Journal Subline

LNCS 3880

Transactions on Aspect-Oriented Software Development I

Awais Rashid · Mehmet Aksit
Editors-in-Chief

Springer

Awais Rashid Mehmet Aksit (Eds.)

Transactions on Aspect-Oriented Software Development I

Springer

Volume Editors

Awais Rashid
Lancaster University
Computing Department
Lancaster, LA1 4YR, UK
E-mail: awais@comp.lancs.ac.uk

Mehmet Aksit
University of Twente
Department of Computer Science
Enschede, The Netherlands
E-mail: aksit@ewi.utwente.nl

Library of Congress Control Number: 2006921902

CR Subject Classification (1998): D.2, D.3, I.6, H.4, K.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-32972-2 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-32972-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11687061 06/3142 5 4 3 2 1 0

Editorial

Welcome to the first volume of *Transactions on Aspect-Oriented Software Development.* Aspect-oriented methods, tools and techniques are gaining in popularity due to their systematic support for modularizing broadly scoped properties, the so-called *crosscutting concerns*, in software systems. Such crosscutting concerns include security, distribution, persistence, mobility, real-time constraints and so on. As software systems become increasingly ubiquitous, mobile and distributed, the modular treatment of such crosscutting concerns also becomes critical to ensure that software artifacts pertaining to such concerns are reusable, evolvable and maintainable. This modular treatment of crosscutting concerns by aspect-oriented techniques is not limited to code level. In fact, aspect-oriented techniques cover the software life cycle, handling crosscutting concerns in requirements, architecture, design, code, test cases, system documentation, etc.

The aspect-oriented software development community is growing fast, with an increasing number of researchers and practitioners across the world contributing to the development and evolution of the field. The community launched its own conference in 2002, which has since been held with great success on an annual basis. Recent reports from Burton and Gartner groups have put aspect-orientation on the *plateau of productivity* on the evolution cycle of new technologies. One of the key indicators of the maturity of a field is the availability of high quality research of an archival nature. The launch of *Transactions on Aspect-Oriented Software Development*, therefore, signifies a key milestone for the maturity of work in this area. The journal is committed to publishing work of the highest standard on all facets of aspect-oriented software development techniques in the context of all phases of the software life cycle, from requirements and design to implementation, maintenance and evolution. The call for papers is open indefinitely and potential authors can submit papers at any time to: taosd-submission@comp.lancs.ac.uk. Detailed submission instructions are available at: http://www.springer.com/sgw/cda/frontpage/0,,3-164-2-109318-0,00.html. A number of special issues on current important topics in the community are already in preparation. These include special issues on AOP systems, software and middleware; AOP and software evolution; dynamic AOP, and Early Aspects. Calls for such special issues are publicized on relevant Internet mailing lists, Web sites as well as conferences such as the Aspect-Oriented Software Development conference.

The articles in this volume cover a wide range of topics from software design to implementation of aspect-oriented languages. The first four articles address various issues of aspect-oriented modeling at the design level. The first article, "Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value", by Lopes and Bajracharya, proposes a methodology and a tool to show how aspects can be beneficial as well as detrimental to a certain design. The second article, "Modularizing Design Patterns with Aspects: A Quantitative Study", by Garcia et al., analyzes and compares the aspect-oriented and object-oriented implementations of design patterns with respect to quality values such as coupling and cohesion. The article "Directives for Composing Aspect-Oriented Class Models", by Reddy et al., proposes models for expressing aspect-oriented and non–aspect-oriented properties of systems and defines techniques to compose these models together. In the article "Aspect Categories and Classes of Temporal Properties", Shmuel Katz defines a method for classifying aspects with respect to their temporal properties so that application of aspects in a system can be better understood and analyzed.

