), based on Parnas's definition of module and design structure matrix modeling. We preliminarily assessed several canonical and design patterns, showing that this metric has the potential to compare designs in terms of modularity concisely and precisely.

Baldwin and Clark's net option value (NOV) analysis provides a more comprehensive ways to assess design modularity. Our metric is consistent with NOV analysis in that the higher the LI, the more independent modules exist, and hence more option values can be generated .Our metric is simpler and easier to use in that computing LI does not require the user to estimate vague parameters required by NOV analysis, such as technical potential.

The metric needs to be further developed to account for the sizes of each individual module. For example, it is possible that large number of variables is aggregated into one big but independent block that needs to be further decomposed. On the other hand, if such a big blob exits as an independent module, then the user can visualize it immediately from the DSM, making it unnecessary to rely on a number to detect its existence.

The key enabler of this metric is to correctly clustering a DSM so that independent modules can manifest themselves. We are developing an algorithm to automatically cluster a DSM to manifest its independent modules. The DSM in Figure 1 is automatically generated from our prototype tool that implements the algorithm.

References

[1] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.

[2] Cai, Y., Huynh, S., and Xie, T. A framework and tool supports for testing modularity of software design. *ASE 2007*, pages 441–444, November 2007.

[3] Figueiredo, E. et al.: Evolving software product lines with aspects: An empirical study on design stability. ICSE 2008.

[4] Garcia, A. et al. Modularizing design patterns with aspects: a quantitative study. AOSD 2005, pages 3–14.

[5] Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, December 1972.

[6] Sullivan, et al. The structure and value of modularity in software design. In *ESEC/FSE '01*, pages 99–108.

Towards Probabilistic Assessment of Modularity

Kevin Hoffman Patrick Eugster

Purdue University, Department of Computer Science
305 N. University St., West Lafayette, IN 47907
kjhoffma@cs.purdue.edu peugster@cs.purdue.edu

1. Introduction

With recent trends showing increasing rates of software evolution and complexity, modularity is more important than it ever has been for on-time, on-budget software development. Assessing modularity is important for both evaluating current software quality and planning future changes. This latter use case is becoming more relevant as open-source models become more common and developers contribute to code with which they may be quite unfamiliar.

Software engineers tasked with implementing specific changes to unfamiliar software might ask questions such as:

Q1 Which program elements are the most depended upon across the system? ... within a component?

Q2 Which program elements are the most fragile (likely to change)?

Q3 Which program elements are most likely to be affected by ripple effects given a set of changes?

Q4 Which program elements were most influential during some execution of a test case?

Traditional modularity metrics do not provide immediate answers to these questions. To assist in answering such questions, we propose new modularity metrics based on probabilistic computations over weighted design structure matrices (DSMs) [1], also known as dependency structure matrices. We define how these metrics are calculated, discuss some practical applications, and conclude with future work.

2. Probabilistic Analysis of Modularity

DSMs are an established technique for assessing the modularity of a software system [7, 8]. DSMs model the pairwise dependencies between design elements and program elements in matrix format. Both rows and columns are labeled with these elements. Each matrix cell models the degree to which the element corresponding to that row depends on the element corresponding to that column. Clustering and partitioning algorithms can be applied to organize dependencies within the system. Tools such as LDM [7] generate DSMs by examining static dependencies within source code. In LDM dependency weights can be counts of individual dependencies (e.g., number of method calls, class instantiations, etc.) or can be knowledge based (such as the number of classes referenced).

Our goal is to extrapolate from these individual pairwise dependencies more information about the dependent structure of the system as a whole. We propose that the DSM can be used to form a probabilistic model measuring system-wide dependency, termed the *Dependency Propagation Ranking,* as follows: Define the *Assessment Graph* as a weighted directed graph in which the vertices are design or program elements and the edges represent the probability of the source element affecting the target element if the source element changes.

These edges are derived from the dependency information in the DSM, such that if element A depends on element B in the DSM, then an edge will be created from vertex A to vertex B in the Assessment Graph. The weight of the edge is a function of both the corresponding weight in the cell of the DSM and a normalization algorithm. Note that there never is more than one edge between any given pair of vertices. This allows the graph to be represented as a *weighted edge matrix*, where the value of the cell at row i and column j is the weight of the edge from i to j or 0 if there is no such edge.

The weighted edge matrix, M, of the Assessment Graph can represent the stochastic matrix of a time-homogeneous Markov chain if the following two conditions are met:

1. Normalized edge weights: For each vertex i, the sum of all of the probabilities moving from vertex i to other vertices must be exactly 1. In other words, the values must sum to 1 for each row in the matrix. This is accomplished by applying a normalization algorithm, with the simplest one being $M'_{i,j} = M_{i,j} / \sum_k M_{i,k}$.
2. All vertices must have at least one edge: In our graph construction algorithm above, elements that did not depend on any other elements would have zero edges in the graph and thus be sinks. This can be fixed by adding edges from each sink vertex to every other vertex in the graph, giving weight $1/|M|$ to each edge. This does not affect the metric values produced from the probabilistic model [2].

This Markov chain now represents the probabilistic propagation of dependencies within the system. A full armament of probabilistic analyses can be applied to the Markov chain.

One such analysis is the calculation of the stationary distribution of the Markov chain, which would provide a global ranking of dependency (such that components that are more highly depended upon are ranked higher). This is analogous to the representation and calculation of global web page ranking in PageRank [6] and of global trust values in EigenTrust [4]. Computational calculation is straightforward and proceeds by iteratively calculating the principle left eigenvector of M. Care must be taken to ensure that the calculation will converge. Applying a *damping factor* in the iterative calculation ensures convergence, and we postulate that it will not unduly affect the usefulness of the results, as was the case in [6, 4]. Dependency Propagation Ranking can help answer motivating question Q1.

The same algorithm can be used to calculate other probabilistic metrics by (a) varying how weights in the input DSM are generated, (b) considering only certain types of dependencies when generating the DSM, (c) varying what vertices in the Assessment Graph represent, (d) changing how Assessment Graph edges are constructed from the DSM, and (e) adjusting how weights are normalized. By inverting edge direction in the Assessment Graph, we can model the probabilistic propagation of changes in the system and rank those components most likely to be affected by changes, termed Impact Propagation Ranking, helping with Q2.

Also, we define the *Impact Shift Ranking* as the difference between the Impact Propagation Ranking values for two versions of a program, allowing developers to immediately understand the relative impact on ripple affects for a set of changes (Q3). Finding a way to 'un-normalize' the Impact Propagation Ranking values would allow us to define Aggregate Impact as the total of all un-normalized Impact Propagation Ranking values, as a measure of total system susceptibility to ripple effects.

Information from static analyses, such as a data flow analysis, could be used to create weighted DSMs, allowing for measurement of the propagation of indirect coupling. Forming DSMs constructed from execution trace data rather than source code or static analysis is also envisioned, allowing one to examine system-wide modularity properties as exhibited during runtime (helping with Q4).

Altering the normalization function could be useful in focusing the scope of the analysis on certain program elements. For example, instead of normalization evenly diving the weights, it could give greater weight to those program elements with the scope of interest. For example, the normalization function could give greater weight to dependencies outside of a program element's package or partition, emphasizing the propagation of dependencies between separate modules in the system.

Once a DSM of the software is constructed, the above metrics can be computed quickly, allowing for the possibility of an interactive style exploration of the probabilistic metrics both system-wide and within components and individual packages or even classes.

3. Related Work

Net option value analysis has been applied to DSMs [5, 8] to quantify the global value of modularity already built in to the system. It compliments our suite of metrics proposed herein, which focus on understanding the actual modularity of the system, rather than the potential benefits of modularity.

4. Future Directions

We have outlined new modularity metrics based on probabilistic models, but these metrics have yet to be validated or evaluated on real programs. The metrics need to be more precisely defined, and the nuances of their definitions explored, especially for programs with distinct disconnected components. Other sources for the DSM weights should be considered, such as logical coupling [3]. A full case study across multiple applications is needed in order to gain insight into how effective these metrics are for answering our motivating questions. We need to verify that utilizing a damping factor to ensure convergence does not cause major bias in the metric values. We anticipate exploring how to generate DSMs based on static analyses and runtime execution traces. In conclusion this paper is a first step towards probabilistic metrics for measuring system-wide (and subsystem-wide) modularity properties.

References

[1] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, 1999.

[2] Monica Bianchini, Marco Gori, and Franco Scarselli. Inside PageRank. *ACM Transactions on Interet Technology*, 5(1):92–128, 2005.

[3] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[4] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *WWW '03*, pages 640–651, 2003.

[5] Cristina Videira Lopes and Sushil Krishna Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05*, pages 15–26, 2005.

[6] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Technical Report 1999-66, Stanford University, 1999.

[7] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *OOPSLA'05*, pages 167–176, 2005.

[8] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9*, pages 99–108, 2001.

Assessing Modularity of Feature Models with ACNs

Kanwarpreet Sethi Sunny Huynh Yuanfang Cai
Drexel University
{kss33, sunny, yfcai}@cs.drexel.edu

1. The Problem

Feature oriented programming (FOP) is an emerging approach to simplifying construction of software in product lines (Prehofer 1997). FOP works by developing the base program for the common part of a product line and individual features for each variation. After that, the base program code is automatically merged to produce a fully functional software system. In Model Driven Development (MDD), programs are defined in high level domain specific modeling languages and automatically transformed into code for the target platform. Trujillo et. al. combined FOP and MDD into FOMDD (Trujillo et al. 2007) so that code can be automatically generated for individual features by specifying high level models and automatically compose the individual features together to form a whole product line.

The problems is that since the features are developed independently of each other and automatically composed together, modularity of the resulting composed code cannot be guaranteed. The modularity of these products may be affected further by duplicated code in the individual features. This duplication not only increases the size of the executable and increases build times (Liu and Batory 2004) but also adversely affects modularity. With the advent of Model Driven Development, there is a need to assess the modularity of design models before they are transformed to source code to identify potential issues with modularity of the end products.

2. Our Idea

Our idea is to formalize the transformation of a software product line FOP model using a high-level design model that can be used to assess design modularity. After that, instead of using traditional coupling and cohesion metrics, we plan to use net option value (NOV) (Baldwin and Clark 2000) to assess the modularity of the resulting system.

The reasons to model and assess the transformation of FOMDD, instead of assessing the source code modularity after the source code is automatically generated, are as follows. First, after the source code is generated, the code that was cloned, shared or extended become undistinguishable at the source code level. As a result, the decisions about which parts represent commonality, and thus should remain stable, and which parts reflect variations that can be changed independently, are all blurred. Second, the higher-level model should be more abstract and smaller in scale, so that the user can understand system modularity without looking into the code in detail.

We plan to NOV analysis instead of coupling and cohesion metrics because the NOV model captures how modularity in design creates values, the essence of software product lines. Modularity has been shown to add value to software in terms of real options (Baldwin and Clark 2000) as it allows for investment of a superior replacement for a module or keep the current implementation, if thats the better choice. Hence, in a product line, modularity allows sustenance of the architecture and design for ease of maintenance and growth of the product line in terms of features supported.

According to Baldwin and Clark, *design rules* decouple a system, and create independent *modules*. Design rules are stable design decisions that decouple otherwise coupled decisions. The analogous of design rule in a product line is the commonality among features.

As long as the shared common parts remain stable, the independent features provide a portfolio of options that can be substituted with better versions. We believe that NOV can be a better metric for product lines.

3. The Approach

Our propose to capture FOMDD model transformation using an *Augmented Constraint Networks* (ACN) (Cai et al. 2007), which can be used to automatically generate a *Design Structure Matrix* so that the modular structure of a system can be visualized, and NOV value can be calculated (Cai et al. 2007).

The core of an ACN is a constraint network that consists of variables representing design decisions and logical constraints representing the dependencies among design decisions. An ACN also contains a *dominance relation* that formalized the notion of design rules as a pair-wise $cannot - influence$ relations. Based on the constraint network, we have formally defined what it means by "one decision depends on the other". The dominance relation allows for automatic decomposition of the system into *modules*, each representing an independent task assignment.

A DSM is a square matrix in which columns and rows are labeled with variables, and the matrix is populated with a pair-wise dependence relation that can be automatically derived from an ACN. In order to reveal the modular structure of a design, the user can aggregate all the design rules (commonalities) into the first block of the matrix. After that, the user can assess if all other blocks along the diagonal (modules) are independent from each other (there are no off-block dependencies). After that, the NOV value can be computed according to the number of modules, their sizes and their dependency relations.

We plan to model each feature and base model with a complementary ACN model. Within each feature, we model all the dimensions as variables, and model the relations within and among features as constraints. Similar to how features are picked to synthesize the final constructed software, the individual ACN models of the features can be combined to form the ACN for the final software product, with every possible model combination having a complementary ACN model.

Take the small calculator generator in (Batory et al. 2004) as an example. The base model consists of a $clear()$, an $enter$, and three global variables, and the extended parts include $add()$ and sub functions. We thus use five design variables to model the five dimensions in the base. We also use another two ACNs to model the two extensions, each having one dimension. To assess the modularity properties of the resulting system, we combine these ACNs together. Depending on how the FOMDD translator works, the dependencies among these three ACNs should be automatically detected. For example, all the extended functions will depend on the three global variables, which should be automatically detected and transformed.

The challenge is to combine individual feature ACNs to compose the full ACN, we would need an algorithm similar to algorithms used in FOP (such as AHEAD (Batory et al. 2004)) to combine individual features to produce the final product. However, ACNs being more abstract models, might be easier to combine. Since ACN uses logical expressions as the computational core, when ACN combination is needed, inconsistent constraints should be detected and resolved, if possible. Each feature's interaction with the base model and other features will need to be analyzed to construct a well formed full ACN. Once this ACN is produced by combining selected feature ACNs, we can assess the modularity of this model using NOV metrics (Cai et al. 2007), as well as other metrics such as volatility and concern scope/impact.

References

Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

Yuanfang Cai, Sunny Huynh, and Tao Xie. A framework and tools support for testing modularity of software design. In *22nd IEEE/ACM International Conference on Automated Software Engineering*, November 2007.

Jia Liu and Don Batory. Automatic remodularization and optimized synthesis of product-families. In *3rd International Conference on Generative Programming and Component Engineering*, October 2004.

Christian Prehofer. Feature oriented programming: A fresh look at objects. In *11th European Conference on Object Oriented Programming*, 1997.

Salvador Trujillo, Don Batory, and Oscar Diaz. Feature oriented model driven development: A case study for portlets. In *29th International Conference on Software Engineering*, May 2007.

Measuring Design Volatility against Design Rule Stability

Yuanfang Cai, Kanwarpreet Sethi
Department of Computer Science
Drexel University
Philadelphia, PA, 19104
{yfcai, kss33}@cs.drexel.edu

Abstract

Design stability is usually measured based on direct and indirect dependencies among components within software source code. But designers need to understand how well a design can maintain desired modularity properties under given changes before coding or making changes. It is also possible that a piece of code is highly coupled but not subject to change. The problem is that current design stability assessment does not explicitly take into account environmental parameters and their changeability. In this paper, we propose to develop a new design stability metric based on the concept of design rule and design structure matrix. The idea is to measure design volatility against design rule stability, that is, how design rules can resist environmental changes. Design structure matrices allow us to explicitly represent environmental parameters and quantitatively assess design stability.

Keywords: *Design Stability, Modularity, Metrics*

1. Introduction

Measuring design stability is fundamental in software development. Stability is usually measured based on how software components depend on each other syntactically, such as the number of classes outside a package that depend on classes within this package [7]. Large number of empirical studies has been conducted to assess how modern programming paradigms influence software stability [3][4].

Three issues remain: first, assessing source code stability is not sufficient because designers usually have to estimate change impact before coding or before making changes. Second, it is possible that some part of the system is highly coupled but is not subject to environmental changes. As a result, the design could have a low stability value, but is highly stable. Finally, the stability of interfaces of modules is much more important than the stability of their implementations that are much less visible. Current design stability assessment does not take this distinction into account.

We propose a *volatility* metric based on Design Structure Matrix (DSM) modeling and design rule theory [1] [6]. A DSM is a square matrix in which rows and columns are labeled with design variables and marked cells represent dependencies among these decisions. Design rules are stable design decisions serving as interfaces that decouple otherwise coupled design decisions. Sullivan et al. [6] have shown the Parnas's information hiding criterion [5] can be precisely captured in a DSM: if a design is information hiding modularized, its design rules should be invariant to environmental changes. The new stability metric is derived from this observation.

We assess design stability in terms of both how design rules are subject to environmental changes, and the impact scope of design rules. The rationale is that if a design rule influence a lot of other decisions, then its stability is more important than other design rules that just impact a few decisions. We call the new metric as *volatility*. DSM modeling allows us to model and quantitatively assess environmental parameters, design rules, and their impact scopes. Our prior work [2] shows that DSMs can be automatically generated from logical models with precise semantics. As a result, the volatility value can be automatically computed from logical design models.

In the next section, we present the definition of the *volatility* metric, and use the canonical Key Word in Context (KWIC) example [5] to show how the metric can be used to compare the stability of two designs. Figure 1 and Fig-

ure 2 depict DSM models of the information hiding design and the sequential design that Parnas analyzed informally [2][5]. Rows 1-4 in both DSMs represent environmental parameters (EPs), the gray blocks are the design rules, and the lower right part of the DSMs show subordinating decisions (SDs) that constitute independent modules.

2. A New Design Stability Metric

We propose to measure design volatility in terms of design rule stability, *stability(dr)*, and design rule impact scopes, *scope(dr)*. The DSM-based definitions are as follows:

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
envr_input_forma	1	.																			
envr_core_size	2		.	x																	
envr_input_size	3		x	.																	
envr_alph_policy	4				.																
alph_ADT	5					.															
linestorage_ADT	6						.														
output_ADT	7							.													
circ_ADT	8								.												
master_ADT	9									.											
input_ADT	10										.										
master_impl	11					x	x	x	x	x	x	.									
circ_ds	12		x	x									.	x							
circ_impl	13						x		x				x	.							
output_impl	14					x	x	x							.	x					
output_ds	15														x	.					
linestorage_ds	16		x	x													.	x			
linestorage_impl	17			x			x										x	.			
input_impl	18	x					x				x								.		
alph_impl	19				x	x	x		x											.	x
alph_ds	20				x															x	.

		x	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
envr_core_size	x	.		x															
envr_input_forma	2		.																
envr_input_size	3	x		.															
envr_alph_policy	4				.														
alph_sig	5					.													
output_sig	6						.												
input_sig	7							.											
circ_sig	8								.										
circ_ds	9	x		x						.	x	x							
input_ds	10	x		x						x	.	x							
alph_ds	11			x	x					x	x	.							
master_impl	12					x	x	x	x				.	x					
master_sig	13													.					
output_impl	14			x	x		x			x	x	x			.	x			
output_ds	15															.			
circ_impl	16			x					x	x	x						.		
input_impl	17		x	x				x			x							.	
alph_impl	18			x	x	x				x	x	x							.

Figure 2. KWIC Sequential Design

For any design rule, *dr*, its stability is:

stability (dr) = number of EPs that dr depends on.

Its impact scope is:

scope(dr) = number of SD that depend on dr.

Its volatility is:

*volatility(dr) = stability(dr)*scope(dr)*

The volatility of the whole design, *d,* is thus defined as:

volatility(d) = ∑volatility(dr)

The higher the *volatility*, the less stable the design is.

For example, consider the design rule *circ_ds* in Figure 2. *stability (circ_ds)* is 2, because it depends on two environmental parameters. *scope(circ_ds)* is 5 because five other design variables depend on it. Its volatility is thus 10. Similarly, the volatility of *input_ds* is 12, and the volatility of *alph_ds* is 8. As a result, the volatility of the sequential design is 30. By contrast, in the information hiding DSM shown in Figure 1, none of the DRs depend on any environmental parameters, so its volatility is 0. These numbers show that the information hiding design is much more stable than the sequential design.

3. Conclusions and Discussions

In this paper, we proposed a design stability metric, *volatility*, based on design rule stability relative to environmental conditions, and design rule impact scope. This metric can be quantified based on DSMs. We have used this metric to compare the KWIC information hiding design and sequential design, show that the information hiding design is a lot more stable than the sequential design.

The metric can be further developed in several ways. For example, instead of just counting the number of EPs a design rule depends on, we can also model the probability that the EPs will change, taking uncertainty into account. Design rule impact scope can also be extended to assign weight to each dependent if they require different level of efforts to change.

Using this metric requires the user to identify design rules and identify environment parameters. Some design rules can be implicit and identifying all environmental conditional is difficult. We are exploring the possibility of automatically generating EPs from requirement documents. In addition, a DSM model can always be extended whenever an implicit decision is discovered and recorded.

References

[1] Carliss Y. Baldwin and Kim B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.

[2] Cai, Y. and Sullivan, K., Simon: A tool for logical design space modeling and analysis. ASE 2005.

[3] Figueiredo, E. et al.: Evolving software product lines with aspects: An empirical study on design stability. ICSE 2008.

[4] Greenwook, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. ECOOP 2007.

[5] Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, December 1972.

[6] Sullivan, et al. The structure and value of modularity in software design. In *ESEC/FSE '01*, pages 99–108.

On the Assessment of Pointcut Design in Evolving Aspect-Oriented Software *

Raffi Khatchadourian †
Ohio State University
khatchad@cse.ohio-state.edu

Phil Greenwood Awais Rashid
Lancaster University
{greenwop,awais}@comp.lancs.ac.uk

1. Introduction

The pointcut expression (PCE) is a key mechanism in enabling Aspect-Oriented Programming (AOP) (Kiczales et al. 1997) to improve the localization of crosscutting concerns. PCEs quantify over well-defined points in the execution of the program called *join points*. A *join point shadow*, on the other hand, refers to base-code corresponding to a join point (Xu and Rountev 2008), i.e., a point in the code where the compiler may perform the weaving (Masuhara et al. 2003). Advice *joins* at these points to allow the crosscutting concerns to be composed in an appropriate manner. PCEs need to be well-designed to ensure that they are correct in terms of identifying relevant join points to make certain the crosscutting concerns are composed correctly. Furthermore, PCEs should be robust in the midsts of base-code alterations. That is, changes to the base-code can lead to join points incorrectly falling in our out of scope of the pointcut expressions. Such situations are problematic in that they can cause crosscutting concerns to be composed incorrectly. If undetected, this could cause the composed program to behave unexpectedly, thus causing errors to occur. PCEs that result in such unexpected behavior of the composed program due to evolution are often referred to as "fragile" (Koppen and Stoerzer 2004).