The following four articles discuss various programming language issues. The article "An Overview of CaesarJ", by Aracic et al., gives an overview of the CaesarJ programming language, which aims at integrating aspects, classes and packages so that large-scale aspect components can be built. In the article "An Expressive Aspect Language for System Applications with Arachne", Douence et al. motivate the applicability of the Arachne language in improving systems written in the C language, where system dynamicity and performance play an important role. Monteiro and Fernandes define in their article, "Towards a Catalogue of Refactorings and Code Smells for AspectJ", a catalogue that helps in detecting aspects in object-oriented programs and in improving the structure of extracted aspects within the context of the AspectJ language. The final paper in the language category is "Design and Implementation of An Aspect Instantiation Mechanism" by Sakurai et al. It proposes association aspects as an extension to AspectJ for flexible descriptions of aspects whose instances are associated with more than one object.

The final article in this volume, "abc: An Extensible AspectJ Compiler", by Avgustinov et al., describes a workbench for implementing aspect-oriented languages, so that easy experimentation with new language features and implementation techniques are possible.

The inception and launch of *Transactions on Aspect-Oriented Software Development* and publication of its first volume would not have been possible without the guidance, commitment and input of the editorial board and the reviewers who volunteered time from their busy schedules to help realize this publication. We thank them greatly for their help and efforts. Most important, we wish to thank authors who have submitted papers to the journal so far. The journal belongs to the community and it is the submissions from the community that are at the heart of this first volume and future volumes of *Transactions on Aspect-Oriented Software Development.*

Awais Rashid and Mehmet Aksit
Coeditors-in-chief

Organization

Editorial Board

List of Reviewers

John Grundy	Wim Vanderperren
Charles Haley	Kris de Volder
Stephan Hannenberg	Robert Walker
Jan Hannemann	Nathan Weston
Wilke Havinga	Jianjun Zhao

¥410.00元

Table of Contents

Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value[*]

Cristina Videira Lopes and Sushil Krishna Bajracharya

Department of Informatics,
Donald Bren School of Information and Computer Sciences,
University of California, Irvine
{lopes, sbajrach}@ics.uci.edu

Abstract. The design structure matrix (DSM) methodology and the net option value (NOV) model have been used before to show how aspects can add value to a design. Following an in-depth analysis of that study, this paper demonstrates how aspects can be beneficial as well as detrimental. The structural transformations involved in aspect modularizations are carefully analyzed in the context of DSMs. This analysis exposes the unique *reversion* effect on dependencies that aspect modules are known for. To capture that effect within the NOV model, we extend its original set of six modular operators with an additional *reversion* operator. Using a design case study, its NOV worksheet and NOV experiments' curves are presented to show a simulation of the evolutionary patterns of modules, including aspect modules. These patterns show how subtle dependencies, or the lack of them, bring down, or up, the value of an existing design. Based on the observations made in this case study, preliminary design guidelines for aspects are formulated.

Keywords: Aspect-oriented programming and design, modularity, design space matrix, net option value.

1 Introduction

Software design is a complicated process that tries to balance several factors, some of them contradictory. Bad design decisions can have disastrous consequences. Therefore, whenever new design concepts are proposed, they must be carefully assessed, so that their scopes of appropriate applicability can be identified. Such is the case with aspect-oriented design. To do that, one needs to use appropriate assessment methods. Conventional techniques for evaluating software design are based on metrics, quality attributes and heuristics [14, 17, 35]. While they can be useful for a posteriori analyses, they are not thought of for assessing the design options at certain decision points. But in the case of aspects, one needs to be able to assess when an aspect modularization is more beneficial than its nonaspectual alternatives.

[*] This work has been supported in part by the National Science Foundation's grant no. CCF-0347902.

This paper presents a case study where several object-oriented and aspect-oriented design variants for a software application are compared and analyzed in depth using a new methodology. This methodology uses the design structure matrix (DSM) as a design representation and net option value (NOV) as an analytical model. The paper explores this new methodology for assessing design options, and at the same time, it demonstrates how aspect-oriented modularization can cause beneficial as well as detrimental effects in an existing object-oriented design.

DSM, also known as design space matrix or dependency structure matrix, is an analysis and design tool used in various engineering disciplines [1,15,38,40]. In its simplest form, a DSM is an adjacency matrix representation of the dependencies between design elements. The idea of using DSMs to model complex systems was first introduced by Steward [40]. DSMs are widely used in system design, independent of NOV. Various analysis techniques, metrics and tools have been developed that are based on DSMs. MacCormack et al. present an empirical study that compares the structure of large-scale complex software (Linux and Mozilla) using DSM-based metrics [28]. A commercial tool for analyzing software architecture based on DSMs has been developed by Lattix [2]. These related works demonstrate the applicability of DSMs in analyzing large software systems.

NOV is a model for evaluating modular design structures based on the economic theory of real options. Baldwin and Clark formulated NOV and first demonstrated its usage in analyzing design options [13] in the computer hardware industry. There are two fundamental components in Baldwin and Clark's work: (a) a general theory of modularity in design with six modular operators as sources of design variation;[1] and (b) NOV as a mathematical model to quantify the value of a modular design: the mathematical expressions for NOV tie together modular dependencies, uncertainty and economic theory in a cohesive model.

Sullivan et al. first demonstrated how the methodology using DSMs and NOV can be used in the analysis of software design [42]. Their work extends the DSM structure by introducing *environment parameters*, and applies this extended model to the design of KWIC (Keywords in Context), the program originally presented by Parnas [31]. Using NOV analysis they showed how information-hiding design is superior to the protomodular one. Information hiding is achieved by defining appropriate interfaces as *design rules*, which facilitate future changes in the design by reducing intermodular dependencies.

DSMs and NOV have also been used in analyzing aspect-oriented modularization [26]. This was the first work that looked into a new form of modular construct, *Aspect* [22], in addition to the conventional constructs for creating independent modules with representations for data structure, interface and algorithm.

In the context of the prior work mentioned above, namely [26,42], the contributions of this paper are as follows:

[1] The six modular operators are: (i) splitting, (ii) substitution, (iii) augmenting (augmentation), (iv) exclusion, (v) inversion, and (vi) port(ing).

- The paper provides examples that give insights on the correlation between module dependencies and the benefits/disadvantages of aspects, using a realistic case study.
- Based on the detailed analysis of the design evolution of the case study, preliminary guidelines for aspect-oriented design are presented.

In addition, the work presented in this paper is one of the most detailed applications of NOV to software design to this date (late 2005). It explores the applicability of NOV in evaluating software design options and exposes limitations that need to be further resolved. In the context of NOV, a new modular operator for aspects is defined that has been named *reversion.*

The paper is organized as follows. The software application used as the case study is described in Sect. 2. Further detail on DSMs as applied to this paper is given in Sect. 3. The process of exploring design variants for the case study is detailed in Sect. 4. This starts with studying a third-party application to identify the design parameters within it. These parameters are changed to obtain a design for a new application which is further modified to obtain rest of the variants, the last two being the results of aspect modularization (Sects. 4.3–4.7). Each of these design changes is described in terms of one or more of the six modular operators from the NOV model. A new modular operator called *reversion* is formulated in Sect. 5 based on the structural changes that aspects bring in and the effect they have on module dependencies. Section 6 summarizes the mathematical model of NOV and details all the assumptions made about the NOV parameters for the case study in this paper. Section 7 discusses the NOV analysis of the case study, and, based on several observations, it formulates preliminary aspect-oriented design guidelines. Section 8 describes the limitations of the analysis, the open issues in using NOV to evaluate software design and further work we intend to pursue. Section 9 concludes the paper.

2 Case Study

The case study used throughout the paper is a Web application that uses Web services to meet most of its functional requirements. The application, *WineryLocator*, uses Web services to locate wineries in California. This section describes what the application is about and how is it structured.