The skill required to design a robust PCE, especially in languages such as AspectJ (Kiczales et al. 2001), is often considered a "dark-art", as well as associated with many common pitfalls (Colyer et al. 2004). Typically, a number of alternative PCEs exist that are equivalent in terms of their end composition effect. For example, if all method executions within a class called `Test` are intended to be advised, multiple strategies may be employed. For instance, a generic PCE could be used that quantifies over all method executions (e.g., `execution(* Test.*(..))`), or each method could be enumerated individually (e.g., `execution(* Test.methodA (..))|| execution,...`).

* This material is based upon work supported by European Commission grants IST-33710 (AMPLE) and IST-2-004349 (AOSD-Europe).

† This work was administered during this author's visit to the Computing Department, Lancaster University, United Kingdom.

```
1 public class FooBar {
2   private int foo, bar;
3   public void setFoo(int f){this.foo = f;}
4   public void setBar(int b){this.bar = b;}
5 }
```

Figure 1. Example base-code.

Deciding which strategy is best in order to balance robustness, correctness, and precision is a non-trivial task. Apart from simple aforementioned PCEs, it is often impossible to ascertain prior to making maintenance changes whether the PCE will be, in fact, robust. Normally, it is only when maintenance changes have been made that fragile pointcuts are uncovered, which is an undesirable scenario. This paper outlines our intent to provide quantitative indicators in estimating the ability of a given PCE to preserve its semantics despite base-code alterations that may take place in the future. These indicators may then serve as a basis for suggesting alternative, more suitable PCEs.

2. Motivation

Consider the base-code snippet depicted in Figure 1 which defines a simple class `FooBar`. The class declares two integer fields, `foo` and `bar`, which are modified by two instance methods `setFoo(int)` and `setBar(int)`, respectively.

Suppose the developer wishes to advise the executions of methods that modify the state of `FooBar`. The most obvious PCE to capture such join points would take advantage of the `set` naming convention, possibly taking the form `execution(* FooBar.set*(..))`. Further suppose that the class is modified to introduce a method `incFoo()`, whose sole functionality is to increment the current value of `foo` by 1. Due to its construction, the current PCE would not capture

this new method; thus, the PCE, in this case, fails to capture the *true* intentions of the developer.

Through analyzing the currently advised shadows, we can extract a set of patterns that describe the underlying intentions of the developer. In this example, a common pattern exists that revolves around both advised shadows setting some field in `FooBar`. This pattern then can be subsequently applied to the modified version of `FooBar` and will suggest the newly introduced `incFoo()` method to be included in the PCE due to it also setting some field in `FooBar`. It is inevitable that patterns will be extracted which do not represent the developer's intentions and so cause incorrect suggestions to be made. To indicate the level of confidence in a pattern/suggestion quantitative indicators should be attached to each pattern, and subsequently each suggestion, to indicate how useful the suggestions may be. Such indicators can then be used to infer the how closely the original PCE captures the developer's intentions.

3. Pattern Metrics

The quality of the patterns can be measured in terms of the number of current shadows which they are representative of which can be used to infer their potential ability to capture new shadows in future versions of the software. However, it is equally important to measure the number of execution points which the pattern is also representative of but are not a shadow according to the PCE. This accuracy can be measured in terms of four indicators:

- True Positives (TP) - the number of actual shadows which the pattern matches.
- False Positives (FP) - execution points that match the pattern but are not a shadow.
- False Negatives (FN) - the number of actual shadows that are not matched by a particular pattern.
- True Negatives (TN) - counts how many potential shadows the pattern could have suggested but correctly did not.

From these four indicators a confidence metric can be calculated which is a ratio between recall and fall-out metrics:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{Fall-Out} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

$$\text{Confidence} = 1 - \frac{\text{Fall-Out}}{\text{Recall}}$$

A confidence value can be calculated for each PCE specified which is the average of all patterns derived from each PCE. This can be used to indicate how representative a particular PCE is of the underlying intentions of the developer and subsequently how accurate it will be as changes are made to the base-code in terms of preventing shadows incorrectly falling in or out of scope.

Although the final confidence value for each PCE is useful in itself, the confidence values of each individual pattern that has been derived from the analysed PCE can be used to improve the design of the pointcut. For example, if an intention pattern is found with a high confidence (i.e. tending towards 100%) then the developer should look to express the pointcut in terms of that pattern. This is exemplified in the `FooBar` class whereby a set pattern is discovered which closer to the true intentions of the developer and is also able to ensure the newly introduced shadow is correctly advised.

4. Conclusion and Future Work

We have discussed initial insight into quantitatively assessing the quality of pointcut expressions in terms of their ability to accurately capture the underlying intentions of the developer. We envision a tool that would be able to predict the robustness of a given pointcut expression, thus reducing the need for pointcut maintenance. Future work consists of a rigorous treatment of the evaluation metrics, as well as an empirical evaluation of a tool possibly extending current approaches (Dagenais et al. 2007; Khatchadourian and Rashid 2008).

References

Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley Professional, 2004.

Barthélémy Dagenais, Silvia Breu, Frédéric Weigand Warr, and Martin P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *Int. Conf. Automated Software Engineering*, 2007.

Raffi Khatchadourian and Awais Rashid. Rejuvenate pointcut: A tool for pointcut expression recovery in evolving AO software. In *Int. Work. Conf. on Source Code Ana. and Manip.*, 2008.

Gregor Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Eur. Conf. Object-Oriented Programming*, 1997.

Gregor Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Eur. Conf. Object-Oriented Programming*, 2001.

C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem. In *Eur. Int. Workshop on Aspects in Software*, 2004.

H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *International Conference on Compiler Construction*, 2003.

Guoqing Xu and Atanas Rountev. Ajana: a general framework for source-code-level interprocedural dataflow analysis of aspectj software. In *Int. Conf. Aspect-Oriented Software Development*, 2008.

Assessing the Malleability of Modular Design

Giuseppe Valetto
Drexel University
Department of Computer Science
valetto@cs.drexel.edu

Abstract

One of the main intents of modularization is the creation of a malleable design, one that can easily accommodate change over time. The malleability of a modular structure can be defined operationally in a variety of ways, depending on what development practices or modularization techniques are adopted. We exemplify that by discussing the different meanings of malleable design in agile vs. traditional software development practices, and how it can be differently measured in those two cases. We propose to devise appropriate operational definitions of malleability for the various contemporary modularization techniques, and use them to assess the quality of modular design.

Categories and Subject Descriptors D.2.2 [**Software Engineering**]: Design Tools and Techniques - *Modules and interfaces.* D.2.8 [**Software Engineering**]: Metrics – *Product Metrics.*

General Terms Measurement, Design.

Keywords modularization assessment, malleable design, quality of design.

1. Introduction

Modularity plays a paramount role in design, as it allows to break down and conquer system complexity, and to assign clear-cut development responsibilities within the development team [6]. Another major benefit of a well-modularized design is its resilience to the unavoidable changes that occur throughout a software project. As remarked in [8], modules and their interfaces provide designers with a set of options for extending and enhancing the software product. Modularity makes design **malleable**, that is, able to sustain change over time, with little or no disruption to its established intent and structure. Our emphasis on the temporal factor with respect to malleability is intentional. Contrast that with, for example, measures of *design (in)stability* [10], which express the likelihood of a given design to be impacted, whenever some change is required to the system. Malleability, instead, is meant to measure how well software can withstand the "injury of time" (that is, the accumulation of change occurring across its life span), while avoiding the progressive obfuscation and decay of its design

2. Position statement

Malleability can be defined in abstract terms as a comparison over a period of time between the amount of change to the modular structure of the system and the amount of change occurring in its implementation. As such, it offers a quantitative perspective on the effectiveness of modularization. This concept of malleability is general, and independent of the mechanisms, practices and technologies used for design and implementation within a project. In operational terms, however, the conceptual definition of malleability must translate into an array of different measures. In particular, whereas implementation changes can be arguably measured generally, for instance via metrics such as code churn [4], changes in modularity are instead quantified differently, depending on the specific modularization technique adopted in a project.

Similarly, malleability may be measured differently, also depending on the adopted development process. As an example, we discuss in the remainder the meanings of malleable design in agile vs. traditional software development, and how different measures are applicable in those two cases. In the discussion that follows, we implicitly assume a "classic" approach to modularization, based upon information hiding. However, analogous considerations could be made for different modularization techniques.

3. Planned modularization

For a long time, software design was seen primarily as a planning activity, as design artifacts would represent the

blueprints and the master plan for the software. In that master plan, modularity plays a prominent role, since it enables the assignment along module boundaries of system functionality and development responsibilities. Accordingly, what we could call *planned modularization* ensues, a top-down process of refinement, starting with architectural specifications, and then further breaking down the product into finer-grained elements.

In such a context, a key concern is *conformance* between the planned modularization and its implementation in code. Developers must strive to implement their tasks within the planned modular framework; in turn, each design iteration must strive to reconcile the modular structure with any implementation variation that has remained unaccounted for. Whenever, in the face of substantial implementation work, the system succeeds in remaining conformant to the planned design, we have evidence of a design that's malleable. Huynh et al. [3] propose a technique to automatically enumerate elements of non-conformance within Design Structure Matrices (DSMs) [1] that capture the planned modular structure vs. how it is actually implemented. Following that approach, malleability of a planned modular structure can be measured as **the observed level of conformance, normalized against the amount of code churn in the implementation**.

4. Emergent modularization

With the advent of agile methods, the focus of design activities has shifted. By embracing continuous change and refactoring, agile projects consider design not as a master plan, but as a short-lived, fluid configuration of the software product, which can be modified, as per the stakeholders' needs, at any time across many micro-iterations. In agile methods design is not "dead" [2]; rather, it emerges incrementally from a participatory, evolutionary and possibly implicit process carried forward by all developers from the bottom up. Planned modularization is thus replaced by *emergent modularization*.

In such a context, a key concern is *design convergence*. Continuous refactoring may lead to periods of volatility in the modular structure of the software. Ideally, however, as the project progresses volatility should decrease, and converge towards a high-quality, largely stable design (possibly discounting recurrent oscillations due to design churn [9]). As an emergent modularization becomes incrementally less volatile across agile iterations, it shows increasing levels of malleability in the face of ongoing changes. In this case, malleability is best measured in terms of derivatives, that is, **as the inverse of the rate of design change over time, normalized by the rate of implemented changes, i.e., relative code churn over time, as defined in [5]**.

5. Conclusions

This paper proposes to assess the quality of modular design in terms of its malleability. Malleability is defined as the resilience of a design to the accumulation of software changes that occur over time in a project.

We maintain that malleability is a generic concept that can be used across the spectrum of modularization techniques, as well as the diverse design approaches embraced by different process models. However, we have shown by example how operational ways to measure malleability may differ in all of those contexts; therefore, finding appropriate measures should become a subject of research.

Further research questions regard the relationships between malleability and other known measures of design (e.g. coupling and cohesion [7], or stability [10]). Malleability could also have interesting consequences onto other qualities of the software product and the development process. In the next future, we plan to investigate the relationship of malleability with fault proneness of a software product, as well as with the effort and cost in a project.

References

[1] Baldwin, C.Y., and Clark, K.B. "Design Rules, Vol. 1: The Power of Modularity". The MIT Press (2000).

[2] Fowler, M. "Is Design Dead?", keynote at the XP Conference 2000, http://martinfowler.com/articles/designDead.html

[3] Huynh, S., Cai, Y., Song, Y., and Sullivan, K. "Automatic Modularity Conformance Checking". In Proc. of ICSE 2008 (2008).

[4] Munson, J.C., and Elbaum, S. "Code Churn: a Measure for Estimating the Impact of Code Change, in Proceedings of the 14th IEEE International Conference on Software Maintenance (ICSM'98), (1998).

[5] Nagappan, N., and Ball, T. "Use of Relative Code Churn Measures to Predict System Defect Density". In Proc. of ICSE 2005 (2005).

[6] Parnas, D.L. "On the criteria to be used in decomposing systems into modules". Comm. of the ACM, 15(12), (1972)

[7] Stevens, W., Myers, G., Constantine, L. "Structured Design" IBM Systems Journal, 13(2): 115-139(1974).

[8] Sullivan, K, Cai, Y., Hallen, B, and Griswold, G.W. "The Structure and Value of Modularity in Software Design". In Proceedings of FSE'01 (2001).

[9] Yassine, A. et al. Information Hiding in Product Development: The Design Churn Effect. Research in Engineering Design, 14:145-161 (2003)

[10] Yau, S.S., and Collofello, J.S. "Design Stability Measures for Software Maintenance" IEEE Transactions on Software Engineering, 11(9):849-856, September 1985

Using Metadata in Aspect-Oriented Frameworks

Eduardo M. Guerra
Instituto Tecnológico da Aeronáutica (ITA)
Computer Science Division
guerraem@gmail.com

Jefferson O. Silva
Instituto Tecnológico da Aeronáutica (ITA)
Computer Science Division
jefferson.o.silva@uol.com.br

Fábio F. Silveira
Federal University of São Paulo (UNIFESP)
Technology and Science Department
fsilveira@unifesp.com.br

Clóvis T. Fernandes
Instituto Tecnológico da Aeronáutica (ITA)
Computer Science Division
clovistf@uol.com.br

Abstract

A traditional approach for building aspect-oriented frameworks has difficulty in modularizing crosscutting concerns that have different behaviors in diverse situations. Each crosscutting behavior needs to be implemented by an advice, which can lead to a combinatorial explosion in the number of advice. Intercepted classes can use metadata to adapt its behavior to these diverse situations. This paper presents an approach that merges aspect orientation and metadata creating a more flexible and extensible solution that results in a smaller number of advice.

Keywords aspect; framework; metadata; @OP

1. Introduction

An important characteristic of software frameworks is their ability to promote reuse. In fact, they can be defined as collections of classes that make up a reusable design for a specific class of software [7]. Basically, their reuse consists of capturing a common design for an application specific domain. Other important features are extensibility, modularity, and inversion of control (IoC). IoC means that the control flow shifts from the application to the framework.

Aspect-oriented programming (AOP), due to its crosscutting capability, opens a new research line with respect to reuse, extensibility and modularity in the development of applications. Aspect-oriented framework (AOF) aims at the same purpose of an object-oriented framework, however, an AOF counts on the AOP composition mechanisms, which include aspects, beyond classes [3] [4]. AOP composition mechanisms augment modularization of applications, since they allow encapsulation of crosscutting concerns [3]. There are papers that analyze modularization of functional concerns [11] [5] and non-functional ones [1] [12].

This work discusses the advice combinatory explosion that may occur with the utilization of aspect-oriented techniques. It also proposes metadata usage in intercepted classes to allow aspects to have behavior variations in a single crosscutting concern.

This paper proposes a structure that allows the expansion in the metadata schema, providing more flexibility in frameworks. In order to validate the presented ideas, it presents a case study that uses the technique addressed in this research.

This paper is outlined as follows. Section 2 details the complexities involved in the building of frameworks using aspect-oriented approaches. Section 3 discusses the concepts of metadata in an AOF and how they can help to reduce complexity. Section 4 presents the Metadata-based Logger, which is an AOF developed to exemplify the concepts in this paper. Section 5 analyzes a solution proposed by Camargo and Masiero [3]. Section 6 summarizes this research, outlining the main contributions and possible extensions.

2. Aspect-Oriented Frameworks

Consider these requirements for building a framework that logs method calls: calls to client methods must be divided by the logger in method name, declared exceptions, thrown exception, arguments, declared parameters, declared method return type and the actual method return. Each method call may be logged with any combination of the method parts previously mentioned. For instance, the framework could log only the method name for one method call, while it could log the whole method signature for another one. In addition to that, the framework must allow logging in two different locations: a database and a file. Analogously, it may log in one of the two locations or in both of them.

The conventional aspect-oriented solution for the above requirements typically implements the method parts and the log locations as regular object-oriented classes. Since logging is a crosscutting concern, it is usually modularized by aspects.

The framework aspects need to include logging references inside advice. This entails a high syntactic coupling relationship [14] among advice and the framework classes that implement the logging.

Figure 1 depicts a UML class model representing the relationship among advice, the logging implementation classes and the client code.

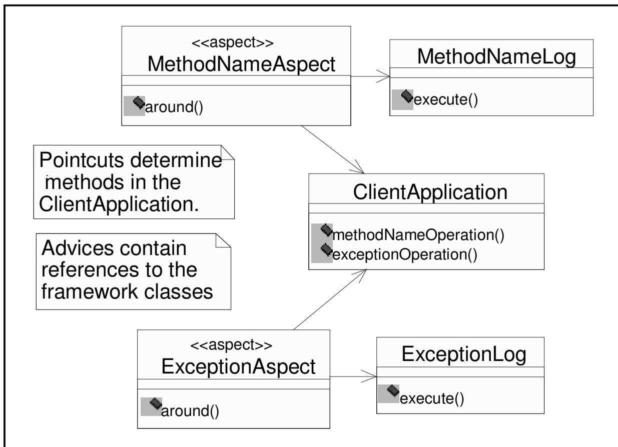

Figure 1. Aspects, framework classes and the application.

The logger framework contains only one crosscutting concern, which is to capture method call information from client methods and send them to the logging implementation classes. However, two advice are needed for capturing the information of the two different client methods.

The main reason for having two advice is that both of them have a reference to a framework class. Thus, in Figure 1, for collecting a method name it is necessary to use the aspect *MethodNameAspect*. Similarly, for collecting the method declared exceptions, it is necessary to use the aspect *ExceptionAspect*. It means that, using the AOP approach, a crosscutting concern implemented by an advice cannot be reused for different framework logging classes.

A characteristic that can vary in one crosscutting concern is called variability. In the conventional aspect-oriented technique, the number of variabilities is directly proportional to the number of advice.

Figure 2 displays the number of framework variabilities and the number of advice. This relationship can be represented by the function $f(x) = x$.

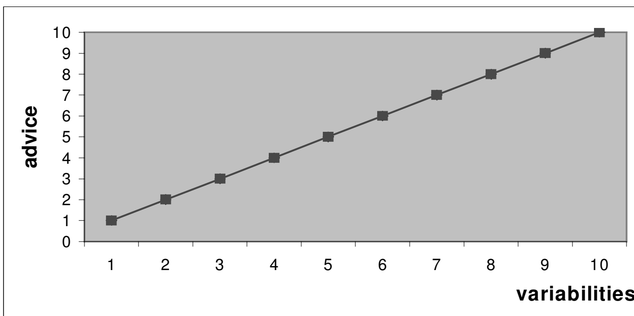

Figure 2. Relationship among advice and variabilities

The function $f(x)$, portrayed in Figure 2, does not consider any variability combinations. Some business rules cases specify that variabilities can be combined. For instance, in the framework cited previously, combining the database and file variabilities would cause the framework to log in both locations, implying in the definition of a new behavior. That is, a new advice would have to be created, implementing the crosscutting behavior that allowed the logging to be performed in both locations.

When it is considered the possibility of variability combinations, it is necessary to redefine the function that relates the number of advice and variabilities. The new function can be determined by the fundamental principle of counting, which regards combinations. Thus, it is defined as follows: $g(x) = 2^x - n$, where x is the total framework variabilities and n is the number of variabilities that must be present. For instance, in the logger framework at least one method call part must be captured for logging and at least one location must be chosen for the log information to be registered. Therefore, for the logger framework case $n=2$.

Figure 3 illustrates the advice growth scale.

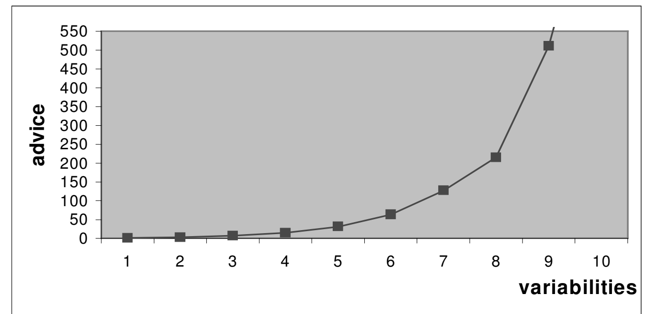

Figure 3. Advice and variabilities with combinations

When variability combinations are considered, the number of advice that needs to be created grows exponentially. The number is significant even for a little variability.

The extensibility is affected as well. Suppose that an application needs the framework to support another variability. In the best scenario, which is the case of variabilities that cannot be combined (Figure 2), a new advice must be created. In the worst one, which is the case of variability combinations (Figure 3), the number of advice that must be created is defined as follows: $g'(x) = g(x) + n$, where x is the total framework variabilities and n the number of variabilities that must be present. The function $g'(x)$ can be factored into: $g'(x) = 2^x$.

3. Aspect-Oriented Frameworks Based on Metadata

Niso [9] defines metadata as structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource. This paper defines frameworks that utilize metadata as a basis for decision-making in runtime as metadata-based frameworks (MBF).

In the Java language, with the introduction of JSR 175 [8], it became possible the usage of attribute-oriented programming (@OP). Attribute-oriented programming is a program-level marking technique that allows the programmers to mark program elements (e.g. classes and methods) to indicate that they maintain application-specific or domain-specific semantics [13].

Some frameworks, as EJB 3.0 [6] and Hibernate [2] for example, make extensive use of @OP. EJB 3.0, for example, uses metadata, defined in annotations or XML files, for the definition of crosscutting concerns variabilities like transaction management and access control. As EJB 3.0 is a specification, it does not define how the behavior must be implemented.

The difference between @OP and MBFs is purely conceptual. While @OP focuses on coding (with annotations), independently of the manner metadata will be processed, MBFs focus on metadata processing in runtime, irrespectively of the way they are stored.

A convenient fashion to solve the syntactic coupling drawback [14], detailed in section 2, is to use metadata for the definition of crosscutting concern variabilities in an AOF. The technique consists of a two-step activity: (i) remove all references to variabilities from advice; and (ii) include the necessary information for the composition activity in metadata.

Metadata usage prevents the combinatory explosion that may result as a consequence of the previously mentioned coupling, creating a more effective modularization. New behavior defined by variability combinations can be configured in metadata, dispensing with the need for the creation of new advice. Therefore, a single crosscutting concern can support many variabilities.

Neto et al [14] present a definition for syntactic and semantic coupling. Moreover, Yang and Tempero [15] have shown criteria for dealing with indirect coupling. The use of metadata helps to mitigate the syntactic coupling, however, it does not eliminate all kinds of semantic or indirect couplings.

This approach also allows the insertion of new functionality in the framework by extending the metadata schema without having to create additional advice.

4. Metadata-based Logger Framework

Metadata-based Logger [10] is an aspect-oriented framework based on metadata, which we developed to comply with the requirements described for the framework in section 2. It uses metadata to indicate what method call information must be logged and the locations where the log must be recorded. The framework collects attributes through metadata, which can be stored in annotations or in an XML file. For instance, a client method annotation could specify that an aspect must log the method name in the database and file. Another annotation could determine that the name and the declared exceptions of a method must be logged only in the database.

Implementing Metadata-based Logger framework using the usual aspect-oriented techniques would lead to an explosion in the number of advice, since there would be numerous combinations.

Metadata should be placed prior to client methods declarations when annotations are used. In the case of an XML file, it must conform to the framework XML syntax. Metadata configure which method call parts will be collected and where the logging will take place.

Figure 4 shows a client code snippet using annotations to specify to the framework the logging information for a method.

```
@LogMarker(
    logInfo = { LogKeys.METHOD,
                LogKeys.PARAMETER,
                LogKeys.ARGUMENT,
                LogKeys.DECLARED_RETURN_TYPE,
                LogKeys.RETURN_TYPE,
                LogKeys.EXCEPTION,
                LogKeys.THROWN_EXCEPTION },
    logLocation = {
                LogKeys.FILE,
                LogKeys.DATABASE }
)
public void testLog(Integer i) throws NullPointerException, IOException {
    System.out.println("Method executed");
}
```

Figure 4. Client method configured with an annotation

Metadata-based Logger utilizes a flexible architecture. The *LogKeys* interface contains the prefixes of the classes that implement the variabilities. Each variability has a correspondent suffix. The keys put in the *logInfo* attribute are suffixed with *LogInfo* to form the name of the variability class, while keys put in the *logLocation* attribute are suffixed with *Location*. For instance, a class that implements the logging in a file is named with the combination of the prefix defined in *LogKeys* and the correspondent suffix, resulting in the name *FileLocation*.

Analogously, Figure 5 portrays the same previous configured method following the XML syntax. The variability name formation rules are the same detailed for annotations.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<logmarker>
   <method signature="void logging.LoggerApplication.testLog(Integer)">
       <info>
           <value>Method</value>
           <value>Parameter</value>
           <value>Argument</value>
           <value>DeclaredReturnType</value>
           <value>ReturnType</value>
           <value>Exception</value>
           <value>ThrownException</value>
       </info>
       <type>
           <value>DB</value>
           <value>File</value>
       </type>
   </method>
</logmarker>
```

Figure 5. Method configured with the XML syntax

The framework advice are responsible for capturing the common crosscutting behavior, which in the case of the logger framework consists of mapping method calls information and delegate the rest of the process to a factory that is able to interpret the specified metadata. For increasing the expressiveness of the pointcuts, the aspect that encapsulates the crosscutting behavior was made abstract (`AbstractAspectLogger`).

The framework factories contain logic for metadata consumption. They interpret the defined metadata for each method and recover the variability prefix. In this manner, the factories can instantiate the correct class. It provides more extensibility because if a new variability is created, all that is required is to create the class that implements the functionality and define its prefix in *LogKeys*. The Logger implements a structure that uncouples the metadata consumption logic from the component logic and also allows many kinds of metadata storage. Figure 6 depicts a UML class diagram that abstracts the architecture of the framework.

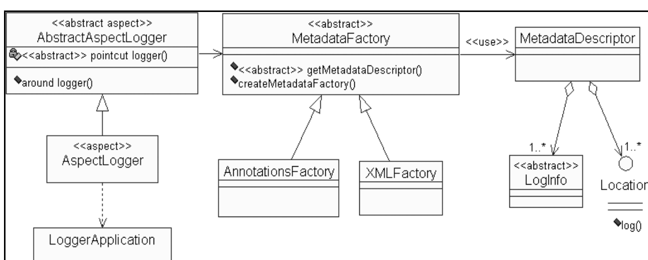

Figure 6. Metadata-based Logger UML class diagram

For comparison sake, if the traditional aspect-oriented approach have been used to implement Metadata-based Logger framework, we would needed a plethora of advice to represent all the behavior combination possibilities. The Metadata-based Logger framework has a total of ten variabilities. The function *g(x)*, demonstrated in section 2, calculates the number of advice that need to be created for a specified number of variabilities. For every combination made, there must be at least two variabilities: one for collecting data for the logging; and another one for choosing a location for the logging to take place. It can be verified that it would be necessary 510 advice to cover all possible combinations.

5. Table-based Calculation Framework Revisited

This section introduces the Table-based Calculation, a framework that utilizes aspect-oriented techniques for treatment of business rules [3]. The aim is to suggest a metadata-based solution for the framework.

Camargo and Masiero [3] present a crosscutting framework that modularizes a business rule that has the objective to increment or decrement intercepted values from the base code. A percentage calculation is performed on the captured value according to its range. The authors provide two business rule examples: the Meal Ticket, as a rule without reduction; and Income Tax, as a rule with reduction.

There are variations on the manner these values can be obtained. Table-based Calculation provides composition alternatives, which are implemented in abstract aspects that contain the logic for the extraction of values from the base code.

The necessity for composition alternatives are justified because a value to be captured may be in many places such as: in the method return; in the arguments of a method; in a local variable of the calling object (*this*); or in a local variable of the target object (*target*).

The authors present the concretization of an aspect that implements the Meal Ticket business rule (Figure 7).

```
public pointcut obtainValue():
   call (* BaseSalary.getValue())
   && withincode (* Employee.calculateMySalaryBasedOnEvents(int, int));
public String getNameOfTableClassWithPackage(){
      return "tableBasedCalculation.instantiation.MealTicket";
}
public String getValueType(){ return "float";}
```

Figure 7. Meal Ticket implementation aspect [3]

The pointcut `obtainValue()` determines the method where the aspect is going to act. The method `getNameOfTableClassWithPackage()` indicates the class of the framework that implements the business rule. Similarly, the method `getValueType()` comprises information about the method return type. As discussed previously, the syntactic coupling among advice and framework classes that implement business rules imply in the creation of many advice.

In Figure 7, the high coupling occurs because of the methods `getValueType()` and `getNameOfTableClassWithPackage()`. If another business rule (e.g. Income Tax) needs to be created there will have to be another aspect which the only difference will be the referenced class.

References to the framework classes do not need to be contained inside the aspect. Base code methods could have been annotated or described in XML to contain these references. Figure 8 illustrates how the client method could be annotated. Although it is used an annotation as metadata, any other type could have been used.

```
@Calculation(
    tableClass = {"MealTicket"},
    valueType = {"float"}
)
public float getValue() {
    return value;
}
```

Figure 8. Annotated method

Metadata-based solutions face a limitation on the base code value extraction. It is possible that the value needed from the base code is in a method local variable. There is a limitation for Java (until version 6) that does not provide a reflection mechanism for obtaining local variables, making it impossible to capture these types of values in runtime. For the specific case of annotations, it is possible to annotate variables, however they will only serve as documentation. For this case, the traditional aspect-oriented approach would have to be used. That is, it would be necessary to use composition alternatives including references to variabilities in the framework aspects.

6. Conclusion

This paper has proposed the utilization of metadata in Aspect-Oriented Frameworks (AOF). Conventional aspect-oriented solutions accomplish all the composition activity inside aspects. The primary consequence of this approach is that advice need to reference the framework classes that implement the business rules. A characteristic that can vary in the same crosscutting concern is called a variability. This syntactic coupling [14] results in the growth of the number of advice directly proportional to the number of variabilities.

Some business rules require the combination of framework variabilities. Variability combinations determine new behavior. When combinations are considered, the scale of the number of advice grows exponentially.

Metadata usage in AOFs breaks the existing syntactic coupling [14] among advice and variabilities, resulting in more effective modularization. References to variability implementation classes can be included in XML files, in annotations or in any other type of metadata.

In this way, metadata can augment reuse and extensibility in AOFs. A crosscutting concern modularized by an advice can be reused for many variabilities. Moreover, it has been verified that the inclusion of a new variability requires less effort when compared to customary aspect-oriented solutions.

The technique presented in this paper, which merges aspect orientation and metadata, is especially interesting for cases when variabilities need to be combined. However, this paper does not make an exhaustive analysis of the possible situations in which the uses of metadata are applicable. An extension to this work would be to analyze different use cases, as the analyses of other use situations, as well criteria for the choice between the traditional aspect-oriented approach and the metadata-based approach.

It is worth mentioning that it is a work in progress and the next phase of the research is to apply the concepts here introduced in a real application as a means to provide an estimate on the number of advice that needs to be created.

7. Bibliographic References

[1] Rashid, A. and Chitchyan, R. (2003) Persistence as an Aspect. In: Proc. of the 2nd International Conference on Aspect Oriented Software Development (AOSD) Boston–USA, March.

[2] Bauer, C. and King, G. (2005) Hibernate in Action. Manning Publishing Co.

[3] Camargo, V.V. and Masiero, P.C. (2005) Frameworks Orientados a Aspectos. In: Anais do 19º Simpósio Brasileiro de Engenharia de Software (SBES'2005), Uberlândia-MG, Brasil, October.

[4] Camargo, V.V., Ramos, R.A. and Masiero, P.C. (2004) Implementação de Variabilidades em Frameworks Orientados a Aspectos desenvolvidos em AspectJ. In: Relatório do 1º Workshop de Desenvolvimento de Software Orientado a Aspectos (WASP'04) – realizado em conjunto com o SBES'2004, Brasília, DF, Brazil, October.

[5] Cibrán, M., D'Hondt, M. e Jonckers, V. (2003) Aspect-Oriented Programming for Connecting Business Rules. In: Proc. of the 6th International Conference on Business Information Systems (BIS'03). Colorado Springs, USA, June.

[6] DeMichiel, L. and Keith, M. (2005) Enterprise JavaBeans (EJB) Specification, 3rd ed.

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

[8] JSR 175. JSR 175 (2003) A Metadata Facility for the Java Programming Language. Available at: http://www.jcp.org/en/jsr/detail?id=175.

[9] Niso (2004) Understanding Metadata. Bethesda MD. Niso Press. Available at: http://www.niso.org/publications/press/UnderstandingMetadata.pdf.

[10] Metadata-based Logger (2008) Metadata-based Logger Framework Available at: https://sourceforge.net/projects/metadatalogger/, July.

[11] Suvée, D., Fraine, D.B. and Vanderperren, W. (2005) "FuseJ: An Architectural description language for unifying aspects and components". In: Proc. of the 1st Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT'05), Chicago.

[12] Vanhaute, B., Win, B. and Decker, B. (2001) "Building Frameworks in AspectJ". In: Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP), Separation of Concerns Workshop. pp. 1-6, June.

[13] D. Schwarz (2004) "Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5," In ON Java.com, O'Reilly Media, Inc., June.

[14] Neto, A. C., Ribeiro, M. M., Dósea, M., Bonifácio, R., Borba, P. (2007) Semantic Dependencies and Modularity of Aspect-Oriented Software. In: 1st Workshop on Assessment of Contemporary Modularization Techniques, May

[15] Yang, H. Y., Tempero, E. (2007) Indirect Coupling as a Criteria for Modularity. In: 1st Workshop on Assessment of Contemporary Modularization Techniques, May

Mining Software Repositories for Evaluating Software Engineering Properties of Language Designs

Hridesh Rajan
Dept. of Computer Science,
Iowa State University
hridesh@cs.iastate.edu

ABSTRACT

Improved separation of concern is important for dealing with increasing complexity of today's software systems. A number of language designs have been proposed in the last decade with the common goal to improve the separation of concerns by providing better modularization mechanisms e.g. mixins, units, roles, layers, hyperspaces, events, aspects, etc. To understand the benefits of a new modularization mechanism, it is important to apply it to real world large scale software systems, where there are real needs for separation of concerns. However, large scale software projects are generally managed very cautiously and adoption of a new technique in these projects is generally harder to achieve. Typically such adoption is driven by demonstrated success of the technique in other large scale projects, a catch-22 situation. In this position paper, I discuss a software repository mining-based technique to achieve the effect of adoption in a large scale software project in a controlled setting. Rich change history available in the version control systems for open source software projects, and advances in software repository mining enable this technique for empirical evaluation of a modularization mechanism.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques — Modules and interfaces; D.2.8 [**Software Engineering**]: Metrics — Complexity Measures; D.3.3 [**Programming Languages**]: Language Constructs and Features — Control structures; Procedures, functions, and subroutines

General Terms

Design, Measurement, Human Factors, Languages

Keywords

modularity, empirical evaluation, software repositories, design for change, information hiding, programming language design, separation of concerns metrics

1. INTRODUCTION

Today, finding an appropriate *separation of concerns* [16] is understood to be perhaps the fundamental challenge in software design. It promotes studying different interests or concerns of a complex problem separately, with none or very little knowledge of other concerns. *Separation of concerns is the key to maintaining overall intellectual control in the face of complex problem and design solutions.* Over the years, the quest for better separation of concerns has led to different technologies, among which are well-established modularization techniques [43, 44] such as structured programming [14, 15, 33, 59], abstract data types [37], and object-orientation (OO) [11, 26, 34, 39, 58].

A whole new set of such modularization techniques as mixins [4], units [22], implicit-invocation [25, 54, 55], hyperslices [28, 42], composition filters [1], adaptive methods [36], roles [35], aspects [31, 48, 49, 45], open classes [10], etc, have emerged in the last decade or so with a common goal to enable improved separation of concerns. Although the Jury is still out on some of these techniques, the overwhelming academic and industrial interest makes it abundantly clear that there is a pressing need for improved separation of concerns techniques. This is primarily because implementation of some concerns are often hard to factor out into separate modules using classical decomposition techniques [56]. For example, the code for a thread policy is spread across the system. These emerging modularization techniques provide engineers with new possibilities for keeping such concerns separate in the source code.

Typical evaluation of a modularization mechanism examines it in the context of canonical examples. For example, figure editor example inspired from JHotDraw [23], a system originally developed as a design exercise by Erich Gamma and Thomas Eggenschwiler, is prevalent in the aspect-oriented programming literature. Evaluation using canonical examples has many advantages. Canonical examples act as a smoke test of the technique, "if the technique does not work in the context of canonical examples, it is unlikely to work in real projects." They also allows readers and researchers to focus on the problem at hand without getting distracted by the essential complexities of the problem domain [8].

Full promise of these techniques is, however, hard to evaluate with just canonical examples. If a technique doesn't work in the context of canonical examples, often we can say for sure that it is not going to work for real world projects, however, if it does work results obtained may not be directly applicable to real world projects. Scalability related issues are almost never examined in a study that uses canonical examples. A new modularization technique can be applied to a real world project to address some of these threats to the validity of its evaluation. A real world study is, however, extremely time consuming. Often it also requires insider

access to subject projects to make the case for adoption. Last but not the least, the case for adoption often depends on the demonstrated success of the modularization technique in the context of other projects, a chicken and an egg situation.

This position paper describes an empirical evaluation technique that we believe could be useful to overcome some of these hurdles. The key idea is to capitalize on the rich version control histories available for the open source software projects to realistically simulate a series of real world changes in software projects. First, an initial and a final version for the open source project are selected. Second, using existing techniques for software repository mining changes between initial and final versions are organized as a sequence of refactorings [13] such that by replaying the refactorings in the specified order on the initial version yields the final version. Third, this sequence of refactorings is transformed into all candidate modularization techniques such that by replaying the modified refactorings in the specified order on the initial version yields the final version, (possibly) improved using a candidate modularization technique. Finally, analyzing each refactored version produced in this manner is compared w.r.t. desired software engineering metrics to compare and contrast modularization techniques. The scope of our evaluation technique is limited to the following:

- *techniques*: language constructs for improved separation of concerns,
- *claims*: in terms of information-hiding modularity [43], design for change, etc, and
- *project settings*: steady-state projects with source control histories.

The main objective of our technique are:

- *overcome the threats to the validity*: to reduce the threat to the validity of empirical evaluations of language designs, our technique should reduce/eliminate biases from the experimental settings, and
- *automation/semi-automation*: to reduce the cost of empirical evaluation and to encourage rigourous, extensive evaluation of modularization techniques, it should be possible to automate much of this technique at a low cost.

In the rest of this position paper, we explain our technique in detail. To make the ideas concrete, we will discuss them in the context of the Ptolemy language recently proposed by Rajan and Leavens [47]. Ptolemy combines ideas from implicit invocation and aspect-oriented languages and has several advantages compared to both. Next section briefly presents this language design. Section 3 presents the proposed evaluation technique and Section 4 concludes.

2. PTOLEMY: A BRIEF INTRODUCTION

Ptolemy is an extension of object-oriented languages with support for quantified, event types [47]. Ptolemy's design is inspired by II languages such as Rapide [38] and AO languages such as AspectJ [30]. It also incorporates some ideas from Eos [50, 48] and Caesar [40]. The key ideas in the language are that it lets programmers declare named *event types* that contain information about the names and types of event arguments (exposed context). An event type declaratively identifies an expression as an event. This event type can then be used to quantify over all such events. Event types reduce the coupling between the observers of the events and the set of events by eliminating the name dependence between the two.

An example Ptolemy program is shown in Figure 1. This code is part of a larger editor that works on drawings comprising points, lines, and other such figure elements [30, 32]. The program is adapted to be more Java-like, whereas the language presented in our previous work on Ptolemy was an expression language [47].

```
1 FElement evtype FEChange{ FElement changedFE; }
2 FElement evtype MoveUpEvent {
3  FElement targetFE; int y; int delta;
4 }
5 interface FElement{}
6 public class Point implements FElement{
7  int x, y;
8  public void setX(int newX){
9   FElement changedFE = this;
10   event FEChange{ x = newX;}
11  }
12  public void moveUp(int delta){
13   FElement movedFE = this;
14   event MoveUpEvent{ y = y + delta; }
15  }
16  public void makeEqual(Point other){
17   FElement changedFE = other;
18   event FEChange{
19    other.x = this.x; other.y = this.y;
20   }
21  }
22 }
23 public class Update{
24  FElement last;
25  public Update Update(){ register(this); }
26  public void update(FEChange next,
27                     FElement changedFE){
28   proceed(next); this.last = changedFE;
29   Display.update();
30  }
31  public void check(MoveUpEvent next,
32   FElement targetFE, int y, int delta){
33   if (delta < 100)){ proceed(next); }
34  }
35  when FEChange do update;
36  when MoveUpEvent do check;
37 }
```

Figure 1: Drawing Editor in Ptolemy

In companion papers [46, 47], we discussed the limitations of II and AO languages. To summarize, compared with AO languages, II languages have three limitations [47]. First, while subject modules are decoupled from observer modules, observer modules remain coupled with subjects. Second, their is no construct equivalent to AO "around advice" that allows to replace the code for an event. Instead, unnecessarily complex emulation code to simulate closures in languages such as Java and C# is required in II languages. Third, *quantification* can be tedious in II languages. The code that describes how each event is handled can grow in proportion to the number of objects from which implicit invocations are to be received.

Compared with II languages, AO languages have four limitations [47]. These limitations are not conceptual, rather, they stem from the fact that implementations of most current AO event models use PCDs based on pattern matching (on names [30], lexical structures [51, 20], program traces [17], etc). First problem is commonly known as the "fragile pointcut problem", which is caused by the use of pattern matching as a quantification mechanism [52, 57]. Pattern matching based PCDs are coupled to the code that implements the implicit event that they describe. Thus, seemingly innocuous changes break aspects [29]. Recent research results such

as Aspect Aware Interfaces (AAIs) [32], Crosscut Programming Interfaces (XPIs) [53, 27], Model-based Pointcuts [29], Open Modules (OM) [2], etc, have recognized and proposed to address the fragile pointcut problem, but none address it completely [47].

Second problem is that current AO event models do not implicitly announce some kinds of events [53, pp. 170]. Therefore, they also do not provide PCDs that select such events. Alternative AO approaches such as LogicAJ provide a finer-grained event model [51], however, PCDs in such techniques become strongly coupled with the structure of the base code and therefore become more fragile [47].

Third and fourth problems have to do with the interface for accessing contextual (or reflective) information about an event. In some AO approaches, this interface is fixed by the language designer and does not satisfy all usage scenarios [53]. In other AO languages (e.g. LogicAJ [51]), virtually unlimited reflective access to the context surrounding the lexical structure using meta-variables is possible but requires that the events form a regular structure [47]. Furthermore, contextual information that fulfills a common need (or role) in the handlers is not available uniformly to PCDs (and handlers) [47].

In Ptolemy, `evtype` declaration allow programers to declare named event types. An event type (`evtype`) declaration *p* has a return type, a name, and zero or more context variable declarations. These context declarations specify the types and names of reflective information communicated between announcements of events of type *p* and handler methods. These declarations are independent from the modules that announce or handle these events. The event types thus provide an interface that completely decouples subjects and observers. Events are explicitly announced using `event` expressions. These expressions enclose a body expression, which can be replaced by a handler. This functionality is similar in expressiveness to **`around`** advice in AO languages.

Finally, the names of `evtype` declarations can be utilized for quantification, which simplifies binding and avoids coupling observers with subjects.

3. EMPIRICAL EVALUATION APPROACH

Conducting an empirical evaluation of the software engineering properties of a new language design is a challenge. For example, claims such as "the quantification based on quantified, event types are less fragile compared to traditional syntactic quantification mechanisms" or that "quantified event-types improve the robustness of the handler code against base code changes, and makes it easier for the handlers to uniformly access reflective information about the event," may not be validated without large-scale use of the language design over an elongated period of time. Conclusions drawn from small examples, although helpful, may not correctly reflect the anticipated software engineering benefits of the language design.

A reasonable evaluation necessitates enough experience with the design to really say for sure it is right. In order to conduct such an evaluation, the pre-condition is the adoption of the language design in real world large-scale projects. Ironically, such adoption is often driven by demonstrated success of the language design in other projects. Taking these considerations into account, our proposed empirical evaluation technique is as follows:

3.1 Select Candidate Software Projects

The primary criteria for selecting a project as a candidate for our empirical evaluation technique is that they should be open source i.e. the source code is available for analysis, presence of existing version history that can be analyzed, large-size i.e. improved separation of concerns has real and perceived value in the context of this project, an active community contributing frequent releases and bug-fixes, which in turn translates to a rich change history in CVS.

For Ptolemy an additional requirement must be imposed. Our current Ptolemy infrastructure is based on Java, therefore, the project should be a Java project. A bonus would be an open and accepting community that can be persuaded to adopt based on demonstrated results.

Current candidate projects for Ptolemy include Eclipse [65], NetBeans [66], Azureus [64] and Ant [63]. We already have some experience with Eclipse [65], Azureus [64] and Ant [63] projects and their current build systems in the context of another research project [18, 19].

3.2 Select an initial version for candidate projects

An older version of the project is extracted from its repository and used as a baseline for the empirical evaluation. For example, a 2001 version of Eclipse with the sticky tag `v20011218` will be selected as the baseline for Eclipse. A challenge in this task is that often earlier versions of a software system use a different set of libraries, runtimes, etc. For example, all of our candidate projects made a transition from using Java 1.4 to Java 1.5 and then to Java 1.6 in the last few years. Recreating the build environment for such projects will be the key, however, once such a build environment is created, it could be reused for a number of candidate projects.

3.3 Use Concern Mining Techniques to Semi-automatically Identify Fragmented and Scattered Concerns for Modularization

After identifying an old version of a candidate project, we use automatic concern mining techniques on this version to extract fragmented and scattered concerns for modularization. Although automatic concern mining is still an emerging area, a number of techniques are available [5, 6, 61, 60]. One such technique by Breu, Zimmermann, and Lindig [7, 5, 6] is also applied to identify fragmented and scattered concerns in Eclipse [65]. Much of the reported results is directly applicable.

3.4 Use Ptolemy and Alternative II and AO Techniques to Modularize Identified Concerns

We then use the hints from automatic concern mining techniques to modularize the fragmented and scattered concerns in candidate projects. Three different starting versions are created in this step, one that uses implicit invocation techniques to modularize, a second that uses AO techniques to modularize, and a third that uses quantified, event types of Ptolemy.

3.5 Replay Changes on All Versions Using Version History

Once we have three versions (Ptolemy, AO and II) of the candidate project's old release, we will extract real changes from the project's CVS repository.

At least three frameworks are available today for this task APFL [62, 12], which is an open source framework for the ECLIPSE programming environment that facilitates the analysis of CVS archives, Kenyon [3], a common extraction, preprocessing, and storage platform for software configuration management and analysis, and Molhado [41], a configuration management and version control infrastructure and methodology.

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components (CDC)	This metric is defined as the total number of classes (including those that announce events, and those that provide handlers that register to these events) the contribute to the implementation of a concern and those that access these classes.
	Concern Diffusion over Operations (CDO)	This metric is defined as the total number of methods and handler methods that mainly contribute to the implementation of a concern and the number of other methods and handlers that access them.
	Concern Diffusions over LOC (CDLOC)	This metric is defined as the total number of points for each concern in the code where there is a transition from one concern to another.
Coupling	Coupling Between Components (CBC)	Total number of classes with which a given class is coupled.
	Depth Inheritance Tree (DIT)	This metrics essentially represents the depth of the inheritance hierarchy in an application.
Cohesion	Lack of Cohesion in Operations (LCOO)	Measures the lack of cohesion of a class in terms of the amount of method and handler methods that do not access the same instance variable.
Size	Lines of Code (LOC)	Total line of code in the application excluding comments.
	Number of Attributes(NOA)	Total number of attributes of each class.
	Weighted Operations per Component (WOC)	Total number of methods of each class and the number of its parameters.

Figure 2: The metrics suite to be used in this project and their definitions[9, 24, 21].

In the context of Ptolemy we use Molhado as it supports a higher-level of abstraction for analyzing changes between versions. For example, a variant of Molhado, MolhadoRef [13] has been used to replay refactorings of object-oriented programs to detect conflicts.

Extracted real changes in the candidate projects will be replayed to mimic software evolution. The key challenge in this activity is that Ptolemy's, II, and AO versions of candidate projects will differ from the baseline old version that we extracted from the repository in that some fragmented, scattered and tangled concerns are now modularized in this version. This is likely to create conflicts when we replay changes.

Our current insight into resolving this issue is to divide modularization of fragmented, scattered and tangled concerns as a set of refactorings. We also divide real changes exhibited in the project as refactorings.

We then use the MolhadoRef [13] tool to reconcile these refactorings with each other. As of today, we perceive this to be the biggest threat to the feasibility of our empirical evaluation.

3.6 Software Evolution Analysis and Metrics for Evaluation

The final step is then to analyze the effect of replaying the changes on all three versions of the candidate software project. This analysis will primarily use the set of metrics suite developed by Garcia *et al.* [24]. Garcia *et al.* [24] refined the object-oriented metrics proposed by Chidambar and Kemerer [9] and described by Fenton and Pfleeger [21] for advanced separation of concerns techniques.

We have adapted these metrics for evaluating II and Ptolemy's solutions. For AO solution, the original metrics defined by Garcia *et al.* [24] will be used. The adapted metrics and their definitions are shown in Figure 2.

The adapted definitions are closer to Chidambar and Kemerer's [9] original definition due to the unification of aspects and classes [50] in Ptolemy's language model.

4. CONCLUSION

In this position paper, we discussed a technique for empirical evaluation of software engineering properties of new language designs that claim to provide software engineers with new capabilities to modularize their concerns. The key idea behind the empirical evaluation is to use the real changes in a open source software project's life time to model software evolution. This promises to reduce the biases in the evaluation. With the help of advances in software repository mining techniques much of the evaluation process could be automated, which is an added advantage. We presented our technique in the context of Ptolemy's evaluation, however, it would be interesting to see whether it generalizes beyond II and AO-like language designs.

Acknowledgements

This work is supported in part by the National Science Foundation under grant CNS-06-27354.

5. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[2] J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. 2005 European Conf. Object-Oriented Programming (ECOOP 05)*, pages 144Ű–168, July 2005.

[3] J. Bevan, S. Kim, and L. Zou. Kenyon: A common software stratigraphy system. |http://dforge.cse.ucsc.edu/projects/kenyon|.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[5] S. Breu and T. Zimmermann. Mining aspects from version history. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[6] S. Breu, T. Zimmermann, and C. Lindig. Aspect mining for large systems. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 714–715, New York, NY, USA, 2006. ACM.

[7] S. Breu, T. Zimmermann, and C. Lindig. Mining eclipse for cross-cutting concerns. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 94–97, New York, NY, USA, 2006. ACM.

[8] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison Wesley, Reading, Mass., second edition, 1995.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[10] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[11] O.-J. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[12] V. Dallmeier, P. Weigerber, and T. Zimmermann. APFL: A preprocessing framework for eclipse. |http://www.st.cs.uni-sb.de/softevo/apfel/|.

[13] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware software merging and configuration management. 2007.

[14] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[15] E. W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[16] E. W. Dijkstra. On the role of scientific thought. *EWD 477*, August 1974.

[17] R. Douence, P. Fradet, and M. Sudholt. Trace-based aspects. *Aspect-oriented Software Development*, pages 141–150.

[18] R. Dyer, H. Narayanappa, and H. Rajan. Nu: preserving design modularity in object code. *SIGSOFT Softw. Eng. Notes*, 31(6):1–2, 2006.

[19] R. Dyer and H. Rajan. Modular program transformations for aspect-oriented constructs. Technical Report 434, Iowa State University, Department of Computer Science, July 2006.

[20] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *APLAS 04*, pages 366–381.

[21] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[22] M. Flatt and M. Felleisen. Units: cool modules for hot languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 236–248, New York, NY, USA, 1998. ACM Press.

[23] E. Gamma and T. Eggenschwiler. Jhotdraw: a java gui framework for technical and structured graphics. |http://sourceforge.net/projects/jhotdraw|.

[24] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM.

[25] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 31–44, London, UK, 1991. Springer-Verlag.

[26] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[27] W. G. Griswold, K. J. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software, Special Issue on Aspect-Oriented Programming*, Jan/Feb 2006.

[28] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 411–28. IEEE Comput. Soc, Los Alamitos, CA, October 1993.

[29] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP '06*, pages 501 – 525.

[30] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[32] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[33] D. E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974.

[34] B. B. Kristensen, O. L. Madsen, B. Muller-Pedersen, and K. Nygaard. Abstraction mechanisms in the beta programming language. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 1983. ACM Press.

[35] B. B. Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2(3):143–160, 1996.

[36] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Co., Boston, MA, USA, 1995.

[37] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.

[38] D. C. Luckham, J. J. Kennedy, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–54, April 1995.

[39] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[40] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[41] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 215–224, New York, NY, USA, 2005. ACM.

[42] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. IBM Research Report 21452, IBM, April 1999.

[43] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, December 1972.

[44] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–66, March 1985.

[45] H. Rajan. *Unifying Aspect- and Object-Oriented Program Design*. PhD thesis, The University of Virginia, Charlottesville, Virginia, August 2005.

[46] H. Rajan and G. T. Leavens. Quantified, typed events for improved separation of concerns. Technical Report 07-14, Iowa State University, Department of Computer Science, July 2007. In submission.

[47] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08: 22nd European Conference on Object-Oriented Programming*, July 2008.

[48] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[49] H. Rajan and K. Sullivan. Need for instance level aspect language with rich pointcut language. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, mar 2003.

[50] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[51] T. Rho, G. Kniesl, and M. Appeltauer. Fine-grained generic aspects. In *FOAL'06*. 2006.

[52] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.

[53] K. J. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *The Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 166–175, Sept 2005.

[54] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. *SIGSOFT Software Engineering Notes*, 15(6):22–33, December 1990.

[55] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–68, July 1992.

[56] P. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[57] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the aosd-evolution paradox. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, March 2003.

[58] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[59] N. Wirth. *Systematic Programming: An Introduction*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973.

[60] I. Yuen and M. P. Robillard. Bridging the gap between aspect mining and refactoring. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, page 1, New York, NY, USA, 2007. ACM.

[61] C. Zhang and H.-A. Jacobsen. Efficiently mining crosscutting concerns through random walks. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 226–238, New York, NY, USA, 2007. ACM.

[62] T. Zimmermann. Fine-grained processing of cvs archives with apfel. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 16–20, New York, NY, USA, 2006. ACM.

[63] Ant website. http://ant.apache.org/.

[64] Azureus website. http://azureus.sourceforge.net/.

[65] Eclipse website. http://www.eclipse.org/.

[66] NetBeans website. http://www.netbeans.org/.

Evaluating the Efficacy of Concern-Driven Metrics: A Comparative Study

Claudio Sant'Anna
Computer Science Department
Federal University of Bahia (UFBA), Brazil
santanna@dcc.ufba.br

Alessandro Garcia
Computing Department
Lancaster University, UK
garciaa@comp.lancs.ac.uk

Carlos J. P. Lucena
Computer Science Department
PUC-Rio, Brazil
lucena@inf.puc-rio.br

Abstract

The inadequate modularization of driving software concerns degrades design modularity. Software metrics are traditionally key mechanisms for detecting modularity-related design flaws. Conventional design metrics are based on the module abstraction. With the emergence of new programming techniques targeting superior concern modularity, a number of concern-sensitive metrics have been proposed for assessing these techniques. These metrics are based on the concern abstraction. However, there is not much empirical evidence on the efficacy of concern-driven metrics. It is also unknown if they promote superior modularity assessment or a valuable complement to conventional metrics. We present an empirical study that compares the efficacy of specific sets of concern-driven and conventional metrics to detect two of Fowler's bad smells. Our findings showed that the concern-driven metrics performed better than the conventional ones and are thus promising mechanisms for supporting modularity assessment.

Keywords software design; modularity; metrics, empirical software engineering

1. Introduction

The achievement of modular designs is far from being trivial as a multitude of widely-scoped concerns need to be simultaneously modularized. A concern is any important property or area of interest of a system that we want to treat in a modular way [9]. Business rules, distribution, persistence, security and caching are examples of concerns found in many software systems. Recent studies have shown that inadequate concern modularizations may lead to multiple design flaws [3, 11, 12, 15, 16]. For example, they can promote violations of important design principles, such as low coupling and narrow interfaces [3, 12, 15, 16].

The systematic assessment of modularity plays a pivotal role in the realm of software design. Software metrics are traditionally key mechanisms for assessing design modularity and identifying modularity-related design flaws [4, 17, 18]. Although typical modularity problems are related to the inadequate modularization of concerns, most of the current design metrics do not explicitly consider concern as a measurement abstraction. To date, design assessment has been mostly rooted at module-based metrics [4, 10]. The object-oriented metrics community has consistently used notions of class coupling, cohesion and interface size to derive measures of modularity [1, 2, 4, 10].

The recognition that concern-based assessment is important through software design activities is not new. In fact, with the emergence of new design decomposition approaches, such as aspect-oriented software development (AOSD) and feature-oriented programming [13], there is a growing body of relevant work focusing on concern analysis techniques [20]. In particular, a number of suites of metrics that can be qualified as "concern-driven" have been recently proposed [5-7, 21, 22, 24]. In addition, several empirical studies involving concern-driven metrics have been carried out [3, 6, 11, 12, 15, 16]. Most of these studies focus on either applying concern metrics to assess aspect-oriented designs or validating them as predictors of defects. However, there is not much knowledge on the efficacy of concern-driven metrics for identifying design flaws in comparison with conventional module-based metrics.

This paper presents a first exploratory study that compares the efficacy of concern-driven and conventional module-based metrics (herein called conventional metrics) on the identification of design anomalies. Sets of concern-driven and conventional metrics were applied to the object-oriented source code of a Web-based information system. The values of the different sets of metrics were separately analyzed by distinct groups of design reviewers in order to identify specific design flaws. The goal was to compare the number of classes correctly identified as suspects of exhibiting a design flaw by using the different kinds of metrics. The study focused on two design flaws, namely the shotgun surgery and divergent change bad smells [14]. The notion of bad smells was proposed in Fowler's book [14] to diagnose symptoms that may be indicative of something wrong in the design modularity. Our findings, although preliminary, suggest that concern-driven metrics are promising to

enhance modularity assessment. We also aim to learn some lessons from this study to support its future replication.

The reminder of this paper is organized as follows. Section 2 introduces the concept of concern-driven metrics. Section 3 describes the study procedures and configuration. Section 4 presents and discusses the results. The study constraints are addressed in Section 5. Section 6 describes related work. Section 7 concludes.

2. Concern-Driven Metrics

Concern-driven metrics are defined to capture modularity properties associated with the realization of concerns in software artifacts. This kind of metric allows the identification of specific design flaws or design degeneration caused by the poor modularization of concerns. Most of the existing concern-driven metrics [5, 6, 21, 22, 24] focus on quantifying the degree of concern scattering and tangling. Scattering is the degree to which a concern is spread over the design elements. Tangling represents the degree to which a concern is mingled with other concerns [9].

Concern-driven measurement approaches are based on a concern-to-design (or concern-to-code) mapping. This means that we have two domains related to each other through a mapping relationship. The source domain is a set of concerns and the target domain is a set of design elements, as illustrated in Figure 1 (inspired by a similar figure presented in [7]). The mapping consists of assigning a concern to the corresponding design elements that realize it. Therefore, before computing concern-driven metrics, it is necessary to identify the design elements responsible for implementing each concern in the system.

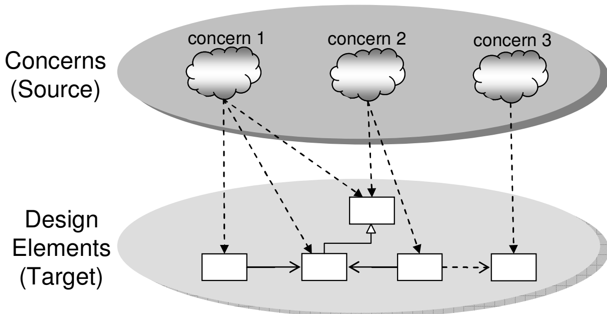

Figure 1. Mapping between concerns and design elements

In our empirical study, we evaluated the efficacy of three concern-driven metrics proposed by Sant'Anna and colleagues [21, 22]: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO) and Lack of Concern-based Cohesion (LCC). These metrics have been used in several studies with the goal of comparing object-oriented and aspect-oriented designs [3, 11, 12, 15, 16]. CDC and CDO measure the scattering of a given concern through the system components (classes and aspects) and operations, respectively. CDC counts the number of components that contribute to the realization of a given concern. CDO counts the number of operations that contribute to the realization of a given concern. The assumption behind these metrics is that a concern spread over a high number of design elements is detrimental to modularity. The understanding of a highly-spread concern demands the analysis of a large part of the design. In addition, a change related to that concern may affect a large number of design elements realizing the target concern.

LCC measures the cohesion of a given component in terms of the quantity of concerns addressed by it. Thus, it counts the number of concerns mapped to each component. The reasoning behind this metric is that a component that encompasses a large number of concerns is unstable. This is because it may suffer from modifications derived from change requests related to any of the concerns implemented by it.

To express the metrics unambiguously and facilitate the replication of our empirical study, we present the formal definition of the concern-driven metrics based on set theory. First, we present the terminology used on the formal definitions. Let S be a system, the classes and aspects of S are called as components and denoted by $C(S)$. Each component c consists of a set of attributes, denoted as $A(c)$, and a set of operations, represented as $O(c)$. The set of all attributes and all operations in system S are represented as $A(S)$ and $O(S)$, respectively. In classes, operations are methods, and, in aspects, operations represent methods and pieces of advice. For each $c \in C(S)$, the set of concerns assigned to c is denoted as $Con(c)$. Let $o \in O(c)$ be an operation of c, the set of concerns assigned to o is denoted as $Con(o)$. Let $a \in A(c)$ be an attribute of c, the set of concerns assigned to a is denoted as $Con(a)$. For each concern con realized on the design of system S, the set of components, operations and attributes to which con is assigned is, respectively, denoted as:

$C(con) = \{c \mid c \in C(S) \wedge con \in Con(c)\}$,
$O(con) = \{o \mid o \in O(S) \wedge con \in Con(o)\}$, and
$A(con) = \{a \mid a \in A(S) \wedge con \in Con(a)\}$.

The CDC, CDO and LCC metrics can now be defined as follows:

$$CDC(con) = \left| C(con) \cup \{c \mid c \in C(S) \wedge (O(c) \cap O(con)) \neq \emptyset\} \cup \{c \mid c \in C(S) \wedge (A(c) \cap A(con)) \neq \emptyset\} \right|,$$

$CDO(con) = |O(con)|$, and

$$LCC(c) = \left| Con(c) \cup \bigcup_{o \in O(c)} Con(o) \cup \bigcup_{a \in A(c)} Con(a) \right|$$

3. Study Settings

The goal of this study is to evaluate the efficacy of a set of concern-driven metrics to detect design flaws. In particular, the purpose of this study is to compare concern-driven and conventional metrics in order to learn which technique is the most effective to support the detection of two specific bad smells: *Shotgun Surgery* and *Divergent Change* [14].

3.1 Bad smells

The Shotgun Surgery and Divergent Change bad smells have been chosen because they are traditionally identified by the use of conventional metrics, mainly coupling and cohesion metrics [17, 18]. However, these bad smells are also believed to be a symptom of design flaws caused by poor modularization of concerns [19]. According to Fowler [14], the Shotgun Surgery bad smell is encountered when every time you make a kind of change, you also have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it is easy to miss an important change.

The Divergent Change bad smell occurs when one class is commonly changed in different ways for different reasons. When explaining the nature of divergent changes, Fowler [14] states: "If you look at a class and say, 'Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument', you likely have a situation in which two classes are better than one."

3.2 Target System

Our study involved the object-oriented design of a Web-based information system called Health Watcher [23]. This system supports the registration and management of complaints to the public health system. The main concerns involved in the Health Watcher design are: graphical user interface (GUI), business rules, concurrency, distribution, exception handling, and persistence.

This system was selected because it met a number of relevant criteria for our intended evaluation. First, it is a real system with an existing Java implementation. The first Health Watcher release was deployed in 2001 by the Public Health System in Recife, Brazil [23]. Since then, a number of incremental and perfective changes have been addressed in later Health Watcher releases.

Second, this system has served as a kind of benchmark for the assessment of contemporary modularization techniques, such as AOSD [12, 16, 23]. Third, the modularization of the concerns in the Health Watcher design has already been extensively studied. Fourth and foremost, the set of modularity-related design flaws for all the Health Watcher modules is well-known and has been investigated through the last 7 years by different developers and researchers. This means that we could rely on a reliable "oracle" when judging the metrics efficacy.

3.3 Subjects and Statistical Relevance

This study involved eight master students attending an Aspect-Oriented Software Development course at Lancaster University. It is important to highlight that we knew from the experiment design outset that the population was not representative and it would not allow us to gather statistically-relevant data. However, this study was a good opportunity to investigate if concern metrics are candidates to improve the state-of-the-art of design modularity assessment and, therefore, would call for additional research. It also allowed us and others to derive some lessons to replicate this study in the future.

The students were grouped in pairs. Each pair worked with different metrics in order to identify classes that were suspected of having one of the two bad smells in the object-oriented design of the Health Watcher system [23]. Two groups worked only with conventional metrics, one group only with concern-driven metrics, and the fourth group with both conventional and concern-driven metrics (hereafter referred as hybrid metrics group).

We estimated each student's relative ability from our previous knowledge of them and a background questionnaire they answered regarding the scope of their previous experience, particularly with regards to object-oriented programming and design, class diagrams, and software metrics. Then the pairs were formed to balance abilities. All the students had previous experience with object-orientation and class diagrams in academic contexts. Half of the students had 2 or more years of experience with professional software development. Two of them had experience with these techniques in the industry context. None of them had previous experience with software metrics.

3.4 Assessed Metrics

The concern-driven metrics group worked with the metrics Concern Diffusion over Classes (CDC), Concern Diffusion over Operations (CDO) and Lack of Concern-based Cohesion (LCC), described in Section 2. The conventional metrics group relied on the following metrics: Coupling Between Object Classes (CBO), Lack of Cohesion in Methods (LCOM), Weighted Methods per Class (WMC). The original definition of these metrics can be found in [4], and the formal definition of CBO and LCOM can be found in [1] and [2], respectively. The metrics Number of Attributes (NOA) and Number of Operations (NOO) were also used by the conventional metrics group. These metrics merely count the number of attributes and methods of each class. The hybrid metrics group worked with all eight metrics.

3.5 Activities

The study was preceded by a training session to allow the participants to familiarize themselves with the evaluated metrics and the target bad smells. At the beginning of the study execution, the participants were then given a document containing: (i) a partial view of the Health Watcher object-oriented design (class diagram), (ii) an introduction of Health Watcher functional and non-functional requirements, (iii) a brief explanation of the design, and (iv) a brief description of the concerns involved in the Health Watcher design. The document also described steps and guidelines the students should follow to conduct the study, the questions they should answer and information they should register.

In addition, we provided the students with the results of the metrics application. Each group only had access to the results referent to the metrics they were assigned to work with. Each group was asked to perform the following steps: (i) read the description of the Health Watcher design, (ii) based on the metrics results, identify the classes with the highest probability of having the bad smell Divergent Change, and (iii) based on the metrics results, identify the classes with the highest probability of having the bad smell Shotgun Surgery.

The time spent on each of these tasks was registered by each group. To identify the classes with bad smells, we asked them to reason about the metrics and identify which of them (one, some, or all) were relevant indicators based on the bad smell description. Also, we asked each group to explain which metrics they used for detecting the bad smell and which ones were not useful.

3.6 Hypothesis

The hypothesis we wanted to test in this study was that the hybrid metrics suite is the most effective to support detection of design bad smells. The intuition behind this hypothesis is that the hybrid metrics group is better equipped to identify the design flaws because both sets of conventional and concern-driven metrics: (i) are intuitively suited for detecting the selected bad smells, and (ii) may be used in a complementary way. To test this hypothesis, we compared the actual instances of the bad smells with the classes identified by each group as the strongest candidates of having the bad smells. Before the study was executed, we made a systematic analysis of Health Watcher artifacts to determine which classes were affected by the bad smells. First, we have used our own extensive knowledge of the system design and its releases. Second, we also checked out the source code of Health Watcher and observed comments and changes made by real developers while both refactoring the Java code and reengineering it with AspectJ [16, 23]. We identified twelve classes affected by Divergent Change and eight by Shotgun Surgery.

4. Results and Discussion

Table 1 shows for each group and each bad smell: (i) the time spent on the identification of the bad smell, (ii) the number and percentage of hits, and (iii) the number and percentage of false positives. A hit occurs when the group identified a class which was in the previously-generated list of classes affected by the bad smell (Section 3.5). A false positive occurs when the group identified a class which was not in the list. The percentage of hits is calculated dividing the number of hits by the number of classes in our list: 12 for Divergent Change, and 8 for Shotgun Surgery. The percentage of false positives is calculated dividing the number of false positives by the total number of classes identified by the group.

As far as the identification of classes affected by Divergent Change is concerned, we can observe in Table 1 that the two groups with conventional metrics performed significantly worse than the others. Both obtained only two hits (17%). Besides, 33% and 50% of the classes they indicated as suspects were false positives. Both conventional metrics groups reported that the most useful metric for identifying Divergent Change was Lack of Cohesion in Methods (LCOM). In fact, this metric presented high values for classes with no design anomaly. This metrics computes cohesion based on pairs of methods that access attributes in common. Because of this it erroneously classified classes with a high number of getters and setters methods as low cohesive, for instance.

Still regarding the Divergent Change bad smell, the group working with concern-driven metrics had 100% of hits, however 36% of false positives. This group reported that they used the Lack of Concern-based Cohesion (LCC) metric to identify Divergent Change. In fact, we did not limit the number of classes to be listed by the groups. So the concern-driven metrics group indicated a high number of classes as having Divergent Change (19 classes). This occurred because they listed all classes with LCC ≥ 2. However, a class addressing two concerns, for instance, does not necessarily represent a Divergent Change. Nevertheless, in the study guidelines, we asked the students to rank their list of classes with the ones with highest probability of having the bad smell coming first. The twelve first classes in the list of the concern-driven group were exactly the same as the previously generated list. The group working with the hybrid suite of metrics also performed well. This group had 75% of hits and no false positive. Lack of Concern-based Cohesion (LCC) and Lack of Cohesion in Methods (LCOM) were the metrics considered useful by this group. However, in this case, the presence of LCC minimized the previously discussed limitations of LCOM. Note that we fairly provided cohesion metrics to every group.

We can also observe from Table 1 that the group working with conventional metrics did not perform well regarding the identification of the Shotgun Surgery bad smell either. They had just 12% of hits. Besides, one of the groups had 75% of false positives and the other 33%. Differently from the analysis of Divergent Change, the performance of the hybrid metrics group was not good for Shotgun Surgery identification: 12% of hits and 75% of false positives. Apparently the reason for the low performance of these three groups is the same: some conventional metrics (in general, interface size metrics) might have introduced "noise" in the design assessment. Metrics such as Number of Operations (NOO), Number of Attributes (NOA) and Weighted Methods per Class (WMC) presented high values for classes not affected by the Shotgun Surgery bad smell. The group working with concern-driven metrics had again a high number of hits (75%), however a large number of false positives (65%). The reason for the high number of false positives was again the high number of listed classes. But again the correctly indentified classes were listed as having the highest probability of having the bad smell. This group reported that they used the Concern Diffusion over Components (CDC) metric to identify Shotgun Surgery.

The study results partially contradict our hypothesis that the hybrid metrics suite would be the most effective to support detection of design bad smells. This occurred mainly because of the results associated with the Shotgun Surgery bad smell. In spite of the high number of false positives, the concern-driven metrics suite was the most effective for identifying the assessed bad smells. Apparently the high number of metrics hindered the analysis made by the hybrid metrics group. As we can see in Table 1, the group working with these metrics took the longest time to finish their tasks. This might be because they spent too much time analyzing non-useful measures for the bad smells under assessment.

5. Study Constraints

This section discusses some constraints and imperfections discovered in the design and execution of this empirical study. The conclusions obtained here are restricted to the involved metrics, bad smells and the target software system. As discussed in Section 3.3, results regarding advantages and drawbacks in using concern-driven metrics obtained in this study should not be directly generalized to other contexts. However, this study allowed us to make useful evaluations on whether the use of concern-driven metrics for assessing design modularity would be worth studying further.

This study involves concern-driven metrics, thus they suffer from limitations related to the fact that the upfront process of assigning concerns to design elements directly impacts on the measurement results. Concern mappings and metrics collection were performed by the experiment designers (and not by the students). It was not our intention to assess the time spent on concern mappings, which it is matter to be addressed in future experiments. To make this process more systematic, we followed a number of procedures: (i) "pair mapping", where the mapping of concerns to design elements was done by two people assisting each other, (ii) consultation of the actual system developers whenever it was possible, and (iii) we followed a guideline that states that a concern should be assigned to a design element if the complete removal of the concern requires the removal or modification of the element [7].

Other issue that limits our findings is the fact that we played a crucial role in the definition of the oracle list, i.e. while deciding which classes were affected by the bad smells. This could have biased the results mainly because the evaluated concern-driven metrics were proposed by the authors of the experiment in previous works [21, 22]. To minimize this issue, this is why we have consulted and observed comments of the real developers of Health Watcher as well as changes made by them to improve the design.

6. Related Work

To the best of our knowledge, no other empirical study that explicitly compares concern-driven and conventional metrics has been undertaken yet. Several studies involving the two kinds of metrics have been carried out [3, 6, 11, 12, 15, 16]. However, these studies use concern-driven and conventional metrics in a complementary way to assess the modularity of a number of systems. In fact, most of them are dedicated to compare the modularity of aspect-oriented and object-oriented designs and implementations.

More recently, Eaddy et al [6] have carried out three experiments involving concern-driven metrics, including two of the metrics used in our study, namely Concern Diffusion

	Conventional Metrics	Conventional Metrics	Concern-driven Metrics	Hybrid Metrics
Divergent Change Identification				
Time (minutes)	9	10	21	31
Hits	2 (17%)	2 (17%)	12 (100%)	9 (75%)
False positives	1 (33%)	2 (50%)	7 (36%)	0 (0%)
Shotgun Surgery Identification				
Time (minutes)	6	10	13	35
Hits	1 (12%)	1 (12%)	6 (75%)	1 (12%)
False positives	3 (75%)	2 (33%)	11 (64%)	3 (75%)

Table 1. Results: identification of Divergent Change and Shotgun Surgery bad smells

over Components (CDC) and Concern Diffusion over Components (CDO). Their experiment aimed at testing the hypothesis that the more scattered a concern's implementation is, the more likely it is to have defects. They found a moderate to strong correlation between the evaluated metrics and defects for all three experiments, which suggested that scattering may cause or contribute to defects. Nevertheless, their experiments do not focus on the comparison between concern-driven and conventional metrics. In fact, their work only encompasses concern-driven metrics.

7. Final Remarks and Ongoing Work

The emergence of new modularization techniques that promise superior separation of concerns has bring the attention of software engineering researchers back to the importance of concern-sensitive assessment of software modularity. Hence, a number of concern-driven metrics have been defined. In addition, several empirical studies involving the use of concern-driven metrics have been undertaken. Most of these studies focus on assessing the benefits and drawbacks of emerging modularization techniques, such as aspect-oriented software development. However, there is not much empirical evidence on the efficacy of concern-driven measurement in order to assess design modularity.

This work represents a first stepping stone towards the evaluation of how concern-driven metrics enhance the process of detection of modularity-related design flaws. We presented an empirical study that compared the efficacy of concern-driven and conventional metrics. Our findings showed that concern-driven metrics are promising means for supporting the detection of modularity flaws. However, it was a preliminary study and it is clear that the number of involved subjects is by no means statistically relevant. Therefore, we have recently replicated this study with more undergraduate and master students (total of 30 students). We also included other bad smells in these new studies. We are still working on the analysis of the results, but at a first glance they seem similar to the ones presented here.

Acknowledgements

This work is supported in part by the European Commission grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

[1] Briand, L. et al. A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering, 25(1), pp. 91-121, 1999.

[2] Briand, L. et al. A Unified Framework for Cohesion Measurement in Object-Oriented Systems, Empirical Software Engineering - An International Journal, 3(1), pp. 65-117, 1998.

[3] Cacho, N. et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proc. of AOSD, Bonn, Germany, 2006.

[4] Chidamber, S., and Kemerer, C. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 1994, 476-493.

[5] Ducasse, S. et al. Distribution Map. Proc. of ICSM 2006, Philadelphia, USA, 203-212.

[6] Eaddy, M. et al. Do Crosscutting Concerns Cause Defects? IEEE Transactions on Software Engineering. (to appear), 2008, available at http://www.columbia.edu/~me133/.

[7] Eaddy, M. et al. Identifying, Assigning, and Quantifying Crosscutting Concerns, Proc. of ACoM'2007, in conjunction with ICSE'07, 2007.

[8] Workshop on AO Requirements Eng. and Arch. Design (Early Aspects), in conjunction with ICSE'2007. http://www.early-aspects.net/, 2007.

[9] Elrad, T.; Filman, R.; Bader, A. Aspect-Oriented Programming, Communication of the ACM, 44 (10), pp. 29-32, 2001.

[10] Fenton, N.; Pfleeger, S. Software Metrics: A Rigorous and Practical Approach, 2.ed. London: PWS, 1997.

[11] Figueiredo, E. et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. Proceedings of the 30th ICSE'08, Germany, pp. 10-18, 2008.

[12] Filho, F. et al. Exceptions and Aspects: The Devil is in theDetails. Proc. of Int'l Symposium on Foundations of Soft. Engineering (FSE), 2006.

[13] Filman, R. et al. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[14] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA, USA, 1999.

[15] Garcia, A. et al. Modularizing Design Patterns with Aspects: A Quantitative Study. Trans. on AOSD, 1, 2006, 36-74.

[16] Greenwood, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. Proc. of ECOOP'07, Berlin, Germany, 2007.

[17] Lanza, M., and Marinescu, R. Object-Oriented Metrics in Practice. Springer, 2006.

[18] Marinescu, R. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. Proc. of ICSM, 2004, 350 - 359.

[19] Monteiro, M., and Fernandes, J. Towards a Catalog of Aspect-Oriented Refactorings. Proc. of the AOSD, Chicago, 2005, 111-122.

[20] Robillard, M; Murphy, G. Representing Concerns in Source Code. ACM Transactions on Software Engineering and Methodology, 16(1), Article 3, February 2007.

[21] Sant'anna, C. et al. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework, Proc. of the Brazilian Symposium on Soft. Engineering, pp. 19-34, Brazil, 2003.

[22] Sant'anna, C. et al On the Modularity of Software Architectures: A Concern-Driven Measurement Framework, Proc. of the 1st European Conference on Software Architecture, Spain, 2007.

[23] Soares, S. et al. Implementing Distribution and Persistence Aspects with AspectJ. Proc. of OOPSLA' 02, pp. 174–190, 2002.

[24] Wong, W.; Gokhale, S.; And Horgan, J. Quantifying the Closeness between Program Components and Features. Journal of Systems and Software, pp. 87-98, 2000.

Assessing Contemporary Modularization Techniques for Middleware Specialization

Akshay Dabholkar
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
aky@dre.vanderbilt.edu

Aniruddha Gokhale
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
gokhale@dre.vanderbilt.edu

ABSTRACT

Middleware specialization is a technique to prune middleware features that are deemed unnecessary by the application domain, and to optimize and customize the relevant features to obtain domain-specific semantics within the middleware. Although contemporary modularization techniques, such as aspect-oriented programming (AOP) and feature-oriented programming (FOP), have been used in middleware specialization, there is a lack of a taxonomy that can assess the strengths and weaknesses of these techniques. To address these limitations, this paper develops a taxonomy that organizes contemporary modularization approaches applied to the problem of middleware specialization within a unified framework. The taxonomy helps assess the applicability of multiple modularization techniques used in concert for specializing system software such as middleware.

Categories and Subject Descriptors

H.4 [**Middleware**]: Specialization; D.2.8 [**Software Engineering**]: [complexity measures, performance measures]

1. INTRODUCTION

A number of applications based on distributed, general-purpose middleware platforms, such as J2EE/EJB, .NET Web Services and CORBA, must specialize these middleware to satisfy their functional and quality of service (QoS) requirements. Middleware specialization is a technique to prune middleware features that are deemed unnecessary by the application, and to optimize and customize the relevant features to obtain domain-specific semantics within the middleware.

Many prior research efforts in middleware specialization have used different modularization techniques, such as Aspect-oriented Programming (AOP) [10] and Feature-oriented Programming (FOP)- [23]. For example, AOP is used to develop resource-efficient system software [14] and context-specific specilized middleware [13], to bypass middleware layers [20], to define fine-grained middleware architectures that can be seamlessly refined [8], and even in dynamic specialization [4]. Modularization techniques have also been used in concert [35] with model-driven development.

Although different modularization techniques have shown how middleware can be specialized, there is a general lack of a taxonomy that helps to assess the strengths and weaknesses of modularization techniques when used individually or in concert. To address these limitations, this paper develops such a taxonomy which is derived from a survey of many research efforts, and is broadly classified along three dimensions of application development: feature-dependent, paradigm-dependent and lifetime-dependent. This taxonomy assesses the applicability of a specific technique for a particular context in middleware specialization. Such a taxonomy can also provide guidelines in specializing other kinds of systems software including operating systems and databases.

To effectively present the taxonomy and assessment of contemporary modularization techniques, we have organized the remainder of this paper as follows: Section 2 proposes our three-dimensional taxonomy of middleware specialization techniques. Section 3 brings forth a brief discussion on the different middleware specialization techniques and provides an assessment of their combinations through qualitative evaluations and guidelines for applying them. Finally, Section 4 concludes the paper and suggests possible future research directions.

2. TAXONOMY OF MODULARIZATION TECHNIQUES FOR MIDDLEWARE SPECIALIZATION

In this section we develop a taxonomy for assessing the different modularization techniques used in middleware specialization. We surveyed multiple research efforts on middleware specialization that use different modularization techniques, which in turn can be categorized with respect to the type of specialization it provides.

2.1 Survey of Modularization techniques used in Middleware Specialization

Lohmann et. al. [14] argue that AOP is well suited for the development of fine-grained and resource-efficient system software product lines where overhead due to the dynamic binding and dispatch of object-orientation is not acceptable when aspects beat objects. *FACET* [8] identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects. The advantage of using AOP is that the hooks and callbacks that were needed using standard object oriented techniques for adding functionality to existing code are no longer required. This removes the need to preconceive where the variation points of the code are needed and also removes the need to refactor large amounts of existing code to insert these hooks after the fact leading to better modularization.

Modelware [35] advocates the use of models and role-based aspect views and aspect libraries to separate intrinsic functionalities of middleware (*i.e.,* a set of coherent components free of crosscutting concerns) from extrinsic ones (*i.e.,* domain variations). This separation effectively lowers the concern density per component and fosters the coherence and the reuse of the components of middleware architectures. Devanbu et. al. [20] adopted AOP to enable bypassing layers of middleware to avoid the rigid layer processing performed by middleware that can lower overall system throughput, and reduce availability and/or increase vulnerability to security attacks. Their focus is to enable developers of middleware-based applications to conveniently adopt a *bypassing design pattern* (or *bypassing style*) (with good tool and run-time support) to speed-up applications, without having to write intricate, low-level, inherently non-portable code.

AOP, however, requires the specification of the middleware functionality which implies a broader knowledge of the structures and the application functionality. To support the runtime identification of the application needs and the *dynamic specialization* of the middleware according to the application requirements, *Aspect OpenOrb* [4] uses AOP to customize the reflective middleware. Aspects that are not in the application code can be dynamically inserted using the meta-object protocol of computational reflection.

FOCUS [13] describes how context-specific specializations can be automated and applied to optimize excessive generality in general-purpose middleware used for product-line architectures thereby improving throughput, average- and worst-case end-to-end latencies and predictability without affecting portability, APIs, or application software implementations while preserving interoperability.

The survey presented above enables us to develop a taxonomy as shown in Figure 1. The taxonomy can be broadly classified along three dimensions of application development: (1) feature-dependent, (2) paradigm-dependent, and (3) lifetime-dependent. The remainder of this section delves into the details of each dimension.

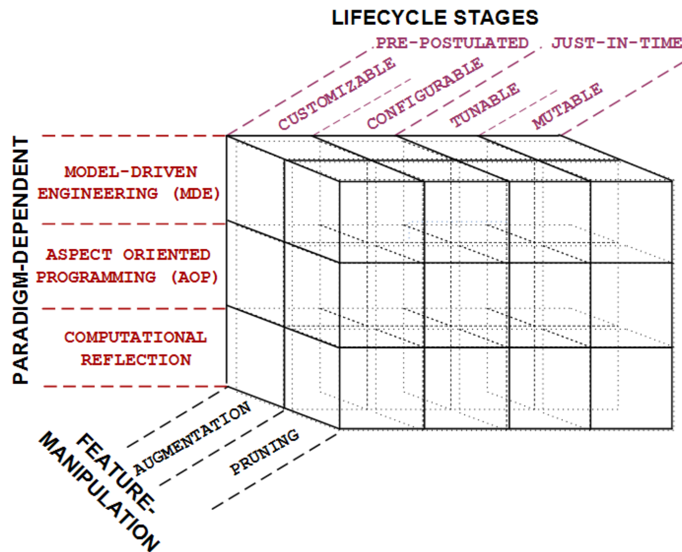

Figure 1: Three Dimensional Taxonomy of Middleware Specialization Research

2.2 Feature-Dependent Specialization

Feature-oriented programming (FOP) captures the variants of a base behavior though a layer of encapsulation of multiple abstractions and their respective increments that together pertain to the definition of a feature [16]. FOP decomposes complex software into features which are the main abstractions in design and implementation. They reflect user requirements and incrementally refine one another. FOP is particularly useful in incremental software development and software product lines (SPLs).

The specialization of a middleware platform along the feature-dependent dimension consists of composing it according to the features/functionalities required by the hosted applications. This is a dynamic process that consists of augmenting/inserting new features as well as pruning/removing unnecessary features. We distinguish between feature pruning and feature augmentation specialization strategies as follows:

2.2.1 Feature Pruning

Feature pruning is a strategy applied to remove features of the middleware to customize it. In this case the original middleware provides a broad range of features but many are not needed for a given use case. These unwanted features are pruned from the original middleware. This approach is taken by FOCUS [13] where unnecessary features are automatically removed from general purpose middleware through techniques such as memoization to provide optimizations for DRE systems.

2.2.2 Feature Augmentation

Feature augmentation is a strategy applied when the specialization is grounded via the insertion of new features, either because the original middleware did not support it or the middleware is composed out of building blocks [1, 3, 30]. The latter variety of middleware platforms are designed to overcome the limitations of monolithic architectures. Their goal is to offer a small core and to use computational reflection to augment new functionalities.

In Section 2.4.2, AOP can be used to compose middleware platforms where the middleware core contains only the basic functionalities [8, 35]. Other functionalities that implement specific requirements of the applications are incrementally augmented in the middleware by the weaver process, when they are required and decrementally pruned when they are not required.

2.3 Lifetime-Dependent Specialization

One approach to classify specialization techniques is based on the time scale at which it is implemented: *pre-postulated* and *just-in-time* [36]. Figure 2 shows this dimension of our taxonomy. If middleware specialization is performed during the application compile or startup time, we designate it *pre-postulated/static specialization*. For example, EmbeddedJava (`java.sun.com/products/em\-beddedjava`) minimizes the footprint of embedded applications during the application compile time. Similarly, if the middleware specialization is performed during the application run time, we designate it *just-in-time/dynamic specialization*. For example, MetaSockets [25] load adaptive specialization code during run time to adapt to wireless network loss rate changes. Notice that in Figure 2, dynamism increases from left to right.

2.3.1 Pre-postulated Specialization

Pre-postulated or Static specialization tailors the middleware before knowing its exact application use case. This process tries to identify the general requirements of possible future applications and defines the middleware configuration that will be used by the applications. It is further divided into customizable and configurable middleware.

- **Customizable specialization** enables adapting the middleware during the application compile/link-time so that a de-

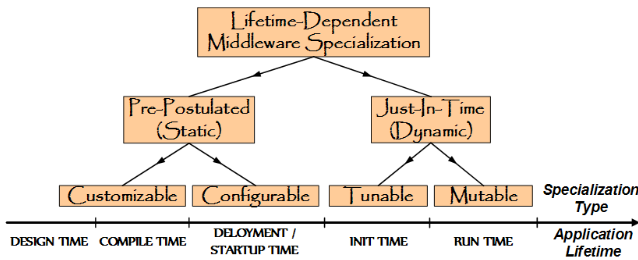

Figure 2: Lifetime-Dependent Middleware Specialization

veloper can generate specialized (adapted) versions of the application. Note that a customized version is generated in response to the functional and environmental changes realized after the application design-time. Examples of specialization mechanisms provided by customizable middleware are static weaving of aspects [10], compiler flags, and precompiler directives [11]. QuO [38] and EmbeddedJava are examples of customizable middleware.

- **Configurable specialization** enables adapting the middleware during the application startup time thereby enabling an administrator to configure the middleware in response to the functional and environmental changes realized after application compile time during its deployment or startup. Some examples of specialization mechanisms provided by configurable middleware include CORBA portable interceptors [19], optional command-line parameters, for example, to set socket buffer-size, and configuration files such as ORBacus configuration file (`www.orbacus.com`).

2.3.2 Just-in-time (JIT) Specialization

Just-in-time (JIT) or Dynamic specialization occurs at run time by identifying the requirements of the running application and customizing the middleware according to the application needs. It can be further classified into tunable and mutable middleware.

- **Tunable Specialization** enables adapting the middleware after the application startup time but before the application is actually being used. Doing so enables an administrator to fine-tune the application in response to the functional and environmental changes that occur after the application is started. Examples of specialization mechanisms provided by tunable middleware are "two-step" specialization approaches (including static AOP during compile time and reflection during run time) employed by David et. al [6] and Yang et. al [34], the component configurator pattern [27] used in DynamicTAO [12], and the virtual component pattern [5] used in TAO and ZEN middleware.
- **Mutable Specialization** is the most powerful type of middleware specialization that enables adapting an application during run time. This specialization is also called Adaptive Specialization. Hence, the middleware can be dynamically specialized while it is being used. The main difference between tunable middleware and mutable middleware is that in the former, the middleware core remains intact during the tuning process whereas in the latter there is no concept of fixed middleware core. Therefore, mutable middleware are more likely to evolve to something completely different and unexpected. Examples of specialization techniques provided by mutable middleware are reflection [3], late composition of components [11], and dynamic weaving of aspects [34].

2.4 Paradigms-Dependent Specialization

Numerous advances in programming paradigms have also contributed to middleware specialization techniques. Although many important contributions have been made in this area, a review of the literature shows that four paradigms, in addition to object-oriented paradigm, play key roles in supporting middleware specialization: computational reflection [4], component-based design [29], aspect-oriented programming [10], and feature-oriented programming [23].

There are other approaches such as program slicing, partial evaluation, policies, automatic tuning of configuration parameters that enable customization of system software. However these approaches are more fine-grained in the sense that they are used to manipulate, customize and verify the correctness of individual programs. However, each of these approaches can be utilized through the more coarser-grained approaches that are being considered in this paper.

2.4.1 Computational Reflection

Computational reflection [4] refers to the ability of a program to reason about, and possibly alter, its own behavior. Reflection enables a system to *open up* its implementation details for such analysis without compromising portability or revealing the unnecessary parts. As depicted in Figure 3, a reflective system (represented as base-level objects) has a self representation (represented as meta-level objects) that is causally connected to the system meaning that any modifications either to the system or to its representation are reflected in the other.

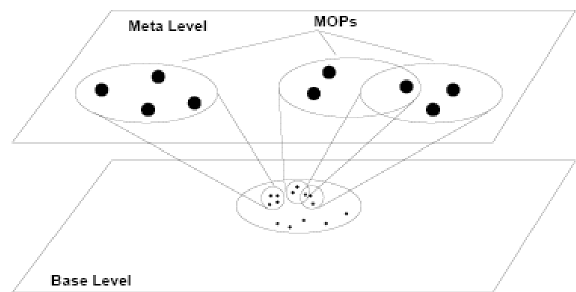

Figure 3: A Reflective System with Causally Connected Meta-level

The base-level part of a system deals with the *normal* (functional) aspects of the system whereas the meta-level part deals with the computation (implementation) aspects of the system. The meta-level contains the building blocks responsible for supporting reflection. The elements of the base-level and that of the meta-level are, respectively, represented by base-level objects and meta-level objects. A meta-object protocol (MOP) [9] is a meta-level interface that enables *systematic* (as opposed to ad hoc) inspection and modification of the base-level objects and abstraction of the implementation details.

Computational reflection is an efficient and simple way of inserting new functionalities in a reflective middleware. Thus, it is necessary only to know components and interfaces. The next generation middleware [3, 7] exploits computational reflection to customize the middleware architecture. Reflection can be used to monitor the middleware internal (re)configuration [24]. The middleware is divided in two levels: base-level and meta-level. According to Figure 3, the middleware core is also represented by base-objects and new functionality is inserted by meta-objects. Figure 4 shows that the meta-level is orthogonal to the middleware and to the application. This separation allows the specialization of the middleware via extension of the meta-level.

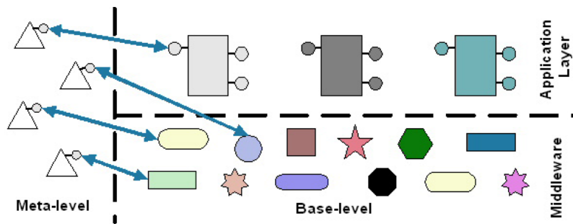

Figure 4: Reflective Middleware

2.4.2 Aspect Oriented Programming (AOP) Techniques

Kiczales et al. [10] realized that complex programs are composed of different intervened crosscutting concerns (properties or areas of interest such as QoS, energy consumption, fault tolerance, and security). While object-oriented programming abstracts out commonalities among classes in an inheritance tree, crosscutting concerns are still scattered among different classes thereby complicating the development and maintenance of applications.

AOP [10] applies the principle of "separation of concerns" (SoC) [21] during development time in order to simplify the complexity of large systems. Later, during compile or run time, an aspect weaver can be used to weave different aspects of the program together to form a program with new behavior. AOP proponents argue that disentangling the crosscutting concerns leads to simpler development, maintenance, and evolution of software. Naturally, these benefits are important to middleware specialization. Moreover, AOP enables factorization and separation of crosscutting concerns from the middleware core [28], which promotes reuse of crosscutting code and facilitates specialization.

In the context of middleware, we refer to AOP approaches as existing software platforms that expose hooks for applications using these platforms, to adapt, alter, modify, or extend the normal execution flow of a service requested. Non-functional features (monitoring code, logging, security checks, etc.) can be transparently woven into the middleware code paths or unnecessary features can be pruned through bypassing code paths or middleware layers. In that sense, the CORBA portable interceptor (PI) mechanisms, although not explicitly positioned as an aspect-oriented approach, belong to this category. Using AOP, customized versions of middleware can be generated for application-specific domains. Yang et al. [34] and David et al. [6] both provide a two-step approach to dynamic weaving of aspects in the context of middleware specialization using a static AOP weaver during compile time and reflection during run time. Other recent examples explicitly positioning themselves as aspect-oriented approaches are the JBoss AOP approach (`www.jboss.org`) and the Spring AOP approach (`www.springframework.org`).

2.4.3 Model-Driven Engineering (MDE)

MDE is an emerging paradigm that integrates model-based software development techniques (including Model-Driven Development [26] and the OMG's Model Driven Architecture) with QoS-enabled component middleware to help resolve key software development and validation challenges encountered by developers of large-scale distributed, real-time and embedded (DRE) middleware and applications. In particular, MDE tools can be used to specify requirements, compose DRE applications and their supporting infrastructure from the appropriate set of middleware components, synthesize the metadata, collect data from application runs, and analyze the collected data to re-synthesize the required metadata. These activities can be performed in a cyclic fashion until the QoS constraints are satisfied end-to-end.

Conventional middleware architectures suffer from insufficient module-level reusability and the ability to adapt in the face of functionality evolution and diversification. As reported in [35], *"intrinsic"* and *"extrinsic"* properties interact non-modularly in conventional middleware architectures. Consequently, middleware architects are faced with immense architectural complexities because the concern density per module is high. The code-level reusability of the *"common abstractions"* is also drastically reduced because the generality of intrinsic components is restricted by the *"extrinsic"* properties in the face of domain variations. A contributing factor to this complexity, is that the code-level design reusability in conventional middleware architectures is incapable of adequately dealing with *"change"* in two dimensions: time (functional evolution) and space (functional diversification).

The reusability in conventionally developed software components is insufficient due to the lack of explicit means to effectively distinguish intrinsic and extrinsic architectural elements. Conventional middleware architectures also lack effective means to reuse *"extrinsic"* properties, especially ones that are crosscutting [10] in nature, *i.e.*, not localized within modular boundaries. Conventional architectures have fallen short of doing so because they are incapable of componentizing and reusing crosscutting concerns as analyzed in [37]. Being able to componentize and to reuse these functionalities tremendously facilitates the construction of middleware systems. To tackle the aforementioned problems, Zhang et. al. [35] propose a new architectural paradigm called Modelware which embodies the *"multi-viewpoints"* [18] approach.

3. ASSESSMENT OF MODULARIZATION TECHNIQUES FOR MIDDLEWARE SPECIALIZATION

In this section we use our taxonomy to assess the strengths and weaknesses of various modularization approaches used for specializing middleware. We then develop a framework for systematic and automated middleware specialization that provides guidelines for middleware application developers to reason about, optimize, customize and tune the middleware according to their domain-specific requirements.

3.1 Qualitative Evaluation of the Middleware Specialization Taxonomy

In the following we use a combination of artifacts of individual dimensions of our taxonomy to assess the pros and cons of various modularization techniques when applied to middleware specialization.

Table 1 summarizes our assessment of different modularization techniques. We briefly discuss below each paradigm with respect to the lifetime dimension of the taxonomy

1. **Pre-postulated Specializations:** FOP, AOP and MDE are widely used at design-time and compile-time respectively to perform feature augmentation and pruning. Although feature modules – the main abstraction mechanisms of FOP – perform well in implementing large-scale software building blocks, they are incapable of modularizing certain kinds of crosscutting concerns [2]. This weakness is the strength of aspects. Caesar [15], AFMs [2] combine FOP with AOP to overcome the shortcomings of "purely hierarchial" feature specifications in FOP. However, reflection has limited application during the pre-postulated phases except during deployment it could be used to inspect the target platform features before the application is deployed.

Table 1: Evaluation of the Combinations of Dimensions

COMBINATIONS	USE CASES	STRENGTHS	WEAKNESSES	RELATED WORK
Pre-postulated + Reflection	Inspect target platform features	Useful during deployment	Difficult to predict runtime conditions	AspectOpenORB, DTO
Just-in-time + Reflection	Introspect runtime application features	Dynamic Adaptation & reconfiguration	Can cause unpredictable behavior	AspectOpenORB
Pre-postulated + AOP	Weave/Prune at compile-time	Transparency without affecting core	Code Bloating	FACET, CLA, FOCUS, Bypassing Layers, AspectOpenORB
Just-in-time + AOP	Dynamic weaving of features	Dynamic Adaptation	Requires native platform support	JAsCo, PROSE, Abacus
Pre-postulated + MDE	Weave/Prune only known features	Elegant design	Runtime specializations not possible	DTO, CLA, Modelware
Just-in-time + MDE	Self-healing/correcting systems	Validation of Specializations	Incur runtime overhead	Models@Runtime
AOP + FOP	ISD and SPLs	Better modularization of crosscutting features	Runtime specializations not possible, cause conflicts	AFMs, Caesar
FOP + MDE	SPLs	Better composition of features	Runtime specializations not possible, cause conflicts	FOMDD [31]
AOP + Reflection	Composition based on application requirements	On-demand feature weaving	May cause conflicts	AspectOpenORB
AOP + MDE + FOP + Reflection	Design/Weave/Prune valid features combinations	Systematic, correct specialization process	Safe specializations is challenging	*Research Needed*

2. **Just-in-time Specializations:** AOP has few use cases at just-in-time where dynamic weaving of feature aspects could be set up with the help of native compile-time platform support, such as Java Virtual Machine (JVM) [22]. JAsCo [32] is an adaptive AOP language used to specialize Web Services implementations [33] whereas PROSE [17] and Abacus [36] are just-in-time aspect-based middleware. Beyond design-time, MDE cannot be applied since it relies mainly on predetermined system feature requirements. However, it can configure dynamic augmentation or pruning of features at runtime. Recently models at run-time has been used for self-healing systems. The principles from those domains need to be applied for specializing middleware dynamically based on models. Computational reflection can be used to support the runtime introspection of the application and perform dynamic augmentation and pruning of features to adapt its internal implementation and reconfigure itself depending upon the dynamic conditions prevalent at run-time. However, This enables support for more powerful dynamic specializations which are useful for power and resource management, and dynamic adaptation as in wireless sensor networks, embedded systems, etc.

3.2 Guidelines for Middleware Specialization

We now provide guidelines for middleware specialization using our taxonomy. We use the lifecycle dimension as the dominant dimension since it imparts a systematic ordering to the process of performing middleware specialization. We believe the guidelines can apply to any systems software, such as an operating system, web server or a database management system.

1. **Development-time specializations:** During development-time the middleware developer can program the application code with features that need to be loaded at initialization-time and features that can be swapped in/out at run-time through strategies. MDE and AOP based techniques are more effective to program development-time specializations. In this phase, feature-augmentation should be the goal.
2. **Compile-time specializations:** Compile-time specializations can be used to transparently weave-in (augment) or weave-out (prune) features code. AOP is the key enabler for performing compile-time specializations.
3. **Deployment-time specializations:** Deployment-time specializations mainly address target platform-specific concerns such as type of data transport, database drivers, etc. The middleware features are matched to make optimal use of the underlying platform feature constraints. Special tools which perform the task of setting up the deployment can use reflection to query the platform features and use AOP to transparently change the underlying bindings or supply the required configuration parameters when launching applications.
4. **Initialization-time specializations:** Feature configuration is performed at initialization-time using the configuration parameters that are pre-programmed either at development-time and/or compile-time or supplied during the application startup-time.
5. **Run-time specializations:** At run-time, features can be swapped in or out using either reflection or dynamic aspect weaving depending upon the conditions prevalent after the application is executing. However, too much dynamism can lead to unpredictable application behavior leading to unstable specializations that are difficult to verify for safety criticality and correctness. To benefit from mutable middleware, we should harness its power using techniques such as safe specialization. So most of the dynamic feature swapping needs to be statically programmed before hand.
6. **Integrated specializations:** Since no single modularization technique can specialize middleware over all phases of the application lifetime, multiple techniques need to be applied and validated in unison starting with MDE and AOP at pre-postulated time whereas computational reflection at just-in-time. It is important to restrict feature changes at run-time that conflict with the design-time feature configurations. Ap-

plying overlapping specializations may cause inconsistencies in the applications. This is the same problem as the feature interaction problem in pattern recognition that needs to be addressed in middleware specialization also. Inconsistency can be caused when FOP, AOP or MDE augments a dependent feature set during pre-postulated phases but reflection prunes one of the features from the set during just-in-time phases which may lead to unpredictable runtime behavior and failures. Inconsistencies can also occur within the same life-time phase. Hence, tools and techniques are needed to validate specializations when multiple customization techniques are applied in tandem not only within a phase but across entire application lifetime.

7. **Optimal specializations:** Finally specialization tools should not only validate but also optimize various feature changes so that they are not only consistent but satisfy the quality of service (QoS) requirements of the applications.

4. CONCLUDING REMARKS

Prior research has shown the usefulness of different modularization techniques to handle middleware specialization challenges. Yet there does not exist a common vocabulary that unifies these efforts. This paper addresses this challenge by developing a three-dimensional taxonomy for middleware specialization. We use this taxonomy to provide a qualitative assessment of the strengths and weaknesses of the modularization techniques. We also provide guidelines for application or middleware developers who are interested in specializing the middleware on how best to use this taxonomy in their project.

Finding an optimized and adaptive middleware specialization solution using current state-of-the-practice middleware specialization approaches is not an easy task. A developer needs to know all available middleware approaches and should spend a lot of time and money to find the optimized solution. Developing tools, techniques and high-level paradigms which can be assimilated into a catalog of specialization patterns that assist a developer in this tedious process is a useful research area that promotes development of adaptive software. Inventing domain-specific specialization pattern languages can serve as guidelines for the synthesis of such tools. Moreover, validating the safety of specialization approaches is hard. Our current work focuses on this dimension of the research.

5. REFERENCES

[1] Gul A. Agha. Introduction. *Communications of the ACM*, 45(6):30–32, 2002.

[2] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *Software Engineering, IEEE Transactions on*, 34(2):162–180, March-April 2008.

[3] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 191–206, London, 1998. Springer-Verlag.

[4] Nélio Cacho and Thaís Vasconcelos Batista. Using AOP to Customize a Reflective Middleware. In *OTM Conferences (2)*, volume 3761 of *Lecture Notes in Computer Science*, pages 1133–1150. Springer, 2005.

[5] Angelo Corsaro, Douglas C. Schmidt, Raymond Klefstad, and Carlos O'Ryan. Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications. In *Proceedings of the* 9th *Annual Conference on the Pattern Languages of Programs*, Monticello, IL, September 2002.

[6] Pierre-Charles David, Thomas Ledoux, and Noury M.N. Bouraqadi-Saadani. Two-step Weaving with Reflection using AspectJ. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, October 2001.

[7] Fábio M. Costa and Gordon S. Blair. A Reflective Architecture for Middleware: Design and Implementation. In *ECOOP'99, Workshop for PhD Students in Object Oriented Systems*, June 1999.

[8] Frank Hunleth and Ron K. Cytron. Footprint and Feature Management Using Aspect-oriented Programming Techniques. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 02)*, pages 38–45. ACM Press, 2002.

[9] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The art of metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[11] Raymond Klefstad, Douglas C. Schmidt, and Carlos O'Ryan. Towards Highly Configurable Real-time Object Request Brokers. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Newport Beach, CA, March 2002. IEEE/IFIP.

[12] F. Kon, M. Roman, P. Liu, J. Mao, T Yamane, L. Magalhaes, and R. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, April 2000.

[13] Arvind Krishna, Aniruddha Gokhale, Douglas C. Schmidt, John Hatcliff, and Venkatesh Ranganath. Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures. In *Proceedings of EuroSys 2006*, pages 205–218, Leuven, Belgium, April 2006.

[14] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. *Transactions on AOSD II*, 4242:227–255, 2006.

[15] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[16] Mira Mezinia and Klaus Ostermann. Variability Management with Feature-oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, 2004.

[17] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. *SIGOPS Oper. Syst. Rev.*, 42(4):233–246, 2008.

[18] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Softw. Eng.*, 20(10):760–773, 1994.

[19] Object Management Group. *Interceptors FTF Final Published Draft*, OMG Document ptc/00-04-05 edition, April 2000.

[20] Ömer Erdem Demir, Prémkumar Dévanbu, Eric Wohlstadter, and Stefan Tai. An Aspect-oriented Approach to Bypassing

Middleware Layers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 25–35, New York, NY, USA, 2007. ACM Press.

[21] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.

[22] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, pages 100–109, Boston, Massachusetts, 2003.

[23] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241, pages 419–443, Jyväskylä, Finland, 9–13 1997. Springer.

[24] Manuel Roman, Roy H. Campbell, and Fabio Kon. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5), July 2001.

[25] S. Sadjadi, P. McKinley, and E. Kasten. Architecture and operation of an adaptable communication substrate, 2003.

[26] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[27] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[28] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Commun. ACM*, 44(10):95–97, 2001.

[29] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[30] Anand Tripathi. Challenges Designing Next-Generation Middleware Systems. *Communications of the ACM*, 45(6):39–42, June 2002.

[31] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature oriented model driven development: A case study for portlets. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 44–53, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive Programming in JAsCo. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 75–86, Chicago, Illinois, 2005.

[33] Bart Verheecke and María Agustina Cibrán. Aop for dynamic configuration and management of web services. In *In Proceedings of 2003 International Conference on Web Services*, page 2004, 2003.

[34] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 85–92, New York, NY, USA, 2002. ACM.

[35] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Generic Middleware Substrate Through Modelware. In *Proceedings of the 6th International ACM/IFIP/USENIX Middleware Conference*, pages 314–333, Grenoble, France, 2005.

[36] Charles Zhang, Dapeng Gao, and Hans-Arno Jacobsen. Towards Just-in-time Middleware Architectures. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2005. ACM Press.

[37] Charles Zhang and Hans-Arno Jacobsen. Resolving Feature Convolution in Middleware Systems. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 188–205, New York, NY, USA, 2004. ACM.

[38] John A. Zinky, David E. Bakken, and Richard Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.

A Close Look at Composition Languages

Florian Heidenreich Jendrik Johannes
Uwe Aßmann

Lehrstuhl Softwaretechnologie
Fakultät Informatik
Technische Universität Dresden, Germany
{Florian.Heidenreich,Jendrik.Johannes,Uwe.Assmann}
@tu-dresden.de

Steffen Zschaler

University of Lancaster
Computer Science Department
Lancaster, United Kingdom
szschaler@acm.org

Abstract

A large number of different composition systems and techniques have been developed over the last years. To compare their relative benefits and drawbacks, we need a common vocabulary for describing elements of composition systems. This paper contributes to the search for such a vocabulary by taking a closer look at the structure of composition languages—that is, languages used for describing compositions—based on a survey of eight different composition systems.

Categories and Subject Descriptors A.1 [*Introductory and Survey*]; D.2.11 [*Software Architectures*]: Languages, Patterns

General Terms Design, Theory

Keywords Elements of Composition Systems, Composition Languages, Terminology

1. Introduction

Decomposition and modularisation have been used successfully to deal with complex system development. Whenever we decompose a system design or implementation, we also need ways of putting the parts together again. Typically, this is achieved by using so-called composition systems. Over time, a large number of composition systems and approaches have been developed. To be able to compare these, we need a common vocabulary describing salient elements of composition systems. This paper contributes to the ongoing search for such common vocabulary.

Composition systems and their elements have been studied for some time. (Medvidovic and Taylor 2000), were the first to discuss the elements of a composed system. They positioned their discussion in the context of Architecture Description Languages (ADLs) and found components, connectors and configurations as the central elements of an ADL. Later, (Aßmann 2003) introduced a more general characterisation of composition systems by distinguishing the elements of component model, composition technique, and composition language. In this paper, we focus on the internal structure of a composition language and define a common vocabulary for elements of a composition language; to the best of our knowledge the first attempt in this direction.

The remainder of this paper is structured as follows: In the next section, we propose new vocabulary for analysing composition descriptions. To evaluate whether this vocabulary is sufficiently general for a diverse set of composition systems, Sect. 3 then applies it to eight specific composition systems. Finally, Sect. 4 concludes the paper.

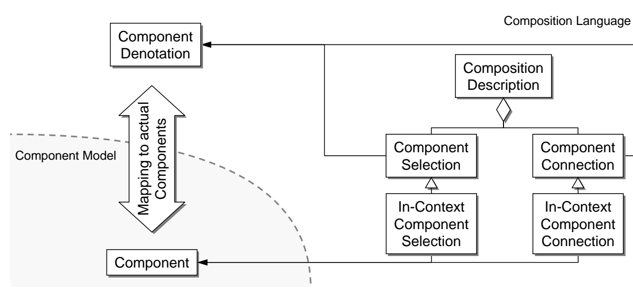

Figure 1. Elements of composition languages

2. Elements of Composition Languages

Figure 1 shows the elements of a composition language as we see them. In the lower left corner are the actual components. We do not care about the specific form in which these components exist: they could be source or binary components, or even, for example, just logical subdivisions of a large monolithic model.

For each of these components, we have a denotation in our composition language. Typically, we use these denotations to express compositions. Component denotations can take many forms: They can be simple names that stand for components, but they can also be symbols that need to be mapped to components in more complex ways—for example, we could use `#ifdef` symbols to denote components in the configuration of a C++ program. To handle this wide range of possible component denotations, a mapping between the denotations and the actual components is required. If denotations are only names, this mapping is simple. However, it can become arbitrarily complex in other cases.

Compositions are described in two steps: First, we need to select the components to be composed. Then, we need to describe how these components are to be connected. Depending on the composition system, both steps will be expressed implicitly on the component level or explicitly on the level of component denotations.

In the following, we define these concepts in more detail.

2.1 Component

We use the term *component* to refer to artefacts or groups of artefacts that will eventually form part of the composed system. Usually, one (ideally reusable) concern is contained in one component. Components need not be clearly identifiable physical entities: A single file may form a component, but a set of related model elements from different models may also form one component. Which information is considered as a component depends on the concrete composition system in use. Components can have hierarchical structure; that is, one component may contain other components.[1]

2.2 Component Denotation

Component selections and connections are often expressed using abstract representations of the actual component. We use the term *component denotation* to refer to such abstract representations. Component denotations can be simple names that stand for individual components, but they can also be complex structures representing components and certain selected properties of these components.

2.3 Mapping from Denotation to Component

The meaning of component denotations is defined by mapping them to actual components. Such mappings can be simple mappings that associate a symbol in the component denotation with a single-artefact component or they can be complex mappings that associate a single component-denotation symbol with a group of artefacts, possibly from different physical containers (e.g., files). Mappings can be defined extensionally by enumerating all denotational elements and their associated components or intentionally through rules that compute the components associated with a denotational element.

2.4 Component Selection

To describe how a system is composed from component, we must first select the components to be used. We use the term *component selection* to refer to this step and, in particular, to the part of a composition program expressing this. Component Selection normally uses component denotations to reference the components to be selected. A special kind of component selection—in-context component selection—is inserted directly in the definition of one component and implicitly selects this component. It uses component denotations to select additional components, though.

2.5 Component Connection

After selecting the components to be composed, we need to specify the connection between them. We call this step *component connection* and, also use this term for the part of a composition program expressing this. Normally, component connections use component denotations to reference all components to be connected. A special kind of component connection—in-context component connection—can be used directly inside the definition of one component and automatically references this component and specific implicit connection points within it. It uses component denotations to reference other components to be connected.

3. A Survey of Composition Systems

In this section, we survey a number of composition systems and show how they fit with the vocabulary we introduced in the previous section.

This survey serves two purposes: 1) to explain the individual terms of our vocabulary by providing concrete examples, and 2) to evaluate our vocabulary by showing that it can be applied to a sufficiently large set of composition systems.

3.1 Black-Box Composition

Classical, black-box component systems, such as CORBA (Object Management Group 2008) or Enterprise Java Beans (EJBs) (Microsystems 2001), have been available for some time. They typically provide an infrastructure that manages binary-format components and establishes connections and communications between them at runtime. One important part of this infrastructure is a naming service that allows components to be identified through some developer-chosen name. Components connect with each other by referencing these names in their source code or in so-called deployment descriptors.

Components come in the form of binary files. Depending on the specific component systems, these are machine-code binaries such as executables or dynamically linked libraries in the case of CORBA or byte-code binaries as in the case of EJB.

[1] The definition is generic on purpose: We focus on composition *languages* and are not overly interested in the details of what constitutes a component.

Component Denotation is done using textual identifiers. Some component systems support hierarchical name spaces and structured names. If so-called trading services are used, components can also be denoted by logical expressions over their properties.

Mapping Standard mapping is performed in the naming service. This mapping is explicitly defined by registering individual components with the naming service under a specific name. In the case of trading services, the mapping is more complicated and involves some amount of reasoning and matchmaking over the divers logical description terms. Still, however, the mapping is explicit insofar as each component is explicitly associated with a number of logical description terms.

Component Selection happens explicitly by naming associated components either in calls to the naming service or in so-called deployment descriptors that are delivered with a component. In both cases, all associated components are mentioned in the context of the component using them. Therefore, component selection is in-context.

Component Connection is combined with component selection. Therefore, it also happens in-context.

3.2 Aspect-Oriented Programming

In Aspect-Oriented Programming (AOP) (Kiczales et al. 1997), pieces of code—advices—are distributed over a core set of modules in a weaving process. Aspect definitions use pointcut specifications (that is, specifications of sets of joinpoints—points in the execution of the core modules) to specify where advice code should be woven. What joinpoints are available depends on the execution model of the programming language of the core modules and on the specific aspect approach.

Components are the core modules and the individual pieces of advice.

Component Denotation Advice components are denoted by simple names that developers associate with an aspect definition. Core-module components are denoted through a rich set of pointcut specifications referring to individual points in the execution of the core modules.

Mapping of advice denotations is implicit: A name used is mapped to all the advice of an aspect definition of the same name. Mapping core modules is also implicit: How pointcut expressions are mapped to specific points in the execution is hard-coded in the aspect weaver.

Component Selection Advice components are selected explicitly by making them available to the aspect weaver. Core module components are selected explicitly through pointcut expressions.

Component Connection is defined explicitly by associating a particular piece of advice with a particular pointcut expression. It is in-context, because pointcuts are expressed directly in the context of an aspect.

3.3 Model Weaving

Model Weaving often refers to linking two or more models by way of a weaving model (AMW Project Team 2008a) or link model (Kolovos et al. 2008). A weaving model contains a set of weaving links that link two or more model elements.

While model weaving can be applied for many model management tasks, it can in particular be used for expressing different kinds of model compositions. We look at two applications of model weaving for composition. In (AMW Project Team 2008b), a weaving model links elements of models that should be combined during composition. In (AMW Project Team 2008c), elements of metamodels are linked from which concrete compositions of models (instances of the linked metamodels) are derived. Note that there might well be other applications of model weaving where composition languages can be found and classified.

Components are models; that is, instances of a metamodel.

Component Denotation Components are denoted by IDs or path expressions that identify single elements in the models (AMW Project Team 2008b) or metamodels respectively (AMW Project Team 2008c).

Mapping is done implicitly: in (AMW Project Team 2008b) it is resolving an ID or expression to a model element. In (AMW Project Team 2008c) it uses the instance-of relationship.

Component Selection is selecting models or model elements to participate in the composition. In (AMW Project Team 2008b), single model elements are selected during the definition of a weaving model. In (AMW Project Team 2008c), metamodel elements are selected during the definition of a weaving model.

Component Connection In (AMW Project Team 2008b), component connections are explicitly defined through the definition of weaving links in the weaving model between the prior selected model elements. In (AMW Project Team 2008c), the component connection is defined on the meta-level.

3.4 Invasive Software Composition

Invasive Software Composition (ISC) (Aßmann 2003; Henriksson et al. 2008) is a language-independent software composition formalism. It defines a basic component model and basic composition operators independent of concrete languages and composition systems. On top of this, composition systems (including composition languages) can be built for arbitrary languages. In our tool Reuseware (Reuseware Project Team; Heidenreich et al. 2008a), we implemented the concepts of ISC for grammar-based and metamodel-based languages and provided a development environment

for composition systems based on ISC. Here we classify this implementation.

A fundamental principle of ISC is that each component—a fragment—has an explicit composition interface through which it can be addressed for composition. In Reuseware, fragments are either models or programs (i.e., instances of context-free grammars). The composition interface exposes selected model or program elements to be accessed during composition. Such elements are distinguished into reference points (can be accessed or extracted) and variation points (can be replaced). Concrete compositions are defined by composition links, linking reference and variation points in composition programs. Composition programs can be executed by statically replacing variation with reference points.

A composition system for a concrete language in Reuseware is created by defining how composition interfaces can be defined for or derived from models or programs written in that language. This definition contains rules that classify elements in models as reference or variation points. Furthermore, Reuseware allows the injection of constructs from Reuseware's generic composition language into a language for which a composition system is defined. This enables definition of composition programs inside of fragments.

Components A model is a component in Reuseware, if a composition system has been defined for the model's metamodel. A program is a component in Reuseware, if a composition system has been defined for the context-free grammar of the corresponding programming language.

Component Denotation Components are denoted by their name and by composition interfaces consisting of reference and variation points.

Mapping is implicit for a concrete composition system. It is, however, meta-explicit since it is defined and can be modified for each composition system separately in Reuseware.

Component Selection happens explicitly by selecting models or programs. It can be done in an external composition program, but also in-context, if the defined composition system supports it.

Component Connection is explicitly defined in form of composition links. This can be done in an external composition program, but also in-context, if the defined composition system supports it.

3.5 Template-Based Code Generation

Templates are documents that contain template parameters instead of concrete data at certain positions. Usually, templates are just regarded as plain text and do not have to conform to any specific language. Therefore, templates can be defined for any kind of digital document (Java classes, HTML pages, configuration files, etc.) and processed by a template engine like JET, StringTemplate, JSP, Velocity or MOFScript. Those engines take templates and data as input and compose them into complete documents. Since all mentioned engines work in a similar fashion, the classification below applies to all of them.

Components are: 1) Structured data (e.g., XML files, Java objects, models) and 2) templates. The space of available components is typically determined in a surrounding program by selecting a set of templates and associating data with named parameters in the template engine.

Component Denotation exists in two forms: templates are denoted through a naming scheme specific to the template engine. Data is denoted through template parameters, which are names sometimes associated with type information.

Mapping happens in two steps: 1) Components are explicitly associated with names by a) providing named templates and b) associating structured data with named parameters in the configuration of a template engine. 2) The template has an implicit internal mechanism for deriving denotations from these names and the components themselves. For example, some template engines will make available all elements of a data structure that are accessible through operations named `getXXX()`. However, these components will be made available under a name that excludes the `get`.

Component Selection is performed explicitly in the context of the templates.

Component Connection is done explicitly in-context at the same position where the selection takes place. Selection and connection are coupled.

3.6 Feature-Oriented Programming

In Feature-Oriented Programming (FOP) a *feature* is seen as an increment in program development and functionality. It can be used in Software Product Line Engineering (SPLE) (Pohl et al. 2005) to define and synthesise programs based on a unique composition of features. One system to define and execute such compositions is AHEAD[2] (Batory et al. 2004), where each feature is a nested tuple of unary functions (also called *deltas*). An example of such functions are Jak files, which add a certain increment in functionality to an existing Java class that shares the same name:

```
feature EnergySaving;

refines class MusicPlayer {
  autoSuspend() {...}
}
```

Components are features contained in files. The features are expressed in a language that can be composed by the AHEAD tool suite. For example, Jak components are

[2] Algebraic Hierarchical Equations for Application Design (AHEAD)

Java classes extended with domain-specific notions for defining refinements.

Component Denotation Components are denoted by the names of the files they are contained in.

Mapping Within AHEAD, features are mapped implicitly to their concrete realisation by using their names.

Component Selection happens explicitly by referencing specific features using their names in an AHEAD composition program.

Component Connection is usually done in-context of the feature module by using the denotation of the base component that is subject for refinement.

3.7 Feature-Driven Product Derivation

Variability modelling is used to express common and variable parts within Software Product Line Engineering (SPLE) and to explicitly define constraints between variable parts—features. Variability modelling abstracts from concrete feature realisation through feature models which is a powerful notion to handle the increased complexity in SPLE (Czarnecki 2005; Kang et al. 1990).

However, to build concrete products from a product line, features have to be realised using software artefacts shared across the product line. While variability modelling resides in the problem space, the realisation of features is part of the solution space. To instantiate products from a product line, feature realisations in the solution space must be included according to the presence of the features in a variant model; that is, a concrete selection of features from a feature model.

To support this transition from problem space to solution space in an automated way, a mapping from features to software artefacts that realise the features is needed. As an example, our tool FeatureMapper (Heidenreich et al. 2008b) allows for both defining and interpreting such mappings in a non-invasive way, that is, without changing the software artefacts.

Components are artefacts in models, that is, elements from models.

Component Denotation Components are denoted using names in the feature models. An important property is the ability to define constraints between components in the feature model.

Mapping Within the FeatureMapper, denotations—features from feature models—are explicitly mapped to concrete realisation components by the use of a mapping model.

Component Selection happens by selecting specific features from a feature model to build a concrete variant of the product line.

Component Connection In its default instantiation, our mapping framework works on components that are connected in solution models. During interpretation of the mapping model, solution artefacts are preserved and removed from the solution models depending on presence or absence of the corresponding features in a variant model. That is, we use a special form of in-context connection where every component is referenced directly.

3.8 Using IfDef Statements with the C Preprocessor

The C preprocessor (CPP) allows the definition and evaluation of `#define` constants. These can be used for many purposes in writing programs and managing their configurations. One very typical use is exemplified in the code snippet below:

```
#define USE_ENERGY_SAVING 1
...
#ifdef USE_ENERGY_SAVING
  // Component that saves energy
  cout << "Switched off your music player?"
       << endl;
#else
  // Component that wastes energy
  cout << "Want to start another device?"
       << endl;
#endif
```

It can be seen that this use of `#define` is actually quite similar to feature-driven development as discussed above: Here also, components are connected in-context, but selected through their denotations which are given as the labels of `#define` constants. Hence, this use of `#define` creates a composition system in our terminology:

Components are pieces of source code surrounded by `#ifdef` ... `#else` or `#ifdef` ... `#endif`.

Component Denotation is done by using `#define` constant labels.

Mapping The mapping of these labels onto components is given explicitly through `#ifdef` ... `#else` ... `#endif` structures in the source code.

Component Selection happens explicitly by defining specific `#define` constants either in some piece of source code or as explicit command-line parameters to the preprocessor.

Component Connection is done completely in-context, given by the order in which components are arranged within source-code files.

4. Conclusions

In this paper, we have proposed new terminology for describing the structure of composition languages. We have shown how this vocabulary can be applied to a large and diverse collection of current composition systems. Such vocabulary is an important prerequisite for comparative studies of different composition systems and composition languages.

In applying our proposed vocabulary to a number of composition systems, we have found—beyond showing that it is sufficiently general to cover a broad range of composition systems—two important factors for distinguishing between composition systems:

1. *Richness of component denotations.* Component denotations used in different composition systems range from simple names that stand for specific components to complex structures providing additional information about a component and its interface (e.g., feature models or component interfaces in invasive software composition). The more complex a component denotation, the more precisely can composition programs be expressed. Less complex denotations, on the other hand, are much easier to use for describing compositions.
2. *Definition of the mapping between denotations and components.* We have found that different composition systems use different techniques for defining a mapping between denotations and actual components. There are three possibilities:
   (a) *Explicit mappings,* where users of the composition language explicitly relate denotations and components in composition programs (cf. Sects. 3.1, 3.5, 3.7, 3.8);
   (b) *Implicit mappings,* where the relation is implicitly defined in the composition system (cf. Sects. 3.2, 3.3, 3.5, 3.6); and
   (c) *Meta-explicit mappings,* where the relation between denotations and components is provided explicitly, but not individually for each composition program, but rather as a set of rules that can be applied to many different composition programs (cf. Sect. 3.4).

Based on these findings, it is now an interesting question to study how these design decisions influence the usability of composition languages. What are good criteria for selecting one or the other type of denotation or mapping strategy for a specific composition language or even project?

Acknowledgments

This research has been co-funded by the European Commission within the FP6 project MODELPLEX contract number 034081 and by the German Ministry of Education and Research (BMBF) within the project feasiPLe.

References

AMW Project Team. Atlas model weaver, 2008a. URL `http://eclipse.org/gmt/amw/`. Last accessed August 08, 2008.

AMW Project Team. Amw use case – aspect oriented modeling, 2008b. URL `http://www.eclipse.org/gmt/amw/usecases/AOM/`. Last accessed August 08, 2008.

AMW Project Team. Amw use case – merge of geographical data (gml) with election statistics into svg, 2008c. URL `http://www.eclipse.org/gmt/amw/usecases/mergeSVG/`. Last accessed August 08, 2008.

Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004. doi: 10.1109/TSE.2004.23.

CPP. The C preprocessor. URL `http://gcc.gnu.org/onlinedocs/cpp/`. Last accessed August 08, 2008.

Krzysztof Czarnecki. Overview of Generative Software Development. In *Proc. Int'l Workshop on Unconventional Programming Paradigms 2004 (UPP'04)*, volume 3566 of *LNCS*, pages 326–341, Le Mont Saint Michel, France, September 2005. Springer.

Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On language-independent model modularisation. *Transactions on Aspect-Oriented Development*, October 2008a. Special Issue on Aspects and MDE (to appear).

Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th Int'l Conf. on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA, May 2008b. ACM. doi: 10.1145/1370175.1370199.

Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Extending grammars and metamodels for reuse: the Reuseware approach. *IET Software*, 2(3): 165–184, 2008. doi: 10.1049/iet-sen:20070060.

K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, Jyväskylä, Finland, June 1997. Springer.

Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, and Fiona A. C. Polack. *Epsilon*. Department of Computer Science, University of York, 2008. URL: http://epsilonlabs.wiki.sourceforge.net/Book.

Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

Sun Microsystems. Enterprise JavaBeans Specification, Version 2.0. Final Release, August 2001.

Object Management Group. CORBA components. OMG Document, January 2008. URL `http://www.omg.org/docs/formal/08-01-08.pdf`.

Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005. ISBN 978-3-540-24372-4.

Reuseware Project Team. Reuseware composition framework webpage. URL `http://reuseware.org/`. Last accessed August 08, 2008.

Towards a Framework for Guiding Aspect-Oriented Software Maintenance Empirical Studies

Marcelo Moura
Sérgio Soares *
Fernando Castor Filho †
Mário Monteiro
University of Pernambuco
{mlmm,sergio,fernando.castor,mqm}@dsc.upe.br

Phil Greenwood
Alessandro Garcia
Lancaster University
{greenwop,garciaa}@comp.lancs.ac.uk

Elliackin Figueiredo
Diego Araújo
University of Pernambuco
{emnf,daa}@dsc.upe.br

Abstract

Aspect-Oriented (AO) software development aims to improve software maintenance through the encapsulation of crosscutting concerns. However, empirical studies assessing the maintainability of AO software are still limited. Even worse, they are rarely replicated by different groups of researchers, thereby hampering the progress of the field. One of the key reasons for this is the lack of support for designing AO software exemplars or benchmarks to be reused across the community. This paper presents a framework for AO software maintenance, which has the purpose of guiding: (i) the planning, evaluation and replication of empirical studies, and (ii) the development, adaptation and evaluation of benchmark applications. The framework defines criteria to be satisfied by representative AO applications and their releases. We assess the effectiveness of the framework by comparing two different designs of the same empirical study: one that leverages the framework and another that was designed by an expert in empirical assessments of AO techniques.

1. Introduction

Aspect-Oriented Software Development (AOSD) is increasingly being applied to a wide range of application domains, such as Web-based systems [20], middleware [3], and software product lines [11]. AOSD aims to simplify software maintenance through the modularization of otherwise crosscutting concerns. However, the systematic evaluation of the benefits and drawbacks of AOSD in terms of maintainability is an often neglected, albeit critical, task. As AOSD is a relatively young paradigm, the preparation and replication of maintainability studies is challenging and time-consuming. In fact, empirical assessments are scarce in the context of aspect-oriented (AO) software maintenance [11; 13].

A fundamental stumbling block is the difficulty of enacting one of the first steps of any empirical assessment: the systematic selection, design, or adaptation of representative applications. The AO community in particular does not have a well-accepted group of exemplars or benchmarks [9; 19] for AO software maintenance. More fundamentally, researchers and practitioners do not have methodological frameworks to guide the construction of such benchmarks. Benchmark applications [25] are the basic empirical mechanisms to advance research and technology transfer in software engineering fields. In fact, there are some examples of their success within the software maintenance community [19].

Notwithstanding, the design of benchmarks for AO software maintenance is difficult for many reasons. First, the number of application domains where AOSD can be applied is rapidly increasing. AO programming was initially used to improve the maintainability of classical widely-scoped concerns in distributed applications [17; 20], such as concurrency control and error handling. Later, it has been used for very different purposes, such as improving middleware adaptability [3] and enhancing design stability of software product lines [11]. Second, the mechanisms that can be classified under the umbrella of AOSD and their hybrid incarnations in emerging programming languages [23] are consistently growing. Finally, such maintainability studies require the investigation of many factors typical in software maintenance tasks, such as different types of changes.

In spite of all the aforementioned challenges, there is a pressing need for methodologically supporting the generation of empirical studies and the construction of benchmark applications, rather than relying on a few "universal cases", which probably do not encompass several possible characteristics essential to different stakeholders goals and domains. In this context, this paper presents a framework that supports the assessment of AOSD techniques in terms of maintainability. The framework is an idealized scheme with specific guidelines and criteria to be realized by benchmark applications for assessing maintainability attributes of AO techniques. The framework guides researchers and practitioners in selecting or adapting applications and their releases that best fit the specific experimental goals. It can also be used to support the design, replication, and evaluation of empirical studies. Sec-

* Partially supported by CNPq, grants 309234/2007-7 and 480489/2007-6.
† Partially supported by CNPq, grants 481147/2007-1 and 550895/2007-8.

tion 2 introduces the proposed framework. Its effectiveness is assessed in Section 3. In Section 4 we reflect upon the many potential framework applications and discuss related work. Finally, Section 5 presents some concluding remarks.

2. The framework

This section presents a framework for AO software maintenance studies, which supports the systematic: (i) elaboration, evaluation and replication of empirical studies, and (ii) selection, generation and adaptation of benchmark applications. Based on the framework guidelines, their stakeholders can determine the extent to which applications, and its maintenance scenarios, are effective to cover study goals and assess AO techniques. It defines guidelines that can be tailored or extended to: (i) fit the specific goals of a maintenance study, and (ii) stimulate the evaluation of AO composition mechanisms. These guidelines are based on our experience from conducting a family of AO software maintenance studies over the last few years [3; 11; 13; 12], as well as analyzing others studies [1; 10].

The framework is structured according to three major components: Process (Section 2.1), Product (Section 2.2), and Maintenance Scenarios (Section 2.3). The first one is related to the framework workflow and how it works, and the other two address complementary assessment issues relevant to AO software maintenance. A concrete example on the application of the framework components is presented in Section 3.

2.1 The process component

This sections details the process and workflow related to the framework usage. The framework can be analyzed as a process to transform inputs into outputs, which may also serve as feedback to improve the framework. First, we classify framework *stakeholders* into two categories: the *designer of empirical studies* and the *benchmark designer*. The first group is interested in conducting a maintainability study involving one or more AO techniques. In this case, the input is the set of experimental requirements and the output is the initial configuration of the experiment. In contrast, the second group includes people who change or add new artifacts to the benchmark application. This group needs to analyze applications and maintenance scenarios to determine if they are suitable to conduct a variety of studies. The output of this process is one or more applications and change scenarios that are appropriate to benchmark AO techniques. Figure 1 shows a schematic representation of the Process component with the framework stakeholders, inputs and outputs.

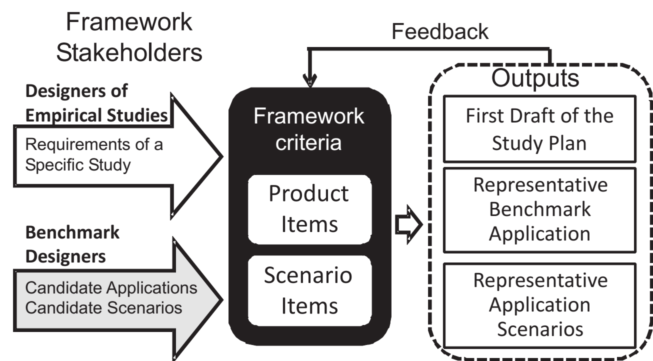

Figure 1. The inputs and outputs of framework process.

2.2 The product component

This section lists several characteristics that can guide and support the various decisions that have to be made, pertaining to the product (target systems), when designing maintainability studies and selecting candidates benchmark applications. We consider a wide range of possible characteristics that a candidate application might exhibit, such as its domain, the development techniques that were employed in its construction, and the types of the crosscutting concerns that appear in its implementation. It is important to stress that it is impossible to define a set of characteristics that applies to every possible empirical study or application domain. Therefore, the proposed list of application characteristics should be constantly evolved and extended to more closely meet the stakeholder's goals. Although, each stakeholder does not need to consider all criteria elements, they should focus on a representative subset that cover their experimental objectives. To ease this identification process and improve the organization of the different criteria, the Product component is divided into two groups: General Attributes, which encompass common application characteristics, and AO Attributes, which consist of specific characteristics of AO applications.

2.2.1 General attributes

A variety of general application attributes must be considered by framework users, such as *System Domain*, *Versions Available*, and *Packaging*. The domain of a system is often an important aspect to consider, so as ensuring that the system exhibits some expected properties (e.g. embedded systems are often of a resource constrained nature). Another important information, in order to evaluate the so mentioned and not so much demonstrated AO improvements, is the availability of OO versions of the same application, which is described in the *Versions Available* attributes. Related to the System Domain and the Version Available, the *Packaging* attribute specifies the technologies and programming languages used in an application, in addition to target platforms/operating systems. When considering an application for use in an empirical study it is important to consider the development artifacts (e.g. requirements specification, architecture documentation, and the design diagrams) available to ensure the goals of the study can be realized. The software *Life-Cycle Documentation* attribute lists all the documentation artifacts available for assessment. In addition to listing the documentation, the *Development Techniques* that have been used to create the system (e.g. design patterns, application toolkits, etc.) should also be described. Different techniques can influence

the quality of the final product and it is therefore important to take this into account when selecting an application.

2.2.2 Aspect-Oriented attributes

A variety of previous work has attempted to classify the different types of crosscutting concerns (CCCs) [1; 7]. The *Crosscutting Concern Classification* attribute provides classification of CCC types that are implemented with aspects according to different dimensions. For instance, the most popular classification of this nature has identified heterogeneous and homogeneous CCCs and also attempted to classify the invasiveness of the concern (i.e. if it alters the control flow of the system). For AO-related empirical studies it is important to have a range of different types of CCC implemented as aspects to guarantee that a variety of AO language constructs are exercised. Our CCC classification is based on a representative subset of the aforementioned existing classifications.

The first dimension of our CCC classification categorizes the concern as being *Functional* or *Non-functional*. A functional concern relates to business functionality, whereas a non-functional concern relates to the quality of the services provided by the system (e.g. security, reliability, distribution, etc.). Both functional and non-functional concerns can further be classified as either being *Homogeneous* or *Heterogeneous*. A homogeneous concern extends a program at multiple joinpoints by adding the same code at each joinpoint. A heterogeneous concern again extends multiple joinpoints but with a unique piece of code at each joinpoint. The final dimension of the CCC classification identifies if a concern affects a single component or multiple components, by either being an *Intra-Component* or *Inter-Component* concern.

Concerns in a system may be composed in a variety of ways. These compositions cause interactions between concerns which may affect system maintainability. Therefore, it is important to understand and classify these different types of composition. The *Concern Compositions* attribute allows the concern composition to be defined in some ways [3]. The simplest form of composition is *Invocation-Based Composition*. This arises if two concerns, C1 and C2, have no classes or aspects in common, i.e., they only communicate via method calls. *Component-Level Interlacing* occurs if multiple concerns have one or more components (classes or aspects) in common but no operation (i.e. method, advice, etc.) in common. As a result, the concerns are interlaced and tangled at the component level only. In contrast, *Operation-Level Interlacing* occurs if multiple concerns have one or more operations in common. In this case, concerns are interlaced at the operation level. Finally, *Overlapping* identifies points where multiple concerns share one or more statements, operations or components. In contrast to interlacing interactions that have disjoint common parts, overlapping concerns share elements (i.e. an entire component operation or statement contributes to multiple concerns).

A previous study reported that the maintainability of the code pertaining to a concern is directly related to the language constructs employed in its implementation [13]. Therefore, it is important to identify the various language features used to implement a particular concern. The *AO Language Constructs* attribute identifies the language elements used to implement a crosscutting concern, such as intertype declarations, pointcuts (the various different kinds of), advices, etc. The rationale in this case is that an application that leverages a large number of language constructs may be a good benchmark application. Language constructs can also be classified into other dimensions, such as, static (e.g. intertype declarations) or dynamic (e.g. cflow pointcut designator), according to the time when they are handled by the compiler (compile-time, load-time, run-time). This information could help the designers to evaluate in another way the performing of constructs at applications.

2.3 The maintenance scenarios component

A framework for empirical studies in software maintenance must also include a catalog of representative real software changes scenarios. This is necessary to ensure the framework can evaluate a variety of recurring maintenance issues. These scenarios are assessed through the attributes presented in the rest of this section, which aims to support decisions of both benchmark designers and empirical studies designers.

The *Scenario Description* attribute provides the name, description, and other general information about the scenario. Another provided information is the *Versions Available* attribute that reports if there are OO versions of each scenario. It is also necessary to specify the *Change Type*, which is partially based on previous works [2; 4; 24]. This involves classifying the change according to the effect it has on the base application. In this work, we consider that changes can be of one amongst three types: corrective, adaptive, or perfective [21]. This attribute also requires the *Goal of the Change* to be specified, which describes the desired effect the change has on the system.

The *Nature of the Change* must also be listed and it encompasses two types: structural and behavioral changes. These change types are orthogonal; therefore, a change scenario can affect both the software structure and its behavior at the same time. When *Structural Changes* are made, these can involve: *additions* (e.g. introduction of a new method), *subtractions* (deletion of an attribute), or *alterations* (to modify an existing element). In contrast, *Behavioral Changes* are changes that modify the software behavior and can be classified as either: (i) *behavior-modifying*, changes that alter the software behavior; or (ii) *behavior-preserving*, changes that preserve the behavior of the software (refactorings).

Finally, it is necessary to actually document the changes that are made. This involves specifying the *Changes at the Requirements Level*, *Changes at the Analysis and Design Level*, and *Changes to the Implementation* attributes. The changes performed in the requirements can influence artifacts at later development stages. These effects can differ depending on the applied requirements engineering approaches [21]. Almost all changes made to the analysis and design artifacts are critical due to these artifacts being the core of software de-

velopment activities. Analysis and design changes are likely to affect the rest of the application and can, consequently, impact software modularity. Finally, changes to implementation artifacts often have less broad-scoped effects.

3. Framework evaluation

This section presents an evaluation of the framework involving a real application, the HealthWatcher system [20]. The goal is to assess the efficacy of the proposed framework when planning a new empirical study in AO software maintainability. Our analysis aimed at examining (i) whether circumstances the framework can assist in identifying omissions and inaccuracies in the planning or execution of empirical studies, and (ii) whether characteristics of the target applications adhere to the framework attributes.

We have selected the aforementioned application for several reasons. First and foremost, it needs to be examined for its exploitation in different empirical contexts. HealthWatcher was being considered by another research group as a target candidate in the planning of an upcoming controlled experiment. The goal of the new experiment was to assess the maintainability of AO Web-based systems. The framework was used in this study in order to guide the elaboration of the experimental design, one of the framework outputs (Figure 1). Second, the application has a number of unique characteristics that make it an appropriate representative of evolving AO software systems. It was developed using different programming languages, such as Java and AspectJ [17]. There are already some recent studies in the research community using it (e.g. [6; 13; 20]). In fact, the system was specially adapted to work as benchmarks for comparing the maintainability of AO and OO techniques [20]. Furthermore, there are multiple releases of HealthWatcher available and each release is the result of applying a number of heterogeneous changes to the previous one. Third, HealthWatcher is a Web-based information system based on a classical n-tier architecture. It is a real system developed by a team of developers in Recife, Brazil [20]. Finally, the system and its releases were originally designed without any access to the framework proposed here.

3.1 Planing a new experiment

As previously mentioned, this initial evaluation of the framework was carried out in the context of planning a new experiment. Considering one of the previous studies that generated several releases and scenarios of HealthWatcher, which focused on analyzing the stability of AO software design in the presence of heterogeneous types of changes [13] *(Exp. A)*. Designers of a new controlled experiment to evaluate the maintainability of AO Web-based systems *(Exp. B)*, performed an informal analysis to verify if the application release and the maintenance scenarios generated in *Exp. A* could be reused in the context of their new experiment. Thus, the framework was used to support the same analysis that occurred in *Exp. B*, guiding experiment designers to analyze and configure, if necessary, HealthWatcher and its scenarios in the light of the key experimental goals. We analyzed the efficacy of the framework by comparing two different drafts of the experimental plan that were devised independently, as described below.

Experimental Settings: Expert vs. Framework User. The first experiment plan was designed using the framework. The other one was designed by an expert that did not rely on the framework. The goal of this evaluation was twofold. First, we expected to be able to systematically verify how the framework can assist the planning of studies focusing on AO software maintenance. Moreover, by comparing the results of using the framework with the study design produced by an expert, we also expected to identify benefits and limitations of the framework. Both drafts of the experiment plan were generated based on the same set of goals. The design of the first plan assisted by the framework was carried out by a postgraduate student who was not expert in architectural analysis but had basic knowledge on experimental design. The other (without using the framework) was prepared by a researcher who was an expert in architectural analysis, had developed a number of empirical studies of AOSD in the last five years, but had no familiarity with the framework.

3.2 Evaluation results

Based on the specific experimental goals, we have examined if the framework criteria were effective in the analysis and adaptation of HealthWatcher. We have centered the assessment on the product and maintenance scenario components. After the execution of this evaluation, we have identified if there was any product element or change scenario that was not present in the framework. This provided a realization of the feedback loop illustrated in Figure 1. The results provided first evidence that the use of the framework might be appealing even for experts on empirical assessment of AOSD. As described in the following, the first draft, generated by the postgraduate student, encompassed some relevant considerations that were not addressed by the expert.

Scenarios for Observing AO Architecture Instabilities. While analyzing the appropriateness of HealthWatcher, the postgraduate student has considered possible architectural instabilities when different characteristics of AO software are present. In the expert plan, the only product characteristics taken into consideration were the diversity of functional and non-functional concerns. This was very limited when compared to the ones defined in Section 2.2. Table 1 shows all occurrences of the items described in Section 2.3.2. Notice that there is no functional crosscutting concern. This gap is a useful information for decision making in the experimental design. If no additional change scenario is included in the application, there is no guarantee that the experiment results will be representative to Web-based applications that do encompass such a characteristic (for example, to make the implementation of business rules more flexible).

Unified Classification of Maintenance Scenarios. Another point in our analysis of the two experiment plans was related

Functional	Non-Functional	Homogeneous	Heterogeneous	Intra-Component	Inter-Component	Invocation-based composition	Component-level interlacing	Operation-level interlacing	Overlapping
-	X	X	X	X	X	X	X	X	X

Table 1. Types of CCC and their Interactions.

to the classification criteria for the maintenance scenarios. We have observed that the expert (Case 2 in Table 2) categorized the changes using his own terminology, differently from the postgraduate student (Case 1), which used the framework categories. The expert included the use of terms for maintenance types that are not widely accepted yet by well-known classifications of software changes [21; 24]. In fact, there is no consensus on the categorization of software maintenance types [21] that are used in practice. This imprecision makes the existence of a framework fundamental to avoid misunderstandings when analyzing and replicating studies on AO software maintenance. Such ambiguity problems can be ameliorated through the use of the framework. It guides different experiment designers to analyze characteristics of their experiments in a coherent way. On the other hand, the draft generated by the expert covered some essential points not covered by the postgraduate student. The expert was more effective to anticipate specific architectural stability metrics to be applied to the change scenarios.

Release	Scenario Description	Case 1	Case 2
R1	Factor out multiple Servlets to improve extensibility.	P	R
R2	Ensure the complaint state cannot be updated once closed.	P	C
R3	Encapsulate update operations to improve maintainability.	P	R
R4	Improve the encapsulation of the distribution concern.	P	R
R5	Generalize the persistence	P	R
R6	Remove dependencies on Servlet response and request.	P	R
R7	Generalize distribution mechanism.	P	R
R8	New functionality added to support querying of more data types.	P	E
R9	Modularize exception handling and include more effective error recovery behavior into handlers.	P	P

Legend: P = Perfective; R = Refactoring; C = Corrective; E = Evolutionary.

Table 2. HealthWatcher maintenance scenarios [13].

4. Discussion and related work

Based on the previously presented results, this section discusses the benefits and limitations of our framework and contrasts it with related work.

On the Symbiosis between Framework and Designers Experience. The framework is not a replacement for the creativity and experience of empirical studies and benchmarks designers. On the contrary, experts can be more effective to include certain change types (e.g. refactorings) that are not explicitly cataloged in popular classifications of software changes. Also, the strict use of the framework might constrain creativity in experimental designs if not applied properly. A good practice might be using the framework only after delivering a first plan draft of the empirical study.

Framework Coverage. It is virtually impossible to come up with a framework that covers all the relevant issues of benchmarking applications for AO software maintenance. Different maintainability studies of AOSD have specific goals that impose particular demands on the used applications. However, the idea is that the framework can assist on the time-effective generation of a first sketch on the experiment plan. As we have noticed in the HealthWatcher case, there are certain relevant benchmarking items that are easily skipped without using the framework. Besides, the framework provides a coherent and unified way of documenting important characteristics necessary to ease replication of empirical studies. The framework criteria can also be extended.

Absence of Benchmarking Frameworks for AOSD. A variety of related research has influenced the development of our framework as documented throughout the previous sections. However, there are very few instances which attempt to consolidate a methodology for designing benchmarks for AO software maintenance. Existing benchmarking frameworks in software engineering have been limited to provide decision support tools for creating such processes [5; 8]. Alternatively, the work presented by Demeyer [9] describes a software evolution benchmark that has very similar goals to ours. However, his framework is very generic and does not include important collaboration attributes.

In contrast, the work by Kienzle [18] derives a set of attributes for assessing language expressiveness based on aspects for ACID properties. The authors claim that these aspects and attributes naturally create a benchmark for assessing AO languages. However, a problem of this benchmark is that it is focused on a particular domain (transactional properties), which limits its applicability. Our proposed approach for benchmarking not only takes maintenance activities into account but is applicable to a variety of domains and does not focus on one particular assessment attribute. This categorization of aspects used by our framework and Kienzle is reminiscent of Aspect Categories [16], which attempts to classify aspects according to the affect they have on the base system. Some of these categories overlap with the attributes specified in our framework (e.g. concern compositions). However, the work by Katz is focused on proving correctness, rather than supporting assessment.

Improving the Body of Knowledge on AO Software Maintenance. Shull et al [22] highlight the need for information sharing and replication in empirical studies. The benchmark

framework described in this paper contributes significantly towards these goals. Furthermore, from the work described in this section it is clear there is a need to document a set of guidelines for creating empirical studies of AOSD. Although a number of such studies have been performed, they are often conducted in isolation using different practices. By providing a framework, studies will promote easier information exchange and it will encourage more AOSD practitioners to conduct and replicate maintainability studies.

Interplay of Testbeds and Frameworks for AOSD. One of the most closely related initiatives to our framework is [14]. This initiative consists of developing an AOSD testbed with a long-term aim of providing a series of benchmark applications (and supporting artifacts). The testbed repository also contains a set of initial metrics and collected results with intention of other software developers reusing these resources to test their approaches. Our framework directly complements this initiative by i) providing a decision support framework for selecting and creating artifacts to instantiate appropriate empirical studies and ii) promote information sharing to allow the replication of studies.

5. Conclusions and Ongoing Work

The proposed framework constitutes a significant stepping stone towards the creation of a compressible methodology for designing AO software benchmarks. Successful technology transfer usually takes a long time to be implemented, often around 18 years [15]. Supported by our first evaluation outcomes, we believe that our framework is an effective contribution to decrease this delay in many ways. First, it has shown to provide systematic ways for facilitating the effective design of maintainability studies on the AOSD arena. Moreover, the framework can also guide the selection, design, or adaptation of representative evolving applications and releases to be used in such studies and their replications. Third, this will help the community to accelerate the improvement of our body of knowledge on AOSD and better support the judgment of industrial decision makers. Finally, studies extensions and replications are time-consuming; in order to ameliorate this problem, we believe that our framework helps to save time on the quality assessment of existing empirical studies. However, the framework is not intended to guide all empirical decision that have to be done when planning, replicating, or evaluating studies. For example, the empirical study designers must evaluate if the relation of the selected set of scenarios with each other can bias the results. As ongoing work, we are planning to conduct a number of applications of the framework.

References

[1] Apel, S., et al. On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?. AOPLE at GPCE.06, 2006.

[2] Buckley, J., et al. Towards a Taxonomy of Software Change. Journal on Software Maintenance and Evolution: Research and Practice, p. 309-332, 2005.

[3] Cacho, N. et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. AOSD.06, 2006.

[4] Chapin, N., et al. Types of software evolution and software maintenance. Journal on Software Maintenance and Evolution: Research and Practice, 13:330, 2001.

[5] Chiew, V., et al. Software Engineering Process Benchmarking. PROFES.02, p. 519-531. LNCS, v. 2559, 2002.

[6] Chitchyan, R., et al. Early Aspects at ICSE 2007. ICSE.07 Companion, p.127-128. ACM Press, May 2007.

[7] Constantinides C., et al. Reasoning About a Classification of Crosscutting Concerns in Object-Oriented Systems. Workshop on AOSD. Bonn, February 2002.

[8] Daoudi, F., et al. A Benchmarking Framework for Methods to Design Flexible Business Processes. Software Process: Improvement and Practice, 12(1):51-63, 2006.

[9] Demeyer, S., et al. Towards a Software Evolution Benchmark. Workshop on Principles of Software Evolution, 2001.

[10] Eaddy, M., et al. Do Crosscutting Concerns Cause Defects? IEEE Transaction on Software Engineering, 2008 (to appear).

[11] Figueiredo, E., et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. ICSE.08, p. 261-270, May 2008.

[12] Garcia, A., et al. Modularizing Design Patterns with Aspects: A Quantitative Study. AOSD.05, March, 2005.

[13] Greenwood, P., et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. ECOOP.07.

[14] Greenwood, P., et al. On the Contributions of an End-to-End AOSD Testbed. Early Aspects at ICSE.07, May 2007.

[15] Jedlitschka A., et al. Relevant Information Sources for Successful Technology Transfer: A Survey Using Inspections as an Example. ESEM 2007, p. 31-40. IEEE, September 2007.

[16] Katz, S. Aspect categories and classes of temporal properties. TAOSD, 3880, Springer, 2006.

[17] Kiczales, G., et al. Getting Started with AspectJ. Communications of the ACM 44(10), 5965 (2001).

[18] Kienzle, J., et al. AO challenge - implementing the ACID properties for transactional objects. AOSD.06, March 2006).

[19] Sim, S., et al. Using benchmarking to advance research: a challenge to software engineering. ICSE.03, IEEE, 2003.

[20] Soares, S., et al. Implementing Distribution and Persistence Aspects with AspectJ. OOPSLA.02, p. 174-190, 2002.

[21] Sommerville, I. Software Engineering. Addison-Wesley, 2006.

[22] Shull, F., et al. Knowledge-Sharing Issues in Experimental Software Engineering. Empirical Software Engineering, 9:111-137, 2004.

[23] Sullivan, K. et al. Information Hiding Interfaces for Aspect-Oriented Design. FSE-13, Sept 2005, Lisbon, Portugal.

[24] Swanson, E. The Dimensions of Software Maintenance. ICSE.76, p. 492-297. IEEE, 1976.

[25] Tichy, W. Should Computer Scientists Experiment More? IEEE Computer, 31(5):3240, May 1998.