A user can give a point of interest in California as a combination of street address, city and zip code. The address need not be exactly accurate. Once this information is given, the user is either presented with a list of matching locations to his/her criteria or is forwarded to another page if the given address uniquely maps to a valid location in California. Once the application gets a valid starting point, the user then can select preferences for the wineries. Based on the preferences and the starting point, the application generates a route for a tour consisting of all the wineries that match the criteria. The result is a set of stops in the route and a navigable map. From the result the user can also get driving directions.

2.1 Functional Decomposition

With the functionality described above, the following types of services are needed:

Finding Accurate Locations (List). A service that takes an incomplete description of a location and returns exact/accurate locations that match the description.

Getting List of Wineries. A service that returns a list of all the wineries around the vicinity of the user's starting point. The user must be able to filter (her) his selection according to the different criteria (s)he wanted regarding the wineries to be visited.

Getting Wineries Tour. Once an accurate starting point is obtained, we need to get a set of wineries around that starting location. This further breaks down as:

- Getting all the winery stops and information that form a tour
- Getting a map for the tour that constitutes the wineries
- Navigating the map that highlights the tour with appropriate marks and supports basic operations like panning and zooming

Driving Directions. Given a route made up of locations, we need a set of driving directions to visit all the destinations in the tour.

We use an existing application for MapPoint Web services [8] called *Store-Locator*,[2] developed by SpatialPoint [9], as our starting design so that we can make changes in it to get *WineryLocator*. StoreLocator is similar in many ways to WineryLocator. Given a starting point of interest, StoreLocator displays several matching locations. Once the user picks the starting location, it generates a navigable map and a list of all coffee stores close to that starting location within a radius specified by the user. The user then can click on each store to get driving directions from the start location.

Hence, as far as the functionalities are concerned, only two changes need to be made in StoreLocator to get WineryLocator: (i) replace the coffee store search with winery search, and (ii) present the user with a tour including the start location and all the wineries, unlike a list of directions from the start location to a selected store in StoreLocator.

In order to locate points of interest, such as coffee stores or wineries, MapPoint allows their service users to either use an already available *datasource* or upload new geographic data as a custom datasource. To bring out more opportunities for design changes, we substitute this functionality from MapPoint by our own Web service *WineryFind*, which provides a list of wineries around a vicinity of an exact start location. WineryFind also allows the users to set their search criteria by giving different preferences related to wines and wineries.

Table 1 shows the mapping of core application functionalities to the available Web services. The implementation was done in Java, using Apache AXIS [7] as well as the SOAP [43] toolkit to access the Web services.

[2] Available online at http://demo.mappoint.net.

Table 1. Mapping tasks to services

Task	Services	Providers	Method Signatures *
Finding set of exact locations	FindService	MapPoint	`FindResults findAddress (FindAddressSpecification)`
Getting wineries matching criteria	WineryFind	Local service we developed	`Destination[] getLocationsByScore (WinerySearchOption)`
Generating route from the tour given set of destinations	RouteService	MapPoint	`Route calculateSimpleRoute (ArrayOfLatLong, String /*dataSourceName*/, SegmentPreference)`
Getting a map representing a route/tour. Also, navigating the map	RenderService	MapPoint	`ArrayOfMapImage getMap(MapSpecification)`
Getting driving directions	RouteService	MapPoint	can be obtained from a `Route` object

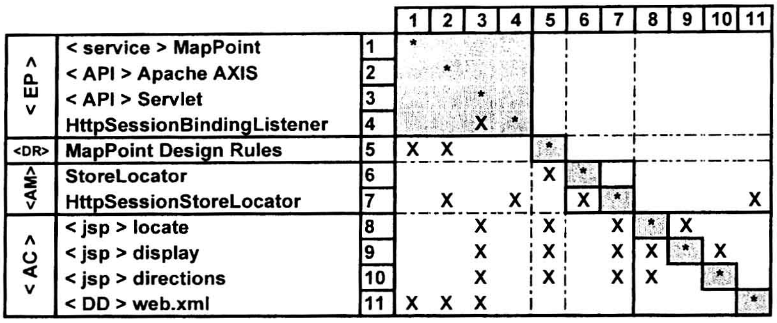
* Showing only the most relevant methods in format - `return_type Web_service_function_name (input_parameter_type)`. The types shown in the list represent the classes in Java that map to the types defined in the MapPoint object model. These classes were autogenerated by the tool WSDL2Java, which is a part of the Apache AXIS toolkit [7].

2.2 Subsidiary Functions

Besides the main functionalities that WineryLocator offers to its end users, we consider two subsidiary functions the application needs to provide. These subsidiary functions, which are not directly visible to the users, are as follows: (1) *Authentication:* Before using any of the MapPoint services the application needs to provide a valid credential (username and password) to it. This credential does not come from an end user, but is managed by the application service provider. MapPoint uses the HttpDigest authentication mechanism for this. (2) *Logging:* A logging feature is introduced in the system as a nonfunctional (subsidiary) requirement to trace all the calls made to the Web services. Such a feature is useful in many scenarios that require maintaining statistics about the access to the Web services within the application. This feature can simply be implemented by tracing every call to a Web service in the system.

3 Representing Design Structures with DSM

Figure 1 depicts the design of StoreLocator in a DSM. Before presenting the design evolution from StoreLocator to WineryLocator in DSMs, we first describe some fundamental design concepts presented by Baldwin and Clark in [13], focusing primarily on software.



			1	2	3	4	5	6	7	8	9	10	11
< EP >	< service > MapPoint	1	*										
	< API > Apache AXIS	2		*									
	< API > Servlet	3			*								
	HttpSessionBindingListener	4			X	*							
<DR>	MapPoint Design Rules	5	X	X			*						
<AM>	StoreLocator	6					X	*					
	HttpSessionStoreLocator	7		X		X		X	*				X
< AC >	< jsp > locate	8			X		X		X	*	X		
	< jsp > display	9			X		X		X	X	*	X	
	< jsp > directions	10			X		X		X	X		*	
	< DD > web.xml	11	X	X	X								*

Fig. 1. DSM for StoreLocator

3.1 Elements of Modular Design in Software

In this paper, interpretation of the terms like modularity, architecture and hierarchy remains the same and as generic as that originally presented by Baldwin and Clark [13]. Almost all of the constituents of design that make up their theory can be seen in the designs for StoreLocator and WineryLocator. We briefly summarize the definitions of the core elements from [13], as they are seen in the examples presented in this paper. All the definitions and vocabulary borrowed from [13] are shown in *italics* below.

1. *Design: Design* is defined as *an abstract description of the functionality and structure of an artifact.* Representations such as software architectures [33, 39] design models in UML or source code fit this definition.
2. *Hierarchies:* The notion of hierarchy concurs with the one defined by Parnas [32]. A module A is dependent on module B if A needs to know about B to achieve its function, i.e., if B is visible to A.
3. *Medium* for expressing design: A designer expresses the basic structure and configuration of design elements with a *medium* (s)he chooses to work with. Examples are Architecture Description Languages (ADLs) for software architecture [30], UML for object-oriented modeling and Java for program design (code). Media are among the highest parameters in the design hierarchy.
4. *Design parameters—the elements of design:* Parameters are the attributes of the artifact that govern the variation in design. Choosing new values for parameters gives new design options. Java is used as the primary medium to express all the design variants presented in this paper, so the basic structural constructs like classes, objects, attributes, methods and packages all could be seen as the design parameters. In the examples, we remain at the granularity of classes and interfaces.
5. *Module:* Structural elements that are strongly connected are grouped together as a module. Modules adhere to these three fundamental characteristics [13]: