ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

RAM-SE'04 — ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution

Walter Cazzola, Shigeru Chiba, and Gunter Saake
Editor

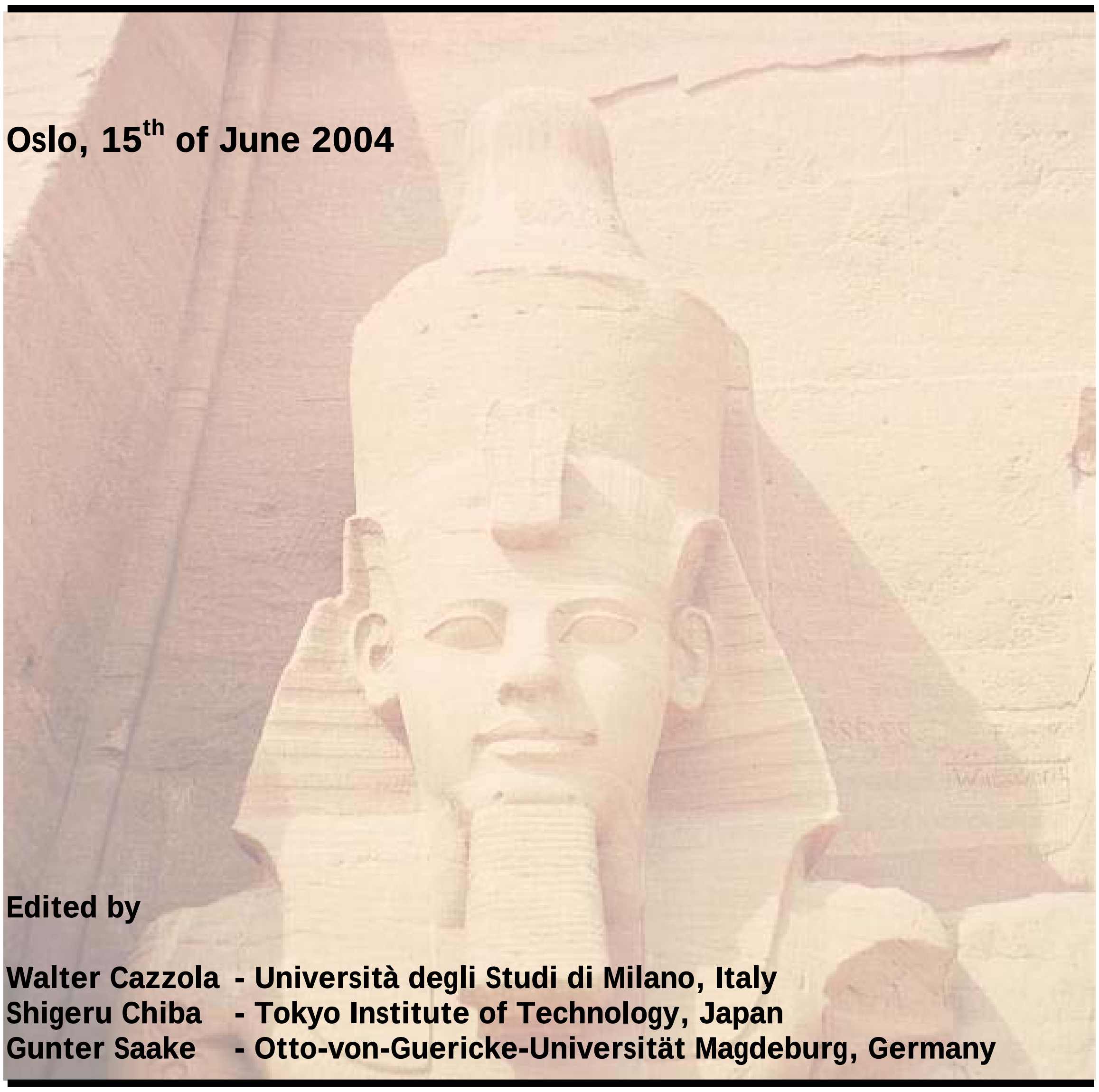
August 2004, C–196

Department of
Mathematical and
Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

RAM-SE'04 - ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution

(Proceedings)



Oslo, 15th of June 2004

Edited by

Walter Cazzola - Università degli Studi di Milano, Italy
Shigeru Chiba - Tokyo Institute of Technology, Japan
Gunter Saake - Otto-von-Guericke-Universität Magdeburg, Germany

Research report C-196, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology.

Preprint No. 10/2004 of Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg.

Foreword

Software evolution and adaptation is a research area, as also the name states, in continuous evolution, that offers stimulating challenges for both academic and industrial researchers. The evolution of software systems, to face unexpected situations or just for improving their features, relies on software engineering techniques and methodologies. Nowadays a similar approach is not applicable in all situations e.g., for evolving nonstopping systems or systems whose code is not available.

Reflection and aspect-oriented programming are young disciplines that are steadily attracting attention within the community of object-oriented researchers and practitioners. The properties of transparency, separation of concerns, and extensibility supported by reflection and aspect-oriented programming have largely been accepted as useful for software development and design. Reflective features have been included in successful software development technologies such as the Java language and the .NET framework. Reflection has proved to be useful in some of the most challenging areas of software engineering, including Component-Based Software Development (CBSD), as demonstrated by extensive use of the reflective concept of introspection in the Enterprise JavaBeans component technology.

Features of reflection such as transparency, separation of concerns, and extensibility seem to be perfect tools to aid the dynamic evolution of running systems. They provide the basic mechanisms for adapting (i.e., evolving) a system without directly altering the existing system. Aspect-oriented programming can simplify code instrumentation providing a few mechanisms, such as the join point model, that permit of evincing some points (*join points*) in the code or in the computation that can be modified by weaving new functionality (aspects) on them in a second time. Meta-data represent the glue between the system to be adapted and how this has to be adapted; the techniques that rely on meta-data can be used to inspect the system and to dig out the necessary data for designing the heuristic that the reflective and aspect-oriented mechanisms use for managing the evolution.

It is our belief that current trends in ongoing research in reflection, aspect-oriented programming and software evolution clearly indicate that an inter-disciplinary approach would be of utmost relevance for both. Therefore, we felt the necessity of investigating the benefits that the use of these techniques on the evolution of object-oriented software systems could bring. In particular we were and we continue to be interested in determining how these techniques can be integrated together with more traditional approaches to evolve a system and in discovering the benefits we get from their use.

Software engineering may benefit from a cross-fertilization with reflection and aspect-oriented programming in several ways. Reflective features such as transparency, separation of concerns, and extensibility are likely to be of increasing relevance in the modern software engineering scenario, where the trend is towards systems that exhibit sophisticated functional and non-functional requirements; that are built from independently developed and evolved COTS (commercial off-the-shelf) components; that support plug-and-play, end-user directed reconfigurability; that make extensive use of networking and internetworking; that can be automatically upgraded through the Internet; that are open; and so on. Several of these issues bring forth the need for a system to manage itself to some extent, to inspect components' interfaces dynamically, to augment its application-specific functionality with additional properties, and so on. From a pragmatic point of view, several reflective and aspect-oriented techniques and technologies lend themselves to be employed in addressing these issues. On a more conceptual level, several key reflective and aspect-oriented principles could play an interesting role as general software design and evolution principles. Even more fundamentally, reflection and aspect-oriented programming may provide a cleaner conceptual framework than that underlying the rather 'ad-hoc' solutions embedded in most commercial platforms and technologies, including CBSD technologies, system management technologies, and so on. The transparent nature of reflection makes it well suited to address problems such as evolution of legacy systems, customizable software, product families, and more. The scope of application of reflective and aspect-oriented concepts in software evolution conceptually spans activities related to all the phases of software life-cycle, from analysis and architectural design to development, reuse, maintenance, and, therefore also evolution.

The overall goal of this workshop was that of supporting circulation of ideas between these disciplines. Several interactions were expected to take place between reflection, aspect-oriented programming and meta-data for the software evolution, some of which we cannot even foresee. Both the application of reflective or aspect-oriented techniques and concepts to software evolution are likely to support improvement and deeper understanding of these areas. This workshop has represented a good meeting-point for people working in the software evolution area, and an occasion to present reflective, aspect-oriented, and meta-data based solutions to evolutionary problems, and new ideas straddling these areas, to provide a discussion forum, and to allow new collaboration projects to be established. The workshop is a full day meeting. One part of the workshop will be devoted to presentation of papers, and another to panels and to the exchange of ideas among participants.

This volume gathers together all the position papers accepted for presentation at the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04), held in Oslo on the 15th of June, during the ECOOP'04 conference. We have received many interesting submission and due to time restrictions and to quality insurance we had to choice few of them, the papers that, in our opinion, are more or less evidently interrelated to feed up a more lively discussion during the workshop. Now, a month after the workshop, we can state that we achieved our goal, presentations were interesting and the subsequent panels grew up lively and rich of ideas and proposals. We are sure that in the next months we will see many papers by the workshop attendees and fruit of such a lively discussions.

The success of the workshop is mainly due to the people that have attended it and to their effort to participate to the discussions. The following is the list of the attendees in alphabetical order.

Paulo Borba	Jordi Alvarez Canal	Ruzanna Chitchyan
Yvonne Coady	Peter Ebraert	Ahmed Ghoneim
Phil Greenwood	Günter Kniesel	Hidehiko Masuhara
Nicolas Pessemier	Sonia Pini	Tobias Rho
Yoshiki Sato	Lionel Seinturier	Maximilian Störzer
Eric Tanter	Emiliano Tramontana	Naoyasu Ubayashi
Nesrine Yahiaoui	Joseph W. Yoder	Akinori Yonezawa

A special thank is for the four chairmen (Yvonne Coady, Joseph W. Yoder, Günter Kniesel, and Hidehiko Masuhara) that governed the panels at the end of each session.

We have also to thank the Department of Informatics and Communication of the University of Milan, the Department of Mathematical and Computing Sciences of the Tokyo institute of Technology and the Institute für Technische und Betriebliche Informationssysteme, Otto-von-Guericke-Universität Magdeburg for their various supports.

July 2004

W. Cazzola, S. Chiba, and G. Saake
RAM-SE'04 Organizers

Contents

Part (I): Reflective Middleware for Software Evolution

Chairman: Yvonne Coady, University of Victoria, Canada.

Reflections on Programming with Grid Toolkits

Emiliano Tramontana[1] and Ian Welch[2]

[1] `tramontana@dmi.unict.it`,
WWW home page: `http://www.dmi.unict.it/~tramonta`,
Dipartimento di Matematica e Informatica, Università di Catania,
Viale A. Doria, 6 - 95125 - Catania, Italy
[2] `ian@mcs.vuw.ac,nz`,
WWW home page: `http://www.mcs.vuw.ac.nz/~ian`,
School of Mathematical and Computing Sciences, Te Kura Pangarau, Rorohiko,
Victoria University of Wellington, Wellington, New Zealand

Abstract. Grid applications are fragile when changes to service implementations, non-functional properties or communication protocols take place. Moreover, developing Grid applications with current toolkits result in a tangling of toolkit-specific and application-specific code that makes maintenance and evolution difficult. This paper proposes solving these problems by using reflection to open up Grid toolkits, and to allow Grid applications to be developed as if they were centralised applications. This would allow changes to be handled dependably, and a clean separation between toolkit-specific and application-specific code.

1 Introduction

Grid computing is concerned with "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organisations" [Fos01]. Virtual organisations (VOs) cover the spectrum from long-lived collaborations between static sets of organisations to short-term collaborations between dynamic sets of individuals. Grid applications are built out of heterogeneous resources offered by VOs that use a range of communication technologies to interoperate.

Grid toolkits aim to simplify the task of developing Grid applications out of Grid services. The toolkits automatically generate code for communication between clients and services but programmers must still add toolkit-specific code to both client and service implementations, particularly if non-functional properties such as security are to be implemented. This tangling of toolkit-specific code and application-specific code makes it difficult to maintain applications. For example, porting existing applications to new toolkits may require manual changes to client and service code.

As the resources comprising the VO may change while an application is running, Grid applications should be able to cope with dynamic changes such as changes to communication protocols, service interfaces or the arrival or departure of services, Grid toolkits cannot do this transparently. There is no support for switching communication protocols at runtime, and coping with the other changes requires explicit programming by the application developer.

We argue that these shortcomings of existing toolkits could be addressed by adopting work done on using reflection to treat distribution as a non-functional concern and to open up the implementation of middleware.

2 Features of the Globus Toolkit

We use the Globus Toolkit version 3.0 (GT3) as an example of a state-of-the-art Grid toolkit. GT3 is a next-generation implementation of the Globus Toolkit based on Open Grid Service Architecture (OGSA) mechanisms [Fos01]. The OGSA uses emerging web services to ease the task of building Grid programs. The next two sections describe the limitations of GT3 with respect to distribution transparency, and transparent implement of non-functional concerns.

2.1 Distribution Transparency

The GT3 toolkit [Glo03] can automatically generate stub and skeleton implementations from a Web Services Description Language (WSDL) [CCMW01] description. This provides a degree of distribution transparency but additional toolkit-specific code must be added to both the client and service implementation. At the client side, code must be written to explicitly bind an instance of a stub class before using it to access the remote service. At the service side, the service implementation must inherit from the skeleton class or provide some additional methods to allow the skeleton to delegate operations to the service implementation.

The current approach requires regeneration of the stub and skeleton code, and manual changes to source code whenever the interface to services change or what is a local resource is replaced by a service. Handling service arrival or departure is supported by web service protocols but requires explicit programming at the client side. Ideally these concerns should be transparently implemented. This would ease maintenance and evolution as once concern could be changed independently of the other.

2.2 Non-Functional Concerns

GT3 provides bindings that allow services to be hosted by a range of containers. These containers can transparently implement some non-functional concerns such as security. Containers can usually either only implement a fixed set of non-functional concerns or application-level concerns. Implementing non-functional concerns at the infrastructure level, for example changing the underlying communication protocol, cannot be done because new non-functional concerns are implemented by intercepting application-level messages.

Ideally, the toolkit should allow new non-functional concerns to be implemented at both the application and infrastructure-level. These should still be able to be declaratively specified for a service thereby allowing a clean separation between non-functional concerns and application code. This would aid maintainability and evolution. Additionally, providing a facility to install or remove non-functional concerns at runtime would allow changes without stopping a running application.

Although non-functional concerns can be enforced transparently at the service side, implementing the complementary concerns at the client side requires the programmer to add toolkit-specific code to their program. For example, when using GT3, providing the security non-functional concern at the service side simply requires adding security configuration information to a deployment descriptor for the servce. However, implementing the other half of the security concern at the client side requires some code setting the appropriate properties for the service's remote proxy.

Ideally, there should be the same separation of concerns on the client side as the service side. As for the service side, these concerns should be able to be installed and removed dynamically. Furthermore, to remove the possibility that clients and services get out of synchronisation, there should be support for synchronously installing and removing concerns at both the client and service side.

3 Proposed Approach

The proposed approach aims at supporting developers building object-oriented applications without making applications tangled with distribution related concerns and without requiring programmers to change applications when an adaptation is required to consider new technologies.

There are two aspects to this approach. (1) Programs are developed in a centralised manner and transparently distributed. (2) An open implementation of GT3 is used that allows dynamic changes at runtime.

3.1 Centralised Development

The Addistant [TSCI01] system provides distributed execution of "legacy" Java bytecode. The definition of legacy is programs that were originally developed to be executed on a single Java virtual machine (JVM). The users of Addistant specify the host where instances of classes are allocated, and how remote references are implemented. Addistant automatically transforms the bytecode at load time and uses a special configuration file to separate the specification of class location etc. from the actual program implementation.

In order to further automate distributing a centralised Java application so as to choose the most appropriate host for each object, a reflective software architecture has been proposed [DSPT02]. In such an architecture, at load time a component analyses each application class and transforms it so that allocation of instances will be performed on the basis of the calculated class parameters and the run time conditions of hosts and network. The architecture facilitates the integration of additional allocation policies to be easily inserted to consider other specific needs of classes. For example, an allocation policy could be proposed to match the needs of a class, in terms of remote services used, with the known web services, in order to find the most appropriate host for executing its instances. Moreover, through interception we can potentially change method invocations on the fly to enforce the syntax of the method allowing access to the web service.

Applying automatic bytecode transformation to the Grid environment would allow programmers to develop centralised versions of their programs and then transparently distribute them. This would avoid the need for inheritance or the manual coding necessary to support the delegation approach. In addition, should services change location then the change could be achieved by modifying the specification rather than the code. Furthermore, to make an application dynamically adaptable to changes of service location, an appropriate adaptation component can be transparently inserted into the application when transforming it into a distributed version. This component would dynamically check the location of a requested service and find its new location when necessary (i.e. when the service has migrated to a new host). Given that the Grid environment also supports resource brokering as a first-class concept then it would make sense to integrate this configuration with existing resource brokering technologies.

Because we are not primarily concerned with legacy code, i.e. where the source code is unavailable, then source-based transformation could be used. This could be useful, since the programmer could intervene to customise the code resulting from the automatic transformation. However, operating transformations on compiled code would support runtime dynamic adaptation. This would be focus of further investigation.

Automatically transforming a centralised application into a distributed one has two further benefits. Firstly, the selection of a primitive (i.e. socket, RMI, GridRPC [NMS$^+$02], etc.) that makes distributed objects communicate can be chosen only when transforming the application, so as to fit the environment where the application is going to be deployed. This approach makes the original application free of remote communication primitives, thus an application is easier to develop and evolve. Moreover, when a new mechanism is available for a different distributed environment, only an adaptation of the transformation tool has to be performed. Secondly, additional features can be added at transformation time to consider the needs of the specific target environment. Thus, components that make the communication reliable or that perform resource allocation could be added both at the client and server side along with the support for communication.

3.2 Reflective Middleware

Ideally, the declarative approach supported by existing toolkits should be retained but it should be possible to easily extend toolkits to integrate new capabilities and support dynamic changes to non-functional properties. Here, we intend to draw upon the exisiting literature on reflective middleware. Reflective middleware can be defined as “a middleware system that provides inspection and adaptation of its behaviour through an appropriate causally connected representation” [Cou]. Proponents of this idea suggest that this middleware will be able to be adapted to its environment and be able to cope with change. For example, an application deployed on a mobile device could use middleware that dynamically detected that the device was no longer plugged into an office network and could switch to using GSM for communications. This would happen transparently with no requirement for changing the application itself, what happens is that the implementation of the middleware itself is changed so the appropriate type of communications is used. Examples of reflective middleware are the DynamicTao [KRL+00], OpenORB [BCA+] or mCharm [Caz00].

The notion here is to open up the Grid toolkit in a principled way. The initial target would be the communication infrastructure. A key problem, especially if considering dynamic adaption, would be how to control the adaptation process. One way of doing this is to use the notion of adaptation policies, as is used for the K-Components framework [DC01]. In this framework an adaptation policy is specified using an extended interface description language. The policy is essentially a declarative language for writing reflective programs that can monitor and reconfigure programs by modifying the metalevel. Having a policy allows reasoning and validation of possible adaptations.

Another target would be the service container. Here the focus should be on an infrastructure for the server side that holds information about the current services that containers offer. This infrastructure would check at run time both the conditions of the containers and whether some service unavailable on a container is being requested. The aim of the infrastructure would be to dynamically and transparently transfer the requests to other containers providing the requested service, as appropriate.

The design of this infrastructure would be based upon the design lessons of existing reflective middleware. Providing first-class support for dynamic evolution of Grid applications would enhance the dependability of services because it allows clients requests to be redirected where they can be honoured, thus avoiding failures on the client side.

4 Related Work

The most closely related work in the Grid community is Othman et. al. [ODDG] who use OpenJava to simplify the implementation of an adaptive resource broker. The adaptive resource broker allows running jobs to be suspended and migrated to other hosts for execution in order to satisfy a required quality of service. Our approach differs in that it considers the goals of making Grid toolkits easier to use and supports dynamic adapation of non-functional concerns.

5 Conclusions

Existing Grid toolkits ease the job of the programmer but could be improved by removing tangling between application code and toolkit code, and allow dynamic installation and removal of non-functional concerns at both the application and infrastructure level. In this paper we have identified related work that applies reflection to distributed systems for similar purposes. We propose extending this work to develop an open Grid toolkit that hides distribution and allows a programmer to develop an application as if it was centralised rather than distributed.

References

[BCA+] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of open orb 2. IEEE Distrib. Syst. Online 2, 6 (Sept. 2001); see computer.org/dsonline.

[Caz00] Walter Cazzola. *Communication-Oriented Reflection: A Way to Open Up the RMI Mechanism*. PhD thesis, Università degli Studi di Milano, Italy, 2000.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl). Technical Report Note 15, 2001, W3C, 2001. `http://www.w3.org/TR/wsdl`.

[Cou] G. Coulson. What is reflective middleware? IEEE Distrib. Syst. Online 2, 8 (Dec. 2001); see computer.org/dsonline.

[DC01] Jim Dowling and Vinny Cahill. The k-component architecture meta-model for self-adaptive software. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88. Springer-Verlag, 2001.

[DSPT02] Antonella Di Stefano, Giuseppe Pappalardo, and Emiliano Tramontana. Introducing Distribution into Applications: a Reflective Approach for Transparency and Dynamic Fine-Grained Object Allocation. In *Proceedings of the Seventh IEEE Symposium on Computers and Communications (ISCC'02)*, Taormina, Italy, 2002.

[Fos01] I. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 6. IEEE Computer Society, 2001.

[Glo03] Globus Project. *Java Programmer's Guide Core Framework*. `http://www.globus.org`, Mar 2003. `http://www-unix.globus.org/toolkit/3.0/ogsa/docs/java_programmers_guide%.html`, last update 09/03/2003, last access 31/03/2004.

[KRL+00] Fabio Kon, Manuel Romàn, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhaes, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed systems platforms*, pages 121–143. Springer-Verlag New York, Inc., 2000.

[NMS+02] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. Gridrpc: A remote procedure call api for grid computing. `http://www.eece.unm.edu/~apm/docs/APM_GridRPC_0702.pdf`, 2002.

[ODDG] Abdulla Othman, Peter Dew, Karim Djemame, and Iain Gourlay. Adaptive grid resource brokering. in preparation.

[TSCI01] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" java software. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 236–255. Springer-Verlag, 2001.

Using Aspects to Make Adaptive Object-Models Adaptable

Ayla Dantas[1]*, Joseph Yoder[2], Paulo Borba[2]**, and Ralph Johnson[2]

[1] Software Productivity Group Informatics Center
Federal University of Pernambuco
Recife, PE - Brazil - PO Box 7851
`add,phmb@cin.ufpe.br`

[2] Software Architecture Group Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
`yoder@refactory.com, johnson@cs.uiuc.edu`

Abstract. The unrelenting pace of change that confronts contemporary software developers compels them to make their applications more configurable, flexible, and adaptive. In order to achieve this, software designers must provide flexible architectures that can more quickly adapt to changing requirements. Adaptive-Object Model (AOM) is an architectural style intended to provide this flexibility by providing a meta-architecture that allows requirements changes to be performed and immediately reflected at runtime. However, AOMs internal structures are sometimes difficult to extend and maintain. In this case, we can say AOM systems are not adaptable, although they are adaptive [1]. This paper proposes the use of Aspect-Oriented Programming in order to make AOM systems simpler to evolve, specially regarding the inclusion of new adaptive requirements.

1 Introduction

Adaptability is an increasingly important requirement of software systems. To achieve this requirement, software designers must provide flexible architectures that can quickly adapt to changing requirements. Sometimes, user requirements are such that the system will even need to adapt at runtime. In those cases, architectures are designed to adapt to new user requirements by retrieving descriptive information that can be interpreted at runtime. Those are sometimes called “reflective architectures” or “meta-architectures”.

This paper focuses on a way to enhance a particular kind of reflective architecture, called Adaptive Object-Model (AOM) architecture [2, 3], through the use of Aspect-Oriented Programming (AOP) [4]. We can make AOMs more flexible by modularizing the adaptation part of the architecture. This modularization through the use of AOP can make AOMs more maintainable and adaptable,

* Supported by CNPq.
** Partially supported by CNPq, process number 521994/96–9.

especially in relation to adaptability requirements. Adaptability here means the ability to change or be changed to fit varying circumstances, such as requirements changes. By using AOM, we organize the code in a way that makes it easier to adapt to such changes. After this organization, a requirement change can be performed, for example, by simply replacing the interpreted metadata, which can be stored on the database or in an XML file. However, AOM systems' code is usually difficult to understand and maintain [3]. In this case, we can say AOM systems are not adaptable because it is not easy to include unanticipated adaptive requirements on them. This happens because the code, not the metadata, should change in this case.

The maintainability problems with AOM's code happen because the adaptive behavior is often mixed with the business logic and GUI code of the application. Business classes that provide dynamic properties or behavior usually contain the code to obtain and interpret the data from a file or database that specifies the new properties or behavior. This is called code tangling. Besides that, such code, sometimes related to the same adaptability requirement, may be scattered throughout many classes, a phenomenon known as code scattering. When this happens, it is hard to understand and change in the code the points where new dynamic data or metadata should be obtained.

Aspect-Oriented Programming is a better way to structure Adaptive-Object Models and implement adaptive applications. For better observing the adaptability implementation problem using only AOM, and how it can be solved combining AOM and AOP, we have gradually implemented some possible adaptive behaviors on a dictionary application.

The remainder of this paper is organized as follows. The two following sections briefly present AOM and AOP respectively. In Section 4, we present the benefits of extending the AOM application by using AOP. Then, Section 5 summarizes this paper, giving some conclusions about using AOP to improve AOM.

2 AOM Overview

Many systems are developed to solve a specific problem and flexibility is not included as one of the requirements. Extending or maintaining these types of systems can be a difficult task. Simple changes can be made by parameterizing system's properties that can be read at runtime from initialization files or databases.

However, parameterizing properties will not work for complex adaptations such as adding new types of entities or properties. Adaptations can be even more complex if the system needs to add new behavior in response to a given property change, or dynamically decide which algorithm to use depending on the property type. For such adaptations, Adaptive-Object Models has been shown to be a good solution.

AOM architectures are usually made up of several smaller design patterns, such as the Composite, Interpreter, Builder, and Strategy [5], along with other dynamic patterns such as TypeObject [6], Property [7], and RuleObjects [8, 9].

Most AOM architectures apply the TypeObject and Property patterns together, which results in an architecture called TypeSquare [2].

By organizing an application using these patterns, we can represent application features, attributes and rules (or strategies) as metadata that can be interpreted in a running system. Since classes, attributes and relationships (which can be a kind of property) are represented as metadata in AOM systems, they have an underlying model based on instances rather than classes. Then, adaptations to the object-model are made by changing metadata, which can then be reflected into a running system by instantiating new EntityTypes, PropertyTypes, and Rules.

The main advantage of AOM systems is ease of change. They can even evolve to where they allow users to configure and extend their systems "without programming" and make the process of changing them quickly. However, there may be a higher initial cost in developing this kind of system, because it must be as general as possible in order to improve extensibility and making possible many sorts of adaptations. Generally, these systems are also difficult to understand because several classes do not represent business abstractions ; the object-model is represented in metadata. However, auxiliary Graphical User Interface (GUI) tools and editors can be used to alleviate this problem. Another problem presented by the AOM architecture is performance, since it is based on the interpretation of metadata.

So, before deciding to use AOM or not, an analysis of the degree of adaptability must be done. This analysis should consider which parts of the system really need to be highly adaptive. For instance, if the developed system has to be a highly configured one, or if the rules or properties of this system might change often, AOM can be a good choice, even considering the problems presented above.

3 AOP

Aspect-Oriented Programming (AOP) is a technology intended to provide clear separation of crosscutting concerns [4]. Its main goal is to make design and code more modular, meaning the concerns are localized rather than scattered and have well-defined interfaces with the rest of the system. In this way, AOP solves the issues raised by some design decisions that are difficult to cleanly capture in code [10]. Those issues are called aspects, and AOP is intended to provide appropriate isolation, composition and reuse of the code used to implement those aspects.

This programming paradigm proposes that computer systems are better programmed by separately specifying the various *concerns* (properties or areas of interest) of a system and some description of their relationships and then, by using AOP environment, these concerns are composed or weaved together into a coherent program [11]. This is especially useful when the *concerns* considered are *crosscutting*. Crosscutting concerns are those that correspond to design decisions that involve several objects or operations, and that, without AOP, would lead to different places in the code that do the same thing, or do a coordinated simple thing. Some examples of *crosscutting concerns* are: logging, distribution, persistence, security, authentication, performance, transactions integrity, etc.

AspectJ is one of the most widely used AOP languages. It is a general-purpose aspect-oriented extension to Java [12]. This language supports the concept of join points, which are well-defined points in the execution flow of the program [13]. It also has a way of identifying particular join points (pointcuts) and change the application behavior at join points (advice).

4 Using AOP to improve AOMs

In this section, we describe how AOP can solve some of the problems noticed in the AOM implementation of some adaptive requirements in a dictionary application. In order to do that, we have implemented some adaptabilities purely using AOM and then, implemented them again combining AOM with AOP, through the *Adaptability Aspects* [14] pattern. By doing that, we could verify when the AOP use was appropriate and when it was not. The *Adaptability Aspects* is an architectural pattern for structuring adaptive applications using aspects. It was used here in order to make a better use of AOP.

4.1 Using AOM in the Dictionary

The dictionary application is a cellular phone application that is capable of translating words from English into Portuguese. It that presents five screens: presentation, main menu, instructions, info and search screens. The main menu presents three options: Query, Instructions and More info...; info screen displays the source and destination translation languages of the dictionary; and in search screen the search is requested and the translation results are shown.

Considering this simple dictionary application, we have implemented some adaptability requirements. One of them intends to provide dynamic source and target translation languages. Another one is able of providing dynamic search engines for the dictionary, making it able of changing the way it performs a search (e.g. if on memory, on a server, etc). Another adaptability requirement is responsible for providing dynamic properties for the dictionary, which can be shown in info screen or in another application screen where they can be edited. The property types may change frequently, changing the way they are shown on the application or even their ability to be edited or not.

To illustrate AOM's use, we show the implementation of one of the dynamic requirements presented above, which we call "Dynamic Dictionary Properties". In order to implement an adaptation with AOM, we must evaluate which patterns should be used to reorganize the application.

As we have previously seen, the info screen presents two properties of the application: the source and the destination translation languages. These properties are implemented as fields of a class called `InputSearchData`, which is part of the model of the application considering the MVC pattern. If the user requests a new property, the developer may add a new field to this class. However, the application may need dynamic properties, that may exist or not and even those the developer cannot anticipate. To provide these properties and to avoid codification rework when a new property is requested, we can apply the Property [7] pattern on the dictionary application as shown in Figure 1. Then, instead of many fields representing different properties, the `InputSearchData` class presents a collection of properties that may be stored in a hashtable, for example.

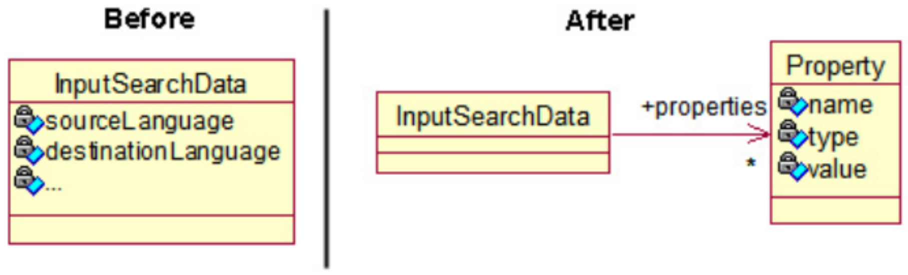


Fig. 1. Property pattern used in the dictionary application

The TypeObject [6] and Strategy [5] patterns can also be used in addition to the Property pattern in order to validate the values of the dictionary properties. Therefore, for each `Property`, there is a `PropertyType` instance associated with it and there are also Strategies for validating dictionary properties. This validation can be used for determining whether a given Property is editable or not and whether it should be presented in a given screen, such as a screen for getting user preferences. Part of the new organization of the application in order to deal with dynamic properties and strategies related to those properties is illustrated by Figure 2.

In the case of the dictionary application, we have implemented an interface, `PropertyValidator`, and classes implementing the `isValid` method, which evaluates if a property is valid according to its type. With such hierarchy, we can use the Strategy pattern to easily change the property validator, by using the `StrategyObject` class. Then, for dynamically changing the validation of the properties shown on a giving screen (such as info screen), we may simply change the `specificInstanceName` attribute of the `StrategyObject` responsible for that.

Following the organization presented by Figure 2, dynamic dictionary properties are obtained from the `InputSearchData` class. The strategies, properties, and their types are represented in XML files that change from time to time and must be interpreted.

There are at least two classes we have implemented that would use the dynamic properties managed by the `InputSearchData` class: the `InfoScreen` class,

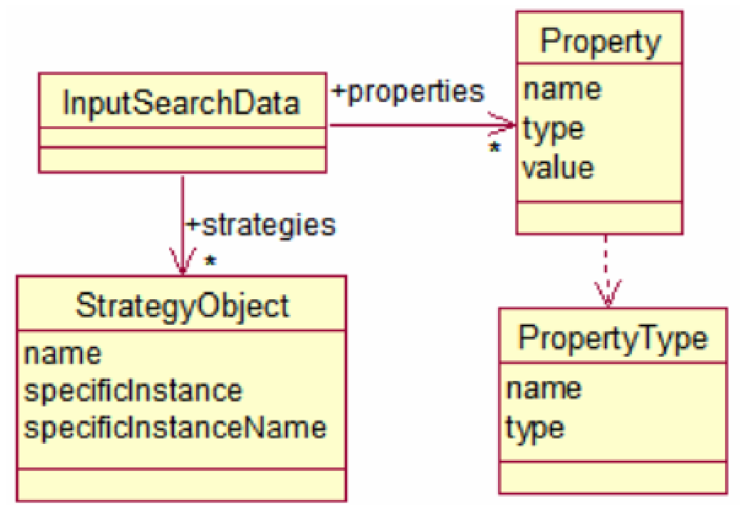


Fig. 2. Using the Properties, TypeObject and Strategy Pattern on the dictionary

which displays all dictionary properties; and the `UserDataScreen`, a new screen responsible for showing dictionary properties that can be edited, such as user name and password.

Reading in the dynamic data from the XML file is done by using an Adaptation Data Provider module implemented using AOM. This module is part of the *Adaptability Aspects* pattern, but can also be used by adaptive systems that do not use aspects (see [14]). It is responsible for obtaining the dynamic properties, types and strategies that represent part of the application. The main class in this module is the `AppAOMManager` class. Any adaptation data is obtained through this class.

In order to build the `InfoScreen` or `UserDataScreen`, the dictionary dynamic properties must be obtained from the `AppAOMManager` and validated, according to their types, which can be dynamically defined. According to this implementation, by simply changing the type of a property, or by including a new property in the metadata that is interpreted, we can dynamically change the application.

This metadata might change frequently and the application might behave differently according to these changes. Therefore, the Adaptation Data Provider objects should be reloaded from time to time. Besides that, the dynamic parts of the application should access this module at certain execution points. For example, before showing `InfoScreen` or `UserDataScreen`, we must rebuild them. To do so, we may request an update of dictionary properties shown by these screens.

The adaptability data is requested when some dynamic information is necessary for the application. This can be done during the dictionary application startup or can be a frequent action performed in several parts of the code. This is especially true when we remember that AOM systems are also known for their ability to immediately adapt to metadata changes [3]. The Observer pattern can also be used to notify an application that updates to the object-model have been made.

In our pure AOM implementation of the adaptive dictionary application, several points of the code call methods to update the application objects due to changes in the adaptability data. This is a problem known as code scattering. For example, in order to update dictionary properties according to dynamic data, we may have to include reload invocations and application object updates before the `InfoScreen` or the `UserDataScreen` are displayed. We might also have to make calls to the reload invocations after the application startup, at the moment some classes are instantiated, etc.

As we can see, the `DictionaryController` class calls methods intended to obtain new dynamic data and reorganize the application objects and the screens to be shown. However, this behavior is not directly related to this class business logic, and can lead to poor maintainability.

For reload operations at certain execution points, we invoke methods from the `InputSearchData` class that request data from the Adaptation Data Provider module. This request also synchronizes some properties and associates validation strategies to these properties. After the `InputSearchData` class initialization, we must also perform some kinds of reloads.

In fact, several problems arise while implementing adaptability requirements using AOMs or similar techniques and they are not specific of dictionaries. One of them is code tangling. This happens in the dictionary because the `InputSearchData`, or any other class that is supposed to obtain dynamic data, has to know about synchronization mechanisms that may vary according to the Adaptability Data Provider module implementation. Besides that, what must change or not change when dynamic data is obtained may also vary according to new requirements. Adaptive applications, especially very dynamic ones, may change a lot the execution points where they must adapt (obtain new data and change). Consequently, if we want to change those points, we need to modify adaptation code scattered throughout many classes and for many versions. Therefore, code tangling and scattering make the adaptability implementation hard to change and thus less adaptable. As the adaptability concern generally crosses many parts of the code, we can say it is a crosscutting concern.

In order to solve those problems and provide a higher degree of reuse and the ability to easily plug in/out adaptation features, we propose the use of aspects. Aspects help isolate the configuration of dynamic adaptations and thus make AOMs more adaptable. To illustrate this, we show in the following how we have extended the dynamic dictionary properties concern implementation using the AspectJ [12] language.

4.2 AOP Use

In this section we describe how AOP can solve some of the problems noticed in the AOM implementation pointed out in the previous section. In order to do that, we use some ideas of the *Adaptability Aspects* pattern. We primarily deal with the Dynamic Dictionary Properties adaptability requirement, pointing out the utility of AOP in improving its AOM implementation.

By using aspects we can modularize adaptations and the application execution points that should trigger them (for example, a given method call or execution, a field get or set, etc.). An aspect in AspectJ is a modular unit of crosscutting implementation. As we have previously seen, the configuration of the adaptations is a crosscutting concern, because it generally crosses many parts of the code and can make that code difficult to understand. For implementing this concern in the case of the dynamic properties, we have used an adaptability aspect called `DynamicProperties`. It is part of the Adaptability Aspects module of the *Adaptability Aspects* pattern. It is responsible for verifying if an adaptation should be performed and then performing the necessary changes, using for both tasks elements of other pattern modules.

By defining pointcuts, aspects identify collections of points in the execution flow of a program where behavior changes must happen. In the case of the `DynamicProperties` aspect, some of the pointcuts we have defined are called `showingInfoScreen` and `inputSearchDataCreation`.

In the first one, we identify the execution of the `showScreen` method, from the `DictionaryController` class, when the screen to be shown is `InfoScreen`. A similar definition is also done for the `UserDataScreen`. In order to perform some actions when these execution points are achieved, we must define some advice. In the advice, as in the pointcut definition, there must be some parameters, which are used to expose the application context at those execution points. With a `before` advice that corresponds to the `showingInfoScren` pointcut we may define what must be done before this execution point. The second pointcut definition illustrated above corresponds to the execution of the `InputSearchData` class constructor. By using an `after` advice, we may define the actions to be performed after this execution. The auxiliary methods called inside these advice reload the source of adaptability data and synchronize the current application objects, or part of them, according to the new data. If we want to change the adaptation points, or what must be done at those points, we simply change the pointcuts or the advice declarations respectively. Both are defined in a modular unit, and the access to the source of dynamic data is confined in the aspect code (or auxiliary classes used by it).

In AspectJ, as in other AOP languages, there is a process for composing the base source code or even compiled code with the aspects code. This process is known as weaving. If the user does not want dynamic data for the dictionary properties, the aspect responsible for that is simply not provided as input for the weaving process. If this aspect is provided, the configuration of the adaptability is localized. So, with aspects, it becomes easier to configure the adaptation. Therefore, the adaptation itself is adaptable.

There is also an additional benefit: the AOM code and the ways to integrate it with the normal code are separated. Therefore, we can more easily change the way we want a system to adapt. We can also more easily change or evolve the non-AOM part of the system.

5 Conclusions

As we could see, adaptability is becoming a common requirement, and implementing it can be hard. Adaptive-Object Models have been, to some extent, successfully used to implement dynamic systems. Understanding AOMs can help developers more quickly build systems that are highly flexible, because part of these systems is represented in metadata that can be easily changed. However, AOMs sometimes lead to solutions that can be hard to maintain in order to include new adaptive capabilities or change the code of the existing ones. This happens because besides reorganizing the application by using some patterns, it also suggests that the system behavior and dynamic elements must be represented using metadata, which is interpreted at runtime. This interpretation of metadata and associated actions are scattered throughout many classes. This makes the business logic code and GUI code become mixed with the code for providing the adaptability.

There may also be some business rules which do not need to be adaptive. By mixing adaptability code with fixed code, code tangling arises. This can lead to problems while maintaining the system; specifically if an extension to the AOM is needed.

In order to minimize those problems, and make AOMs more adaptable, we use Aspect-Oriented Programming. From the previous sections, we could see that AOP can be useful for introducing adaptability with AOMs. It brings two main advantages:

– Makes it easier to change the execution points where dynamic data must be obtained;
– Isolates the adaptability actions from the application business logic and GUI code.

This is a result of the modularization property provided by AOP through the use of aspects. In AspectJ, we change the “adaptability points” by giving new pointcut definitions that generally expose application objects (the pointcut parameters). Then, we define the adaptability actions by using advice (`before`, `after` or `around`), which explore the exposed instances in order to change the application behavior.

By using AOM, we make our applications able to adapt at runtime to users’ or developers’ new requirements. This happens because we represent the parts of the systems intended to be dynamic in metadata that is interpreted and we organize the system using some patterns. However, retrieving dynamic data and updating application objects is a crosscutting concern related to many adaptability requirements. Implementing it using pure OO programming may lead to code that is difficult to understand and evolve. Therefore, we propose the use of aspects for modularizing this concern in each adaptability requirement.

Besides improving AOM implementations, AOP can also be used to implement some of AOM patterns avoiding direct changes in the application code. However, in some cases, this may bring maintainability problems, because the code may become more difficult to understand, as we could see in other adaptability requirement implementation.

By using the *Adaptability Aspects* pattern, lightweight aspects are used in order to avoid problems that may result from a bad use of aspects. The aspects should only be used to avoid code tangling and scattering while implementing adaptability, and where they allow a better comprehension of the code.

After this work, we conclude that AOM and AOP are a good combination in order to provide flexible applications that are easy to evolve both by interpreting metadata or by changing source code. In the latter case, this will happen because the adaptability configuration will be better isolated.

References

1. Lieberherr, K.: Workshop on Adaptable and Adaptive Software. In: Addendum to the Proceedings of the 10th annual Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press (1995) 149–154
2. Yoder, J.W., Balaguer, F., Johnson, R.: Architecture and Design of Adaptive Object-Models. ACM SIGPLAN Notices **36** (2001) 50–60
3. Yoder, J.W., Johnson, R.: The Adaptive Object-Model Architectural Style. In: Working IEEE/IFIP Conference on Software Architecture 2002(WICSA), Montral, Qubec, Canada (2002)
4. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing Aspects of AOP. Communications of the ACM **44** (2001) 33–38
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
6. Johnson, R., Wolf, B.: "Type Object". Pattern Languages of Program Design 3. Addison-Wesley (1998)
7. Foote, B., Yoder, J.: Metadata and Active Object-Models. Collected papers from the PLoP '98 and EuroPLoP '98 Conference Technical Report wucs-98-25, Dept. of Computer Science, Washington University (1998)
8. Arsanjani, A.: Rule Object Pattern Language. In: Proceedings of PLoP2000. Technical Report wucs-00-29, Dept. of Computer Science, Washington University Department of Computer Science. (2000)
9. Arsanjani, A.: Using Grammar-oriented Object Design to Seamlessly Map Business Models to Component-based Software Architectures. In: Proceedings of The International Association of Science and Technology for Development, Pittsburgh, PA (2001)
10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect–Oriented Programming. In: European Conference on Object–Oriented Programming, ECOOP'97. LNCS 1241, Finland, Springer–Verlag (1997) 220–242
11. Elrad, T., Filman, R.E., Bader, A.: Aspect-Oriented Programming. Communications of the ACM **44** (2001) 29–32
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting Started with AspectJ. Communications of the ACM **44** (2001) 59–65

13. Team, A.: The AspectJ Programming Guide. At http://www.eclipse.org/aspectj (2003)
14. Dantas, A., Borba, P.: Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications. In: Third Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'2003, Porto de Galinhas, Brazil (2003) Temporary version at http://www.cin.ufpe.br/~sugarloafplop/-acceptedPapers.htm.

RAMSES: a Reflective Middleware for Software Evolution

Walter Cazzola[1], Ahmed Ghoneim[2], and Gunter Saake[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it

[2] Institute für Technische und Betriebliche Informationssysteme,
Otto-von-Guericke-Universität Magdeburg, Germany
{ghoneim|saake}@iti.cs.uni-magdeburg.de

Abstract. Software systems today need to dynamically self-adapt against dynamic requirement changes. In this paper we describe RAMSES a reflective middleware whose aim consists of consistently evolving software systems against runtime changes. This middleware provides the ability to change both structure and behavior for the base-level system at run-time by using its design information. The meta-level is composed of cooperating objects, and has been specified by using a design pattern language. The base objects are controlled by meta-objects that drive their evolution. The essence of RAMSES is the ability of extracting the design data from the base application, and of constraining the dynamic evolution to stable and consistent systems.

Keywords: Software Evolution, XMI, UML, Reflection, Meta-Objects.

1 Introduction

Many object-oriented information systems today need to dynamically adapt themselves against runtime changes. Some of the changes such as modify its structure and behavior may cause the base-systems to behave in an unexpected way. Therefore, software systems need to be capable of dynamically adapting their structure and behavior at runtime and of checking their consistency to face sudden changes. Software development asked for the way to modify the base objects at runtime without going to rebuild the application again. It requires a new approach, which adapts the base application as well as on advances in software technology. This new perspective reifies the design data of the base application and by modifying such reification it adapts the base application against runtime changes.

A topical issue in the software engineering research area consists of producing software systems able to adapt themselves to environment changes by adding new and/or modifying existing functionalities. There are a number of mechanisms for obtaining adaptability. One of these mechanisms is *reflection* [6, 2]. A non-stoppable software systems provide an excellent way to dynamically adapt itself against runtime changes at its environment. A non-stoppable systems are characterized by long life cycle. Usually, these systems are deployed to be continuously online for several years. During this

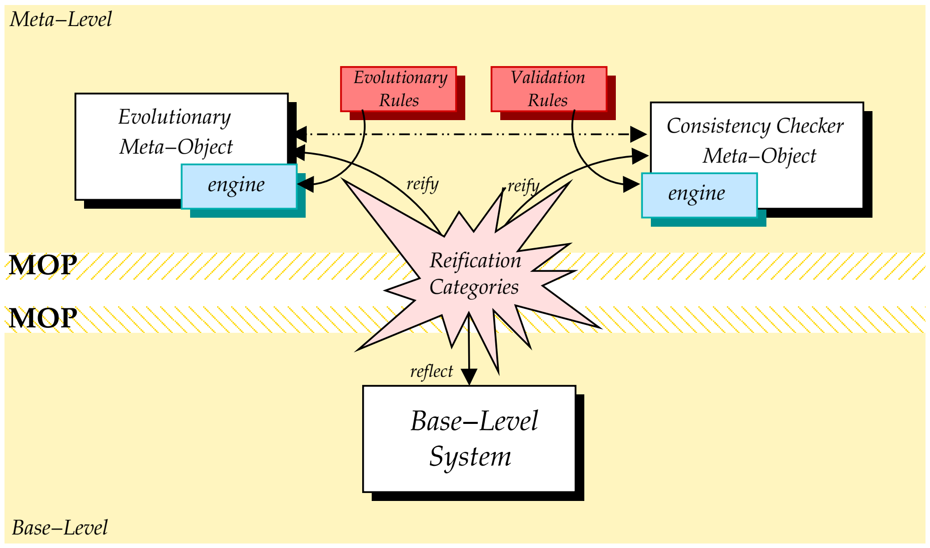


Fig. 1. RAMSES architecture.

lifetime we want to be able to dynamically maintain and adapt these systems against runtime events without bring the whole system to a halt.

We propose an infrastructure to dynamic adapt software systems. In our approach,we adopt a reflective system [4, 5], that allows to render self-adaptable at runtime the base-level system. The meta-level systems is composed of an interpreter engine for managing the evolution and validating consistency processes for runtime changes.

The meta-level behavior is described by a family of patterns [3], the meta-level manages both the evolution and consistency of the base-level system. The cooperative meta-objects at the meta-level consult the engines (see figure 1), and adapting the reified objects for dynamic behavior. Changes to the reified system can be made at runtime and are immediately reflected to its base-components. The evolution and consistency are not hard-coded, neither are they generated. Instead, we build a reflective framework of the base-systems that can be automatically self-adapted for any changes to be active long-life span. Our reflective architecture define two cooperative meta-objects (evolutionary and consistency) both of them refer to the engine to evolve the system and validate the consistency of its semantics at runtime.

2 RAMSES Overview

RAMSES (Reflective and Adaptive Middleware for Software Evolution of Systems) performs two phases to carry our self-adaption. In the first phase, the RAMSES's meta-level extracts the design information as XMI schemas from the base application and it reifies them in the meta-level to constitute the meta-data. Whereas, in the second phase, RAMSES's meta-level plans the dynamic adaptation of the base-level system, gets the runtime events, evolves the meta-data against the detected event, checks the consistency, and finally reflects the modified data to the base-level. This infrastructure is considered to be dynamically adaptive because changes in the execution environment cause objects, attributes and collaborations to be created and modified at runtime to achieve new behaviors not previously foreseen by the original application. This goal is achieved by:

– adopting a reflective architecture which reifies system design information and reflects back the changes on the system design;
– manipulating the design information and checking the system consistency against evolution in the meta-level;
– using configurable rules to govern the system evolution through its design information.

Adaptation and validation are respectively driven by a set of rules which define how to adapt the system according to the detected event and the meaning of system consistency.

2.1 The Reflective Architecture

To render a system self-adapting[3], we encapsulate it in a two-layers reflective architecture as shown in Fig. 1. The base-level is the system that we want to render self-adapting whereas the meta-level is a second software system which reifies the base-level design information and plans its evolution when particular events occur. By using a reflective architecture, thanks to the transparency and separation of concerns properties of reflection, we can render self-adapting every software system without changing its code.

At the moment, this approach allows two kinds of dynamic evolution: *structural* and *behavioral* evolution. This limitation is due to the fact that we just consider the following design information related to the base-level system:

– *object model*, which describes objects and their relationships; this model represents the structural part of the system;
– *sequence diagrams*, which trace system operations between objects (inter-object connection); and
– *statecharts*, which represent the evolution of the state of each object (intra-object connection) in the system.

The meta-level is responsible of dynamically adapting the base-level and it is composed of some special meta-objects, called *evolutionary meta-objects*. There are two types of evolutionary meta-objects: the *evolutionary* and the *consistency checker* meta-objects (see Fig. 1). Their goal consists of consistently evolving the base-level system. The former is directly responsible for planning the evolution of the base-level through adding, changing or removing objects, methods, and relations. The latter is directly responsible for checking the consistency of the planned evolution and of really carrying out the evolution through the causal connection typical of each reflective system.

[3] By the sentence *to render a system self-adapting* we mean that such a system is able to change its behavior and structure in according with external events by itself.

2.2 Design Information as Meta-Data

Through the causal connection, the base-level system and its design information are reified into *reification categories* in the meta-level. Classic reflection takes care of reifying the state and every other dynamic aspect of the base-level system, whereas the design information provides a reification of each design aspect of the base-level system such as the collaborations among its components. The reification categories content is the main difference of RAMSES with respect to standard reflective architectures. Usually, reifications represent the base-level system behavior and structure not its design information. Reification categories are meta-data that represent the base-level system design information in the meta-level. Both evolutionary and consistency checker meta-objects directly work on such representatives and not on the real system, this allows a safe approach to evolution postponing every change after validation checks. As described in [3] when an external events occur as a reaction, the evolutionary meta-object proposes an evolution to the consistency checker meta-object which validates the proposal and schedules the adaptation of the base-level system if the proposal is accepted.

2.3 Evolution Planning and Validation

Adaptation and validation are respectively driven by a set of rules which define how to adapt the system in accordance with the detected event and the meaning of system consistency.

To give more flexibility to the approach, these rules are not hardwired in the corresponding meta-object rather they are passed to a sub-component of the meta-objects themselves, respectively called *evolutionary* and *validation engines*, which interpret them. Therefore, each meta-object has two main components: (i) the core which interacts with the rest of the system (e.g., detecting external events/adaptation proposals, or manipulating the reification categories/applying the adaptation on the base-level system) and implementing the meta-object's basic behavior, and (ii) the engine which interprets the rules driving the meta-object's decisions.

The evolutionary meta-object plans the evolution of the base-level system when an event that requires its adaptation occurs. The evolutionary meta-object passes to its engine all the data about the occurred event and the entities that could be involved by the evolution. On this basis, the engine chooses and applies a group of evolutionary rules that serve to build the plan for evolving the base-level exploiting the reified meta-data. The evolutionary meta-object proposes the planned evolution to the consistency checker meta-object which validates its soundness. Similarly to the planning phase, the consistency checker meta-object demands the validation to its engine that exploits the validation rules and the base-level's meta-data. The plan for the evolution is concretized on the base-level if and only if the consistency checker considers its application sound otherwise a new plan has to be designed.

3 Benefits and Drawbacks

Our approach to software evolution has the following benefits:

– evolution is not tailored on a specific software system but depends on its design information;
– evolution is managed as a nonfunctional features, therefore, can be added to every kind of software system without modifying it;
– evolution strategy is not hardcoded in the system but it can dynamically change by substituting the evolutionary and validation rules; and
– RAMSES decreases the complexity of evolution and validation by defining only one meta-level. That represent the meta-processes of maintaining and evolving the meta-data.

Unfortunately there are also some drawbacks: i) we need a mechanism for converting UML diagrams in the corresponding XMI schemas (problem partially overcome by using Poseidon for UML [1]); ii) decomposing the evolution process in evolution and consistency validation could be inadequate for evolving systems with tight time constraints.

4 Conclusion

We have presented the RAMSES (Reflective and Adaptive Middleware for Software Evolution of Systems) middleware whose aim consists of self-adapting object-oriented systems against environmental changes. In this paper we have given an overview of the whole reflective architecture for dynamically evolving and validating consistency of a software system. The main features of our infrastructure can be highlighted as follows: 1) it allows to extract the system design information as XMI schemas from base objects; 2) by using MOP capability the XMI schemas will be reified to constitute the meta-data used in the meta-level; 3) both evolution and consistency are managed by the collaborations between meta-objects. Finally, 4) by using reflection we reflect the modified design information to the base-level. We are currently working on implementing a prototype of RAMSES.

References

1. Marko Boger, Thorsten Sturm, and Erich Schildhauer. *Poseidon for UML Users Guide*. Gentleware AG, Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany, 2000.
2. Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.
3. Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Pavel Hruby and Kristian Elof Søresen, editors, *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02)*, pages 35–54, Højstrupgård, Helsingør, Denmark, on 20th-22nd of September 2002. Microsoft Business Solutions.
4. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Reflective Analysis and Design for Adapting Object Run-time Behavior. In Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors, *Proceedings of the 8th International Conference on Object-Oriented Information*

Systems (OOIS'02), Lecture Notes in Computer Science 2425, pages 242–254, Montpellier, France, on 2nd-5th of September 2002. Springer-Verlag. ISBN: 3-540-44087-9.

5. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrich, John-Jules Meyer, and Mark D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science, pages 69–84. Springer-Verlag, Heidelberg, Germany, February 2004.
6. Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

Part (II):
Software Evolution and Refactoring

Chairman: Joe Yoder, The Refactory Inc & Joe Yoder Enterprises.

AOP and Reflection for Dynamic Hyperslices

Ruzanna Chitchyan, Ian Sommerville

Computing Department, Lancaster University, Lancaster, LA1 4YR, UK
{rouza | is}@comp.lancs.ac.uk

Abstract. In this paper we present a Model for Dynamic Hyperslices which uses a particular Aspect-Oriented (AO) approach – Hyperspaces – for decomposition and reflection as a means for composition of software modules. This model allows for structured, dynamic, incremental change introduction and rollback, thus, supporting run-time evolution yet preserving component modularity. The applicability of the model is illustrated through a schema adaptation scenario.

1. Introduction

Aspect-Oriented software development (AOSD) is a methodology for software development with an emphasis on the separation of concerns principle. AOSD takes the next step, after OO, in developing well modularised software, by separating the crosscutting concerns. A significant part of work in AO community focuses on software evolution support [1, 2] as well as dynamic change due to run-time weaving of aspects [3-5]. AOSD itself provides new mechanisms, such as joinpoints, pointcuts and introductions, that can be used to facilitate dynamic change, but it does not explicitly provide any structured methodology to manage and support dynamic change in software.
Such a structure, however, is provided by reflective approaches to software development [6, 7]. In reflection the main emphasis is on transparent manipulation of the base level via adaptation of the meta level. Meta level is a handle for controlling the base. This is particularly useful for "non-invasive" run-time adaptation of the base code and dynamic re-configuration using meta-object protocols.
Thus, we suggest that the meta-object protocols of reflection provide control and manipulation mechanism that, in combination with the modularisation and change introduction capabilities of AO, could lead to well-modularised, dynamically evolvable software systems.
This approach has been adopted in development of the Dynamic Hyperslices Model briefly outlined and illustrated through an example in section 2 of this paper. Some implementation-related issues for the model are examined in section 3 and the discussion is summarised in section 4.

2. The model for Dynamic Hyperslices

2.1 Outline of the model

The Dynamic Hyperslices model [8, 9] is intended to support the dynamic evolution of non-stop systems, i.e. systems that cannot be easily taken offline due to high costs of their downtime (e.g. telephone and banking), environmental safety (e.g. nuclear plants), loss of human life (e.g. life support systems) and such like. The model uses the Hyperspaces approach [10-12] to decompose the software system into "single-minded" modules (e.g. a module for `Health` feature of the `Person` object) and the power of reflection [6, 7] along with filters (as discussed in the Composition Filters approach [13]) and architectural connectors [14, 15] for unit composition and run-time manipulation.
The Dynamic Hyperslices aims to provide a composition mechanism that allows all the primary concerns, decomposed in accordance with the Hyperspaces approach, to endure in the composite concerns after composition.
In the Hyperspaces approach [10-12] the software is modelled as a set of modules (called hyperslices) each of which represents only one single concern. These hyperslices are then composed using matching units (e.g. method names) in different hyperslices as join-points. Composition-related concerns are not treated as first class entities, but are transitory units which integrate with primary hyperslices into a composed unit. Composition is a compile-time process and the final composed module has no recollection of its composite parts.
We maintain the decomposition principles of Hyperspaces, but differ in our composition approach. We use connectors for composition. Filters form part of our *composition connectors* where connectors connect hyperslices and not (necessarily) complete object classes or (OO) components. Our connectors don't simply match provided/required services, or specify roles for connected components, but rely on a dynamically updateable composition strategy to build up functionality of coarser-grain components (e.g. object classes) from primary hyperslices[1], as well as carry out the communication between the member hyperslices at run time.
In short, the model:

- Uses the Hyperspaces decomposition approach in separating concerns into single-minded hyperslices (or primary concerns).
- Requires that an additional dimension for Composition concerns is specified in each Hyperspace-type decomposition. This additional dimension contains connector-concerns. At the composition stage the connector concerns are used to compose other concerns.
- Utilises a composition connector to integrate any primary/composite concerns. Consequently, any interaction between other concerns is channelled through a set of connectors.

[1] Thus, the composition strategy in the connectors can be perceived as a kind of "merger algorithm" for producing higher order artifacts. Here the "merger" is performed through run-time message manipulation within connectors, without physically merging the hyperslices.

- Provides connectors with capability to reflect upon their immediately connected concerns, while still keeping these internals hidden from all other connectors and hyperslices.

The Dynamic Hyperslices approach is illustrated using an example of dynamic schema adaptation[2] in the following sub-section.

2.2 Illustrative Example

Dynamic database schema adaptation is desirable since, in database-centric environments (for instance in banking sector), downtime of the central database system is very costly. Consequently, the ability to seamlessly incorporate change into a database schema at run-time promises significant financial gains.

Figure 1(a) below depicts a certain (oversimplified) Object Database schema. The organisation that owns this database has kept records of its clients, the (financial) services that it provides and the registrations that its clients have undertaken for the provided services (e.g. Instant Savings Accounts). Assume the organisation wants to improve its service provision and so intends to encourage the clients to fill in newly introduced questionnaires about the services they use. To motivate the clients, for each filled in questionnaire the clients registration record will be credited with a free quantity of service (e.g. extra 0.1% of interest gained).

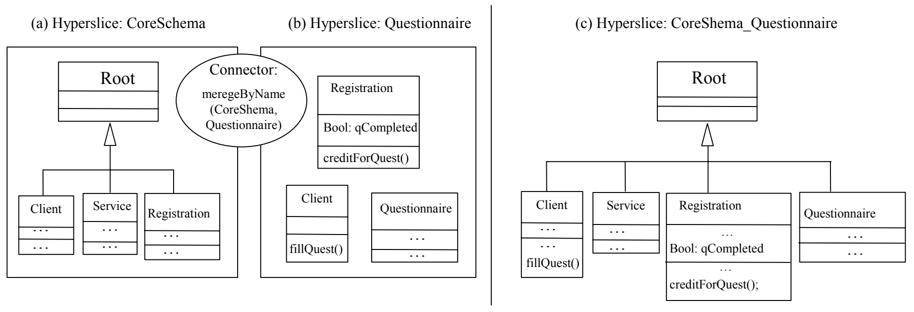


Fig. 1. Illustration of the model for Dynamic Hyperslices.

Using the Hyperspaces decomposition approach, we adopt the existing schema as a composite hyperslice `CoreSchema` (presented in Figure 1.a). Then, we design and develop the newly required set of functionality as a separate hyperslice `Questionnaire` (presented on Figure 1.b), that has only those concerns that deal with the issues related to the questionnaire. The `Questionnaire` hypeslice consists of newly introduced `Questionnaire` class, `Client` class which has only one method (`fillQuest`) to allow clients to fill in the questionnaires, and the `Registration` class which has a Boolean variable `qCompleted` to indicate wheather the questionnaire for the given registration has been completed, and a method for crediting the registration with additional free quantity of service for each completed questionnaire.

[2] More about this subject can be found in [16].

The Connector, depicted as an oval, linking `CoreSchema` and `Questionnaire` hyperslices is the run-time composition mechanism that combines the separate hyperslices into a composite `CoreSchema_Questionnaire` hyperslice view (presented in Figure 1.c). However, while the general view of the updated schema will be that presented on the right of Figure 1 (i.e. part *c*), the initially independent hyperslices for core schema (1.a) and the questionnaire (1.b) will be retained intact as presented on the left side of Figure 1 (i.e. parts *a*, *b*, and the connector). The connector will retain the composition information which leads to the change of the schema, allowing for rollback to the previous version, if required. The connector is also the communication mechanism between the composed hyperslices as well as their clients.

Thus, in this section we have briefly outlined the possible applicability of the Dynamic Hyperslices approach to a database schema evolution problem. We have discussed how with the Dynamic Hyperslices approach a coherent view of the evolving database schema can be retained, along with the details of the historical change (contained in connectors), allowing for rollback, if required. Yet, our appraoch avoids the space usage overheads and coarseness of retained change history, as is the case with the traditional schema versioning approaches (when several versions of the same schema are kept) [17, 18]. We also avoid the pitfalls of class versioning approaches [19, 20] – when a copy of each new version of each class is retained – which result in overcomplicated schema and loss of a single coherent view on it (due to many versions of the same class).

3. Discussion on Implementation Issues

The Dynamic Hyperslices model is currently under development. In this section, we talk about some initial ideas for its implementation.

As discussed earlier, the composition in our model is carried out through reflective adaptation. The partial structure of the meta-object level of the model is presented below:

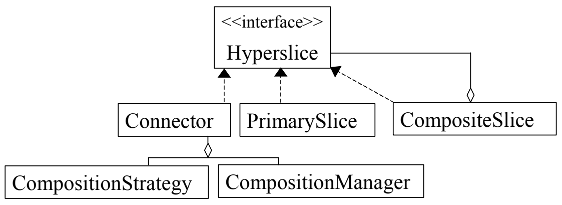


Fig. 2. Partial structure of the meta-model for Dynamic Hyperslices

Figure 2 states that all primary and composite slices and the connectors are hyperslices. The connectors, primary, and composite slices can be composed into new composite slices. The composition details are provided through the composition strategy which is a part of the composition connectors. The composition process is monitored and validated by the `CompositionManager` element of the connectors.

The base level (i.e. application) programmer using the Dynamic Hyperslices model does not need to be aware of the above meta-level. The link between the base and the meta levels is established at load time via an AspectJ aspect which introduces and initialises the corresponding reference variables in the base and meta levels.

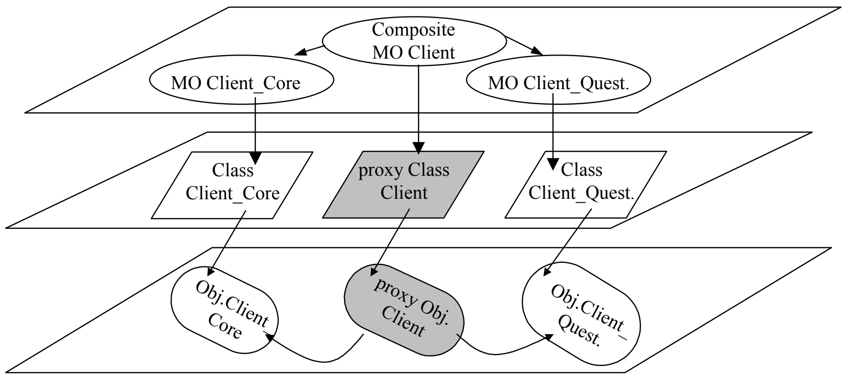


Fig. 3. High-level workings of the model for Dynamic Hyperslices.

The high-level working of the system, also illustrated in Figure 3 above, is as follows:

- The Base and Meta level link is established at load time, with a meta-object created per each loaded class;
- Composed slices are represented by a proxy class at the base level and a composite meta-object at the meta level. Instantiation of a composed class results in instantiations of its components;
- All calls to the base level objects are passed to their meta-objects. The meta-objects resolve each calls in accordance with the composition strategy used and filter it down the composition chain to the resolved primary slice which executes the call;
- The topmost composite meta-object refers to the “combined” interface of all composition participants. This combined interface is displayed to all clients of the composite slice

The implementation is being undertaken mainly with Java and AspectJ. It is likely that byte-code manipulations tools (such as BCEL or Javassist) will also be used. While our preferred option is to maintain module integrity all through its life cycle, including the run-time, we are aware that in medium to long term this approach will have noticeable performance overheads. Consequently we plan to consider various optimisation strategies, e.g. guarded integration of “stable” compositions into coarse-grained slices, with only guard checked for changes, rather then the whole composition chain; or permanent integration of certain changes into module structures (at the system maintainer’s discretion) to improve performance in critical places.

Another challenging issue is that of instance adaptation, i.e. how to make objects consistent with the evolved classes. For example (going back to our example in section 2.2) how will the instances of Client class, created before composition of Questionnaire slice, handle requests to fill in questionnaire? Our present intent is to use conversion of the objects to the new definitions of their classes with a hyperslice for instance conversion handling. Thus, the instance adaptation strategy itself will be evolvable, in correspondence with the evolving schema.

4. Summary and Future Work

In the present paper we have suggested that reflection and AO can be used as complementary technologies, with reflection particularly well suited for dynamic reconfiguration and adaptation and AO as a modularisation mechanism.
We have employed the above principle in the development of the Dynamic Hyperslices model, where we use a particular AO decomposition mechanism (i.e. that suggested by the Hyperspaces approach) in combination with a reflection-based composition (via our composition connectors). The applicability of this model has been illustrated though a schema evolution scenario.
While the Dynamic Hyperslices model simplifies the change introduction and module (i.e. hyperslice) development process, it requires some consideration for the complexity of slice composition. However, the proposed model also provides for treating the composition concerns themselves as 1st class entities, similar to any other slices. Implementation and refinement of the composition mechanism is one of the prime tasks to us at the present time. Some other implementation related issues, besides those already discussed in section 3, are the development of checks for correctness of composition, consideration of ways of incorporating domain-specific knowledge into composition process.

References

[1] A. Rashid and P. Sawyer, "Object Database Evolution using Separation of Concerns," *ACM SIGMOD Record*, vol. 29, pp. 26-33, 2000.
[2] S. Clarke, W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Support for Evolution from the Design Stage," in *Workshop on Software and Organisation Co-Evolution*, 1999.
[3] A. Popovici, G. Alonso, and T. Gross, "Just In Time Aspects: Efficient Dynamic Weaving for Java ." presented at 2nd International Conference on Aspect- Oriented Software Development, Boston, USA, 2003.
[4] E. Truyen, W. Joosen, and P. Verbaeten, "Run-time Support for Aspects in Distributed System Infrastructure," in *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, 2002.
[5] J. Boner and A. Vasseur, "AspectWerkz Web Site, http://aspectwerkz.codehaus.org," 2004.
[6] P. Maes, "Concepts and Experiments in Computational Reflection," presented at OOPSLA, 1987.
[7] G. T. Sullivan, "Aspect-Oriented Programming Using Reflection and Meta-Object Protocols," *Communications of ACM*, vol. 44, pp. 95-97, 2001.
[8] R. Chitchyan, I. Sommerville, and A. Rashid, "A Model for Dynamic Hyperspaces," presented at Workshop on Software engineering Properties of Languages for Aspect Technologies: SPLAT (held with AOSD 2003), 2003.

[9] R. Chitchyan and I. Sommerville, "Composing Dynamic Hyperslices," presented at Workshop on Correctness of Model-based Software Composition (ECOOP 2003), Darmstadt, Germany, 2003.
[10] H. Ossher and P. Tarr, "Multi-Dimensional Separation of Concerns using Hyperspaces," IBM Research Report 1999.
[11] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," presented at Proc. 21st International Conference on Software Engineering (ICSE 1999), 1999.
[12] P. L. Tarr and H. Ossher, *Hyper/J user and Installation Manual*: IBM Research, 2000.
[13] L. Bergmans and M. Aksit, "Composing Crosscutting Concerns using Composition Filters," *Communications of the ACM*, vol. 44, 2001.
[14] M. Shaw, "Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status," presented at Studies of Software Design, Proceedings of a 1993 Workshop, 1996.
[15] D. Balek, "Connectors in Software Architectures (PhD Thesis)," in *Faculty of Mathematics and Physics*. Prague: Charles University, 2002.
[16] A. Rashid, "A Database Evolution Approach for Object-Oriented Databases," in *Computing Department*: Lancaster University, UK, 2000.
[17] W. Kim and H. T. Chou, "Versions of Schema for Object-Oriented Databases," presented at 14th International Conference on Very Large Databases, 1988.
[18] B. S. Lerner and A. N. Habermann, "Beyond Schema Evolution to Database Reorganisation," presented at Proceedings of ECOOP/OOPSLA, 1990.
[19] S. Monk and I. Sommerville, "Schema Evolution in OODBs Using Class Versioning," *SIGMOD Record*, vol. 22, pp. 16-22, 1993.
[20] A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in Object-Oriented Databases," presented at OOPSLA, 1986.

A Reflective Approach to Dynamic Software Evolution

Peter Ebraert[1] and Tom Tourwe[1,2]

[1] Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium
[2] Centrum voor Wiskunde en Informatica, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

Abstract. In this paper, we present a solution that allows systems to remain active while they are evolving. Our approach goes out from the principle of separated concerns and has two steps. In the first step, we have to make sure that the system's evolvable concerns are cleanly separated. We propose aspect mining and static refactorings for separating those concerns. In a second step, we allow every concern to evolve separately. We present a preliminary reflective framework that allows dynamic evolution of separate concerns.

1 Problem Statement

An intrinsic property of a successful software application is its need to evolve. In order to keep an existing application up to date, we continuously need to adapt it. Usually, evolving such an application requires it to be shut down, however, because updating it at runtime is generally not possible. In some cases, this is beyond the pale. The unavailability of critical systems, such as web services, telecommunication switches, banking systems, etc. could have unacceptable financial consequences for the companies and their position in the market.

Redundant systems [1] are currently the only solution available to solve this problem. Their main idea is to provide a critical system with a duplicate, that is able to take over all functions of the original system whenever this latter is not available. Although this solution has been proved to work, it still has some disadvantages. First of all, redundant systems require extra management concerning which software version is installed on which duplicate. Second, maintaining the redundant systems and switching between them can be hard and is often underestimated. What would happen for instance when the switching mechanism fails? Would we have to make a redundant switching mechanism and another switching mechanism for switching between the switching systems? Last, duplicate software and hardware devices should be present, which may involve severe financial issues.

The principle of separation of concerns [2] could provide an improved and more flexible solution to the problem. Applications developed with this principle in mind implement every concern in a separate entity. These entities can then be adapted and substituted without affecting the rest of the application. Depending on the programming paradigm used, an entity can be a function, an abstract data type, a class or a component, for example, or even an aspect if we employ aspect-oriented programming techniques. Whenever an application is decomposed into cleanly separated entities, its evolution boils down to the addition, the removal or the modification of such an entity. If such activities can be performed while the application is running, we call such evolution *dynamic software evolution.*

In practice, the principle of separation of concerns is not always that easy to achieve. As it turns out, no matter how well an application is decomposed into modular entities, some functionality always cross-cuts this modularisation. This phenomenon is known as the *tyranny of the dominant decomposition* [3]. As a consequence, such cross-cutting functionality (often called a *concern*) can not be evolved separately, as it affects all other entities in the application.

Although techniques exist for addressing the problem of the dominant decomposition [4–7], they should be considered too static for supporting dynamic evolution. In effect, they provide a model in which cross-cutting concerns are fixed in the application at compile time. To solve this issue, more dynamic techniques should be investigated. Several prototypes of those techniques do exist: [8, 9], but still lack some dynamic properties as well as practical experience.

2 Towards Separated Concerns

Most currently existing applications do not match with the principle of separation of concerns. This is a serious problem if we want to allow them to evolve dynamically. In order to cope with this problem, we should investigate techniques that are able to discover cross-cutting concerns in existing code, as well as techniques that are able to restructure such code so that it becomes well modularised.

2.1 Aspect mining

Research in the domain of *aspect mining* is concerned with the identification of cross-cutting concerns in existing applications. Although such research is still in the early stages, several prototype tools have already been developed that support developers in identifying cross-cutting code. Many of these tools are semi automatic, which means they require some form of input by the developer [10–12]. More advanced tools, that are able to identify aspects without human intervention, are appearing as well however [13–15]. Our aim is to study if and how these techniques can be used for our purposes. One particular shortcoming of these techniques we already identified is that they do not take into account the dynamic behaviour of the application under consideration. In order to evolve an application dynamically, it is important to know which of its parts are weakly or strongly connected, which communication patterns occur frequently, etc. This is impossible with current-day aspect mining techniques. We are thus considering extending one of them with *reflectional capabilities*, so that we can study and observe a running application and infer often recurring communication patterns.

2.2 Object- and aspect-oriented refactoring

Once the cross-cutting concerns have been identified, we need to restructure the application in order to make it well modularised according to these concerns. Refactoring techniques, as proposed by [16], allow us to modify the internal structure of an application while preserving its overall behaviour. Whenever possible, we will use these refactorings and the appropriate object-oriented features to separate concerns cleanly. However, since some concerns may then still be cross-cutting, our aim also is to investigate the use of refactorings for restructuring an ordinary object-oriented application into an aspect-oriented one. This may involve extending the already existing refactoring techniques, and defining completely new refactorings, specifically targeted toward aspects. Preliminary steps in this direction are taken by [12] and [17].

3 Towards Dynamic Software Evolution

Once we are dealing with well modularized applications – applications with no cross-cutting concerns – we want to allow every module to evolve separately. In this section we present a reflection-based framework that will permit that.

3.1 Reflective systems

A reflective system is able to reason about itself by the use of *metacomputations* – computations about computations. For permitting that, such a system is composed out of two levels: the *base level*, housing the base computations and the *metalevel*, housing the metacomputations. Both levels are said to be *causally connected*. This means that, from the base level point of view, the application has access to its representation at the metalevel and that, from the metalevel point of view, a change of the representation will affect ulterior base computations. Depending on which part of the representation is accessed, the part describing the structure of the program, or the part describing its behavior, reflection is said to be *structural* or *behavioral*.

Figure 1 illustrates the causal connection between base and metalevel, and shows how this can be used in order to change the behavior or the structure of a base-level application. The left part of the figure shows the architecture of a certain application that has cleanly separated entities at the base level. The metalevel houses a representation of this application. The application can use that in order to reason about itself (introspection). Through manipulations of that representation (introspection, the application could self-evolve. The center picture shows that a new entity is added in the metalevel representation of the application. The right picture shows the propagation of the metalevel change down to the base level, thus changing the application's behavior and structure. Using this approach we can update separated entities of a system without having to switch off the system, and thus allow dynamic evolution. Still there are several issues that have to be solved in order to do so.

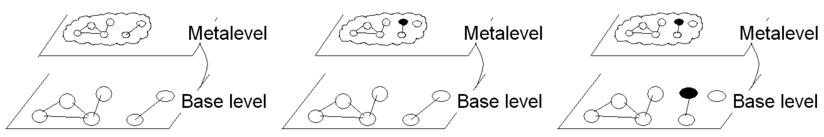


Fig. 1. Dynamically updating an entity through metalevel manipulation.

3.2 The evolution framework

We need a platform that provides both structural and behavioral reflection at runtime and that allows dynamic composition of meta-entities. As a first step, such entities will be aspects. Since we need structural reflection at runtime, we are going to experiment with Smalltalk. The behavioral reflection part will have to be added, based on the ideas of partial behavioral reflection as exposed in [18] and materialized in the Reflex platform for Java. Finally, we plan to inspire from the work on EAOP (Event-based Aspect-Oriented Programming.pdf) [9] with regards to dynamic aspect composition facilities. Although targeted to behavioral issues, Reflex and EAOP underlying ideas can be adapted to deal with structural changes. First, we definitely retain the idea of a global monitor controlling the application, and the selective introduction of hooks within base applications. As long as structural changes are intra-entity – stay locally inside a certain entity – they are straightforward to allow. If they are inter-entity changes, things will obviously get more complicated as we will have to keep track of the inter-entity dependencies. This is an issue that we will have to investigate further.

In a first version of the framework, we plan to apply a two-layered architecture to allow us to modify the behavior of a running application even when it is already running. For doing that, we instrument the running application with calls to the monitor at every point where communication between entities occurs. The monitor has to keep track of that communication in order to make it possible to substitute a certain entity. For that, it holds a representation of the application. During execution, the monitor passes control to the concerned entities (following the representation), making its presence unnoticeable. This is illustrated in figure 2. When changing a given entity, the monitor will queue all calls to the 'old' entity in order to send them to the 'new' one once in place. Our approach implies that any evolvable entity has to be referenced by the monitor, and that the monitor keeps track of entities and inter-entity relations.

3.3 The runtime API

Finally, in order to evolve the application, the user has to change the application's representation in the monitor. To that extent, a runtime API will be included so that the user can interact on-line with the monitor. The functionalities of the API have to include the addition, the removal and the modification of a system entity (aspect or functionality). Adding a new entity is done by writing its code,

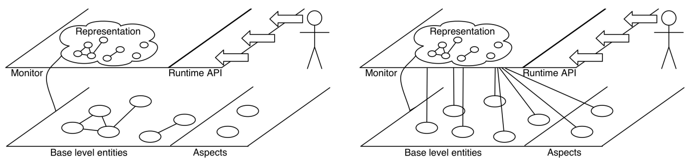


Fig. 2. Runtime evolvability by means of a two layered architecture: inter-entity communications (left) are indirected to the monitor (right).

and by registering it in the monitor. Removing an entity is more complex, as we should make sure that no other entities are dependent of that entity before actually removing it. If that is however the case, the programmer should be warned about that. When a certain entity needs to be modified, we have to write the new entities code, and tell the monitor that it should use the new entity instead of the old one whenever the old one is referenced by an other system entity. In this case, there are also some difficulties that arise, since we should be able to transfer the state from one to another entity. Some formal definition of the before-after behavior should be established in order to avoid conflicts.

4 Validation

The opening perspective of our research is that we will allow an application to evolve dynamically by allowing its composing entities to evolve dynamically. We deliberately do not restrict ourselves to one specific entity, such as a class in the Sun JVM Hotswap API, but will consider entities of any kind. As already mentioned above, such entities can thus be functions, abstract data types, classes, aspects and so on. As this is a very ambitious perspective, we will try to get as close as possible to it, by using a step by step approach.

In a first step, we will consider applications that make use of aspect-oriented technology. Such applications are typically modularized in classes and aspects. First, we will evolve the aspects of such applications. These are the easiest entities we can make evolve, as the base application does not have any reference to those entities. After that, we will focus on the evolution of class entities. This will be a harder challenge as the application has direct knowledge and references to such entities.

After that, we should widen our field of action to other programming paradigms than the object-oriented one as the ultimate goal is to employ the same approach for all existing programming paradigms. Most of the time will be spent on the representation of the application in the monitor. Once that is done, the dynamic evolution capabilities of the program will easily follow.

5 Conclusion

This paper presented a two-step solution for allowing systems to evolve dynamically. In a first step, the application's cross-cutting concerns should be removed, so that it is well modularized. We proposed aspect mining and static refactoring techniques to detect and separate the cross-cutting concerns respectively. In a second step, the well-modularized application should be controlled at the metalevel by a monitor with full reflective capabilities. Such a monitor merged the ideas of EAOP and partial behavioral reflection with the dynamic capabilities of the Smalltalk language. As such, it allows an application to evolve dynamically. Moreover, such an application can be of any programming style, object-oriented, aspect-oriented or any other, as long as it is well modularized.

References

1. O´ Connor, P.: Practical Reliability Engineering. 4th edition edn. Wiley (2002)
2. Dijkstra, E.: The structure of THE multiprogramming system. Communications of the ACM 11 (1968) 341–346
3. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 21st international conference on Software engineering, IEEE Computer Society Press (1999) 107–119
4. Ossher, H., Tarr, P.: Hyper/J: multi-dimensional separation of concerns for java. In: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland (2000) 734ñ–737
5. Szyperski, C.: Component Software : Beyond Object-Oriented Programming. Addison-Wesley (1998)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Proceedings of the European Conference on Object-Oriented Programming, Lecture Notes in Computer Science,. Volume 2072., Springer Verlag (2001) 327 – 353 http://aspectj.org.
7. Akşit, M., Tekinerdoğan, B.: Aspect-oriented programming using composition filters. In Demeyer, S., Bosch, J., eds.: Object-Oriented Technology, ECOOP'98 Workshop Reader, Springer Verlag (1998) 435
8. Popovici, A., Alonso, G., Gross, T.: Just-in-time aspects: efficient dynamic weaving for java. In: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press (2003) 100–109
9. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02). (2002) 173–188
10. Griswold, W.G., Kato, Y., Yuan, J.J.: Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, University of California, Department of Computer Science and Engineering (2000)
11. Hannemann, J., Kiczales, G.: Overcoming the prevalent decomposition in legacy code. In: Proceedings of the ICSE Workshop on Advanced Separation of Concerns, Toronto, Canada (2001)
12. Ettinger, R., Verbaere, M., de Moor, O.: Slicing Based Refactoring in Eclipse. Technical report, Oxford University Computing Laboratory (2004)

13. Breu, S., Krinke, J.: Aspect mining using dynamic analysis. In: GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik. Volume 23., Bad Honnef, Germany (2003) 21–22
14. Sheperd, D., Gibson, E., Pollock, L.: Automated mining of desirable aspects. Technical Report 4, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716 (2004)
15. Tourwé, T., Mens, K.: Mining Aspectual Views using Formal Concept Analysis. In: Submitted to Workshop on Source Code Analysis and Manipulation (SCAM). (2004)
16. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
17. Hanenberg, S., Oberschulte, C., Unland, R.: Refactoring of aspect-oriented software. In: Net. ObjectDays 2003, Erfurt, Germany. (2003)
18. Tanter, E., Noyé, J., Caromel, D., Cointe, P.: Partial behavioral reflection: Spatial and temporal selection of reification. In Crocker, R., Steele, Jr., G.L., eds.: Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003), Anaheim, California, USA, ACM Press (2003) 27–46 ACM SIGPLAN Notices, 38(11).

OASIS: Organic Aspects for System Infrastructure Software: Easing Evolution and Adaptation through Natural Decomposition

Celina Gibbs and Yvonne Coady

Department of Computer Science, University of Victoria
PO Box 3055, STN CSC, Victoria, BC, Canada V8W 3P6
University of Victoria
{celinag@uvic.ca, ycoady@cs.uvic.ca}

Abstract. It is becoming increasingly clear that we are entering a new era in systems software. As the age-old tension between structure and performance acquiesces, we can finally venture beyond monolithic systems and explore alternative modularizations better suited to evolution and adaptation. The OASIS project explores the potential of aspects to naturally shape crosscutting system concerns as they grow and change. This paper describes ongoing work to modularize evolving concerns within high-performance state-of-the-art systems software, and outlines some of the major challenges that lie ahead within this domain.

1 Introduction

With the constant demand for system change and upgrades comes the need to simplify and ensure accuracy in this process. As structural boundaries decay, non-local modifications compound the cost of system modification. Aspect-Oriented Programming (AOP) [7] aims to improve structural boundaries of concerns that are inherently crosscutting – no single hierarchical decomposition can localize both the crosscutting concern and the concerns it crosscuts. This paper explores the application and challenges of an aspect-oriented based solution to the growing problems of evolution and adaptation in system infrastructure software. We propose an approach that is incremental and organic in nature, intrinsically promoting more natural boundaries for change.

This paper begins with a high-level overview of a recently developed non-monolithic implementation of garbage collection in Java. Recent results have demonstrated that a preliminary separation of concerns into an object hierarchy does not compromise, and in fact can be augmented to improve performance relative to existing monolithic implementations [1]. The specific ways in which AOP could facilitate more cohesive system evolution when applied to domain-specific patterns is overviewed, and demonstrated by a sample aspect we have introduced to the system. We then extend these results to apply to the problem of adaptability by considering how these aspects could better facilitate change when dynamically coupled with system state and/or user demands. The precise ways in which we may be able to combine modular crosscutting with dynamism to create a more adaptable system are discussed. Though recent successes in the area of dynamic aspects provide the opportunity to immediately realize evolution of these concerns in a live system [2, 9], many challenges in extending this support to infrastructure software remain. The paper concludes with an overview of some of these challenges.

2 Background

The Jikes Research Virtual Machine (RVM) [5, 6] affords researchers the opportunity to experiment with a variety of design alternatives in virtual machine infrastructure. The project is open source and written in Java. One of the core system elements that stand to receive much attention in this testbed environment is garbage collection (GC). State of the art technologies for improving GC performance are still evolving, in particular for multiprocessor systems.

The benefits of GC are well known and have been appreciated for many years in many programming languages. GC separates memory management issues from program design, in turn increasing reliability and eliminating memory management errors. Though GC has improved significantly over the last 10 years, on going work aims to further reduce costs and meet application specific demands. Costs not only involve performance impact, but also configuration complexity. For example, in 1.4.1 JDK there are six collection strategies and over a dozen command line options for tuning GC [4]. Basic strategies, such as reference counting, mark and sweep, and copying between semi-spaces, have been augmented with hybrid strategies, such as generational collectors, that treat different areas of the heap with different collection algorithms. Collectors not only differ in the way they identify and reclaim unreachable objects but they can also significantly differ in the ways they interact with user applications and the scheduler.

3 The Dawn of Modularized, High-Performance GC

Recently, Blackburn et al. detailed the modularization of a monolithic memory management system for Jikes [1]. Their Memory Management Toolkit (MMTk) was designed with two goals in mind: flexibility, in support of f uture development, and performance, relative to a monolithic implementation and a standard C *malloc* implementation. Results demonstrate that this implementation improves modularity without sacrificing performance. In their study they measure modularity of their new MMTk by comparing metrics such as lines of code, lack of cohesion of methods (LCOM) and cyclomatic complexity with two earlier versions of the RVM. The results show MMTk as the clear winner in all of these comparisons. For example, cyclomatic complexity is a measure of branching and looping complexity within a method. MMTk displays an average ranging from 1.5 to 5 times lower in this area than the previous RVM memory management versions.

The performance tests include a comparison of allocation and tracing rates along with raw speed comparison. The tests run on standard benchmarks reveal MMTk has better performance overall than its monolithic counterpart. Though one comparison shows C outperforming MMTk by 6%, compiler inlining in the MMTk version realizes substantial performance gains up to 60% over the C implementation.

4 OASIS: Shaping Fertile Code

We have just embarked on the OASIS project, which will take the refactoring of this system one step further, modularizing crosscutting concerns within MMTk. We believe aspects will not only serve to increase the flexibility of the system, but will further facilitate system evolution and research. MMTk can now be used as a structured benchmark to measure performance and modularity of an AOP implementation against. By using the same techniques used in the comparison of MMTk to its monolithic counterpart (LCOM, cyclomatic complexity, etc), the level of modularization and performance impact incurred by aspects can be meaningfully quantifiably assessed.

In addition to improving on these metrics, added benefits of an aspect-oriented implementation are the semantic and practical value of the explicit relationship between crosscutting concerns and the concerns they crosscut. Though the internal structure creates a clearer picture of the system for developers to work with when researching new VM techniques, this factoring-out of crosscutting concerns also provides developers with the ability to plug/unplug more system elements as configuration options. Eventually employing dynamic (runtime) aspects would further allow the system to switch GC strategies on-the-fly, at run-time, enabling more effective attempts at feedback-based, system-wide optimization. This presents the possibility of a system that could dynamically switch collectors dependant on system state and/or user input. The following section overviews some of the domain-specific design patterns we have started to shape as aspects within MMTk.

5 Domain-Specific Design Patterns: Shaping Aspects of GC

Blackburn et al. use design patterns in the development of MMTk for reuse and efficiency. Four domain specific design patterns are detailed in [1]. For reuse, a pattern for *prepare and release phases* exploits collection phases to simplify the development of new collectors, and a pattern for *multiplexed delegation* passes on collector tasks to appropriate memory management policies. For efficiency, a pattern for *hot and cold paths* optimizes the common path, and a pattern for *local and global scope* minimizes contention and synchronization between multiple collector threads in multiprocessor systems.

The composition of collectors is broken down into three elements: mechanisms, policies and plans. Mechanisms are collector-neutral and shared among collectors. Policies manage contiguous regions of virtual memory, and are expressed in terms of mechanisms. Plans define the ways in which collectors are composed, as described in terms of policies.

At a top level, the algorithm used by collectors in MMTk integrates the domain-specific patterns previously introduced. This is evident by looking at the control flow through the stop-the-world superclass, extended by all plans that suspend the system while GC takes place. The implementation has three phases: *prepare*, *processAllWork*, and *release*. The prepare/release phases are further specialized by the multiplexed delegation pattern, divided according to the global and local context pattern, and optimized by hot and cold paths. For example, given the simple case of a single collector thread on a uniprocessor, the structure of local and global scopes within prepare and release phases can be captured by the simple finite state machine in Figure 1.

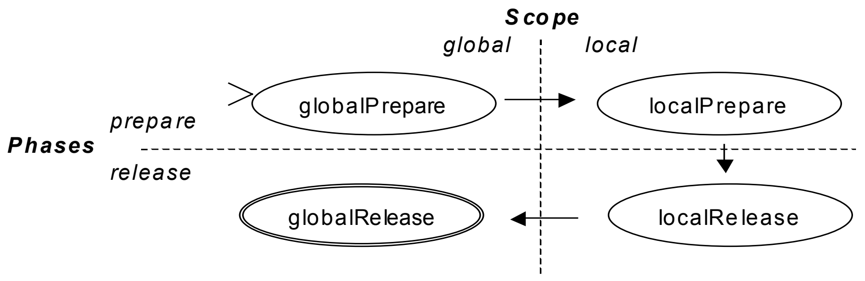


Figure 1 – Structure of Local and Global Scopes within Prepare and Release Phases.

All eight plans in the Jikes RVM adhere to this overall structure. A motivating force behind MMTk was to leverage these patterns to ease the evolution of new plans. For example, division into local and global contexts in multiprocessor environments separates operations that need to be synchronized (global) from those that do not (local). Though there is always a single global state, threads on different processors can manipulate local state concurrently. This division needs to scale well across processors, and may someday be extended to include features such as dynamic join/leave of local participants.

Our experiment to date focuses on the structure of this simple FSM. Using a standard configuration for Jikes, we refactored policy related activities associated with phases and scope as an aspect within one plan. We argue that, in this form, the internal structure of each plan's use of policy becomes unpluggable and clear. It is unpluggable in this form because different combinations of policy can be specified at compile time, and clear because the internal structure associated phases, scope, and delegation can be coalesced in isolation from the rest of a given plan. By targeting these patterns, aspects can shape system elements identified to be critical during evolution.

6 Implementation Details

Figure 2 highlights three of the eight plans in Jikes (*copyMS*, *genRC* and *refCount*) and some of the contents of their four of the key abstract methods in the `StopTheWorldGC` class where plans interact with policies: *globalPrepare()*, *threadLocalPrepare()*, *threadLocalRelease()* and *globalRelease()*. In this form, similarities and differences between plans with respect to memory management policy can be more easily assessed. Some plans such as *genRC* and *refCount* in the right half of Figure 2, share the same combination of policies.

When plan is *copyMS*

and on **globalPrepare** path
copySpace prepare
immortalSpace prepare
markSweepSpace prepare
treadmillSpace prepare

and on **threadLocalPrepare** path
markSweepLocal prepare(*markSweepSpace)*
treadmillLocal prepare(*treadmillSpace)*

and on **threadLocalRelease** path
markSweepLocal release(*markSweepSpace)*
treadmillLocal release(*treadmillSpace*)

and on **globalRelease** path
immortalSpace release
markSweepSpace release
treadmillSpace release

When plan is *genRC* or *refCount*

and on **globalPrepare** path
immortalSpace prepare
refCount prepare

and on **threadLocalPrepare** path
refCountLocal prepare(*refCountSpace*)

and on **threadLocalRelease** path
refCountLocal prepare(*refCountSpace*)

and on **globalRelease** path
immortalSpace prepare
refCount prepare

Figure 2 – Combinations of Policies in *copyMS, genRC* and *refCount*

Refactoring this code to re-introduce policy to plan using aspects better exposes both the symmetry between prepare/release phases and its interaction with the global/local contexts of the protocol. That is, it better captures the internal structure of policy within a given plan. Figure 3 shows the implementation of the simple FSM model for one plan, *copyMS*, using AspectJ [8].

```
package com.ibm.JikesRVM.memoryManagers.JMTk;

privileged aspect PolicyAspect {

    private final int GLOBAL_PREPARE = 0;
    private final int LOCAL_PREPARE = 1;
    private final int LOCAL_RELEASE = 2;
    private final int GLOBAL_RELEASE = 3;
    private int state = GLOBAL_PREPARE;


    after(Plan p):target(p) && (execution(* Plan.globalPrepare(..))
                || execution(* Plan.threadLocalPrepare(..))
                || execution(* Plan.threadLocalRelease(..))
                || execution(* Plan.globalRelease(..))) {
      switch(state){
          case(GLOBAL_PREPARE):
               CopySpace.prepare();
               Plan.msSpace.prepare();
               ImmortalSpace.prepare();
               Plan.losSpace.prepare();
               state++;
               break;
          case(LOCAL_PREPARE):
               p.ms.prepare();
               p.los.prepare();
               state++;
               break;
          case(LOCAL_RELEASE):
               p.ms.release();
               p.los.release();
               state++;
               break;
          case(GLOBAL_RELEASE):
               Plan.losSpace.release();
               Plan.msSpace.release();
               ImmortalSpace.release();
               state = GLOBAL_PREPARE;
               break;
      }
    }
}
```

Figure 3 – PolicyAspect.java

7 Adaptability: Could we do this on-the-fly?

One problem however, for this experiment and those involving other low-level software systems, is that run-time support needs to be highly generic to work with multiple different thread or process models. That is, support for language mechanisms such as that associated with control flow, or *cflow,* needs to be independent from any one definition of thread state in order to work with (and within) a range of JVMs. Techniques that exploit meta-data might be effectively applied to inspect system state and adapt the system accordingly. We believe extensive new avenues of research in VM technologies would result if recent successes in dynamic aspects, such as those presented in [2, 9], could be extended to apply to system infrastructure code as well as to the application domain [3].

8 Conclusions

Modern systems software need not be monolithic to achieve performance. As this tension between structure and performance subsides, alternative modularizations and means of extracting system state can be used to more effectively facilitate evolution and adaptation. OASIS explores the potential of aspects to naturally shape crosscutting concerns as they grow and change within system infrastructure software. Though the project is in its formative stages, preliminary results targeting domain-specific patterns have started to uncover the potential advantages and challenges within the application of these techniques to the evolution of performance sensitive object-oriented software systems.

References

1. Blackburn, S. M., Cheng, P., and McKinley, K. S. Oil and Water? High Performance Garbage Collection in Java with MMTk *ICSE 2004, 26th International Conference on Software Engineering*, (Edinburgh, Scotland, May 23-28, 2004)*(to appear)*

2. Christoph Bockisch, Michael Haupt, Mira Mezini, Klaus Ostermann, Virtual Machine Support for Dynamic JoinPoints, Proceedings of the International Conference on Aspect-Oriented Software Development 2004.

3. Celina Gibbs and Yvonne Coady, *Garbage Collection in Jikes: Could Dynamic Aspects Add Value?*, Dynamic Aspect Workshop, held at the International Conference on Aspect-Oriented Software Development 2004.

4. Brian Goetz. How does garbage collection work? Developerworks, October 2003. www-106.ibm.com/developerworks/java/library/j-jtp10283/

5. Jikes Research Virtual Machine, IBM. www-124.ibm.com/developerworks/oss/jikesrvm/

6. Jikes Research Virtual Machine User's Guide, IBM. www-124.ibm.com/developerworks/oss/jikesrvm/userguide/HTML/userguide.html

7. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP), 1997.

8. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, An overview of AspectJ, Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP), 2001.

9. Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori, *A Selective, Just-In-Time Aspect Weaver*. Proceedings of the second international conference on Generative programming and component engineering, pp 189 – 208, 2003.

Negligent Class Loaders for Software Evolution

Yoshiki Sato and Shigeru Chiba

Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
and Japan Science and Technology Corp
{yoshiki,chiba}@csg.is.titech.ac.jp

1 Introduction

Modern software should be dynamically evolvable. The onward sweep of technology, introducing new features, and fixing immortal security holes, drive this evolution. Moreover, practical demands of on-the-fly changes of running applications eagerly encourage the evolution since the downtime of commercial software directly causes big losses. Enabling the evolution of software components is a major focus of efforts of the designers of imperative, static typing, and high-performance languages, such as C++ and Java.

Recently, the ability of Java's class loaders, which play a central role in runtime flexibility of Java, has been considered to be insufficient for software evolution. Java class loaders do not allow reloading classes that have been loaded since to allow reloading a class tends to make several crucial runtime optimizations difficult [6]. Instead, Java class loaders provide a mechanism for using different versions of a class at the same time although the use of this mechanism is restricted. This restriction is called the version barrier.

This paper presents a negligent class loader, which can relax the version barrier between class loaders for software evolution. The version barrier is a mechanism that prevents an object of a version of a class from being assigned to a variable of another version of that class. In Java, if a class definition (i.e. class file) is loaded by different class loaders, different versions of the class are created and regarded as distinct types. If two class definitions with the same class name are loaded by different loaders, two versions of the class are created and they can coexist while they are regarded as distinct types. The version barrier is a mechanism for guaranteeing that different versions of a class are different types. Regarding two versions as distinct types is significant for performance reasons. If not, advantages of being a statically typed language would be lost.

2 Motivations

First, this section shows some practical examples that bring to us inconvenience of the current Java class loaders.

2.1 Dynamic aspect weaver

Dynamic AOP (Aspect-Oriented Programming) is receiving interests growing in both the academia and the industry. Unlike static AOP systems, dynamic AOP systems allow dynamically weaving and unweaving an aspect into/from a program. Moreover, advice and pointcuts can be changed during runtime. These dynamic features extend the application domain of AOP. Dynamic AOP can shorten the lead time of the edit-deploy-run cycle of software development. It can also allow using aspects for making the behavior of application software adaptable to changes of the runtime environment and requirements.

The version barrier makes it difficult to implement an instance-based dynamic AOP system. Such a dynamic AOP system allows weaving a different aspect with a particular instance of a class. A simple implementation of such an AOP system would use multiple class loaders, each of which loads a different version of a class woven with a different aspect. However, this implementation approach does not work because instances of those versions are not compatible because of the version barrier (Figure 2.1). Therefore, most of dynamic AOP systems adopt complicated implementation techniques such as static code translation [8] [1], just-in-time hook insertion [11], and modified JVM [9] [2] although these techniques imply certain performance penalties.

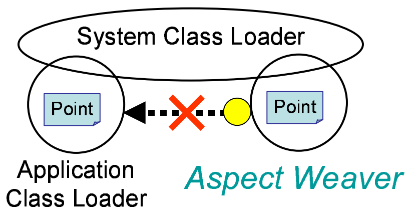


Fig. 1. Aspects can not be woven into loaded classes.

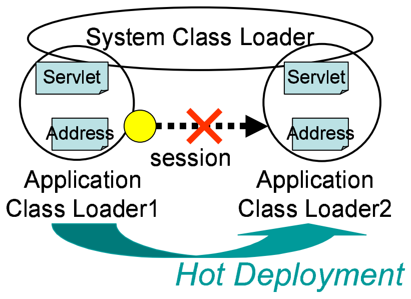


Fig. 2. Hot deployment discards all internal states.

2.2 Hot deployment for application servers

Most of application servers including both commercial (Websphere, Weblogic, etc.) and open-source (JBoss, Tomcat, etc.) provide the hot deployment functionality, which enables software components (EJB-JAR, EAR, or WAR package) to be plugged and unplugged without restarting application servers. This dramatically improves the productivity of software development. To be separately deployed and undeployed at runtime, each component is loaded by a distinct class loader. Thus, a J2EE application can be dynamically customized per component.

However, if a new version of a component is deployed for software evolution, instances of the new version cannot be exchanged for instances of the old version of that component because of the version barrier (Figure 2.1). Such instances are caches, cookies, and session objects and session beans. These instances should be passed to the new component through the shared container of the application server. This is remarkably inconvenient in practice. That is to say, J2EE application servers provide hot deployment but not “hot evolution”.

The version barrier makes it difficult to include a copy of a shared class in every component. If two components share a class, for example, for exchanging data, each component should be able to include a copy of that shared class since an ideal component should contain all the class files that are necessary for running an application. However, the version barrier disables exchanging an instance of that shared class between the two components. To avoid this problem, JBoss provides the UCL (Unified Class Loader) architecture [7] but this architecture prevents different components from including different classes with the same name.

3 Negligent class loaders

We propose a *negligent class loader (NCL)*. The NCL can relax the version barrier if an incoming object is an instance of a class loaded by a sibling class loader specified by the programmer in advance. Here, a sibling means a class loader sharing the same parent loader. For example, a NCL allows assigning an instance of class Customer that has been loaded by a sibling to a variable of Customer loaded by the NCL. To keep consistency, differences between the two versions of Customer must satisfy the rules described below.

Naively relaxing the version barrier may cause serious security problems. For example, a program may access a non-existing method and then crash the JVM. In fact, the version barrier of Sun JDK 1.1 was accidentally relaxed and thus it had a security hole known as the type-spoofing problem first reported by Saraswat [10]. This security hole was solved by the loader constraint scheme [5], which strengthens the version barrier. To avoid this security problem, runtime type checking is necessary. For example, dynamically typed languages such as CLOS and Smalltalk do not provide the version barrier. Since a variable is not statically typed, any type of instance can be assigned to a variable. For security, these languages perform runtime type checking so that, if a non-existing method or field is accessed, an exception will be thrown at runtime. A drawback of this approach is that it requires frequent runtime type checking, which implies non-negligible performance degradation.

The NCL restricts the relaxation of the version barrier only to the classes loaded by a sibling class loader so that runtime overheads due to the relaxation will be reduced. Suppose that a sibling class loader loads a class S and then the NCL loads a class N. The NCL allows an instance of S to be assigned to a variable of the type N if the following condition is satisfied:

1. The class S includes all the members such as methods and fields of the class N. The method bodies can be different between N and S.
2. Or, the class N includes all the members of the class S. In this case, a new class S’ is dynamically generated and a reference in instances of S is changed to indicate

the internal type information block representing S’ instead of S. S’ is a copy of S but it also includes the methods and constructors included in N but not in S. These methods in S’ are called secure handling functions. If they are accessed, they throw a runtime exception representing illegal access. This is for avoiding a serious security hole that can be used for buffer overflow attacks. Thereby the secure execution is guaranteed with respect to N and S as in dynamically typed languages. Without that substitution of S’ for S, a program written for N may violate the boundary of the method table in S.

In both the cases, the class N must have the same super class as the class S and it must not weaken access restriction against the class S. For example, if a method is public in S, then the method in N must be also public.

The type information block (TIB) referred to by an instance contains a method table that holds function pointers to a corresponding method body. For secure execution, the NCL creates the TIB of the class N to be compatible as much as possible with that of the class S when the class N is loaded. In particular, the order of the method table entries in the TIB of the class N must be the same as the order in the TIB of the class S with respect to the methods that both the classes S and N include. Otherwise, for example, a program written for N might invoke a wrong method body if the targt instance is of S. The algorithm for constructing the TIB is shown in Figure 3.

In our architecuture, runtime type checks are performed only when the `checkcast` bytecode instruction is executed. The `checkcast` instruction examines the version of an instance and, if it does not satisfy the condition above, it throws the `ClassCastException`. To do this, we modify the JVM. Other instructions such as `invokevirtual`, `getfield` and assignment instructions like `astore`, do not have to examine the condition. This is because the NCL relaxes the version barrier only for sibling class loaders, which satisfy the bridge-safety property [10]. If this property is satisfied between two class loaders, instances of a class `C` loaded by one class loader are always upcast to a class loaded by their parent class loader before the instances are passed to a class loaded by the other class loader. For example, the instances will be upcast to a super class or an interface of the class C, such as the `Object` class, which is loaded by the parent class loader. Thus explicit downcast must be executed before those instances are assigned to a variable of a type loaded by the NCL (Figure 3).

4 Related Work

There are a number of research activities for software evolution. Liang and Bracha [5] described a programming technique of using an interface type as the type of a variable that can refer to instances of multiple versions of a class. However, this technique requires programmers to define an interface type for every multi-versioned class and access instances of the class through the interface type. Hjalmtysson and Gray [3] implemented dynamic classes in C++ by using templates. Their system uses wrapper (or proxy) classes and methods. This approach does not require runtime system support or language extensions. However, it implies performance penalties. Malabarba *et al.* [6] modified the JVM to make a class reloadable at runtime. However, it also implies runtime penalties because of difficulties in performing runtime optimization techniques,

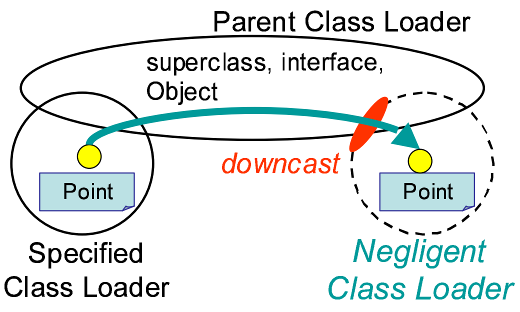


Fig. 3. Downcast enforced by the bridge safety between the NCL and a sibling class loader.

```
if S has been already loaded && S's loader is a sibling of N's one then
    for each member m in S {
        if N includes the member m then
            Push the member m into the type information block of N
        else
            Push a secure handling function into the type information block of N
        end
    }
end
```

Fig. 4. Pseudo code for constructing the internal type information block.

such as method inlining and quick bytecode instructions. The Java2 SDK1.5 will support the hot swap mechanism with the `java.lang.instrument` package. Although it enables reloading a class to a certain degree, a new version of a class does not include a method or field that was not included in the previous version.

5 Current State

We are currently implementing a negligible class loader on the IBM Jikes Research Virtual Machine [4]. We will use this implementation to evaluate the performance overheads of our approach. In addition, we will study the proof of the type safety of our system, and reason about our approach with respect to the Java security architecture. Furthermore, we have not considered how to correctly handle arrays of multi-versioned class types.

References

1. Baker, J., Hsieh, W.: Runtime Aspect Weaving Through Metaprogramming. In: 1st International Conference on Aspect-Oriented Software Development. (2002) 86–95

2. Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual Machine Support for Dynamic Join Points. In: International Conference on Aspect-Oriented Software Development. (2004)
3. Hjálmtyśson, G., Gray, R.: Dynamic C++ Classes: A lightweight mechanism to update code in a running program. In: In Proceedings of the USENIX Annual Technical Conference, New Orleans, Louisiana, USENIX (1998)
4. IBM: Jikes Research Virtual Machine. (2001)
5. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: Proceedings of OOPSLA'98, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications. Number 10 in SIGPLAN Notices, vol.33, Vancouver, British Columbia, Canada, ACM (1998) 36–44
6. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime Support for Type-Safe Dynamic Java Classes. In: ECOOP 2000 - Object-Oriented Programming, 14th European Conference. Volume 1850 of Lecture Notes in Computer Science., Springer (2000) 337–361
7. Marc Fleury, F.R.: The JBoss Extensible Server. In: ACM/IFIP/USENIX International Middleware Conference. Volume 2672 of Lecture Notes in Computer Science., Rio de Janeiro, Brazil, Springer (2003) 344–373
8. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible framework for AOP in Java. In: Reflection 2001. (2001) 1–24
9. Popovici, A., Alonso, G., Gross, T.: Just in Time Aspects: Efficient Dynamic Weaving for Java. In: 2nd International Conference on Aspect-Oriented Software Development. (2003) 100–109
10. Saraswat, V.: Java is not type-safe. (1997)
11. Sato, Y., Chiba, S., Tatsubori, M.: A Selective, Just-In-Time Aspect Weaver. In: Second International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt Germany (2003) 189–208

Part (III):
Join Points and Crosscutting Concerns for Software Evolution

Chairman: Günter Kniesel, University of Bonn, Germany.

Components, ADL & AOP: Towards a common approach

Nicolas Pessemier[1], Lionel Seinturier[12], and Laurence Duchien[1]

[1] INRIA Futurs, USTL-LIFL, Team GOAL/Jacquard, Villeneuve d'Ascq, France
{pessemier, seinturi, duchien}@lifl.fr
[2] Univ. Paris 6, Lab. LIP6, Team SRC, Paris, France
Lionel.Seinturier@lip6.fr

Abstract. Components, architecture description languages and aspect-oriented programming are recognized as new trends in the software engineering of modern applications. They all try to bring solutions for building more easily complex applications. This paper reports on our position that aspect-oriented programming can be used both at the component and the architecture description level to describe the evolution of software. This position is validated by a prototype implementation that extend the existing Fractal component model with some dynamic aspect-oriented programming features.

Keywords: separation of concerns, ADL, component, AOP, Fractal

1 Introduction

Ever since the beginning of computer science, programming language designers have tried to find more and more abstract software artifacts. The goal is to provide programmers with powerful and safe structures to implement their solutions. Approaches such as architecture description languages (ADL) [5], component-based programming (CBP) [11][10] and aspect-oriented programming (AOP) [3] all go towards that direction. By reifying software assemblies, ADLs provide a clear view of the "software map" of the application. CBP promotes modularity and composability by encapsulating units of code into a capsule that can be deployed, configured, and used through some clearly identified interfaces. Finally AOP provides a clear way for modularizing crosscutting concerns, i.e. functionalities that, with object-oriented programming (OOP) or CBP can not be cleanly localized in a single unit of code and crosscut several objects or components.

In this paper, we argue that each of the three approaches ADL, CBP and AOP, address the issue of software evolution. ADL clarifies the way software entities interact. CBP packages software entities in modules that can be more easily reused than objects. AOP removes from objects functionalities that are out of the scope of their primary concern. Our point of view is that ADL, CBP and AOP, extend object-oriented programming in different ways. Far from being conflicting, these ways are complementary. In this paper, we present our first experiment towards a framework that integrates concepts taken from these three domains. Basically, this framework is based on a component model and provides an ADL to describe software architectures with crosscutting concerns.

The paper is organized as follows. Section 2 reviews the existing project, Fractal, on which we based our approach. Sections 3 and 4 present the extensions we introduce in Fractal to obtain a framework that supports the concepts of ADL, CBP and AOP. Section 5 presents some related works. Section 6 concludes this paper and provides our directions for future works.

2 Background

Building a framework that merges the concepts of ADL, CBP and AOP, is a challenge where many choices are to be made. We can design and implement a whole framework from scratch, or we can rely on some existing works. Section 5 reviews some of them. In order to obtain a first working prototype, we decided to start from the Fractal framework [1] and to extend it. Fractal is in our opinion, the project that is closest to the requirements stated in the previous section. The remaining of this section presents Fractal, and the next two sections introduce the way we extended it to support crosscutting concerns.

2.1 The Fractal Component Model & ADL

Many component models such as EJB, .Net or CCM exist and receive much interest from both the academia and the industry. These models are however mainly dedicated to coarse grained components for information system-like applications. Classes implementing these components must enforce programming rules, they must be bundled with XML descriptors, and they need to be executed by application servers. They can not thus be handled as easily as objects are handled by virtual machines. Thus, despite of their wide adoption by the community, there is a need for a lighter component model, closer to programming language concepts and that do not require the extra-machinery of the above-mentioned models. The Fractal component model [1] meets these needs. Further, it comes with an XML based ADL.

The Fractal component model [1] allows the definition, configuration, dynamic reconfiguration and clear separation of functional and non functional concerns. Built as a high level model, it put the stress on modularity and extensibility. The Fractal component model is recursive in the sense that a component may be primitive, or composite. In the latter case, this is an assembly that content other primitive or composite components. Components may also be shared between different composites.

Interfaces play a central role with Fractal. There are two categories of interfaces: business and control. Business interfaces are external access points to components. Fractal provides client and server interfaces; a server interface receive operation invocations and a client interface emits operation invocations.

Thus, a Fractal binding represents a connection between two components (primitive binding) or more (composite binding). The Fractal model is a strongly typed model. As a consequence, the type of a server interface must be a sub type of the type of a client interface. As their name suggests it, control interfaces provide a level of control on the component they are attached to. These interfaces are in charge of some non-functional properties of the component, for instance its life cycle management, or the management of its bindings with other components.

2.2 Programming with Fractal

This section illustrates with an example the concepts of the Fractal component model. The example in figure 1 modelizes a gas station. Each rectangle is a component. Clients use a gun to fill their tank, and pay at a cash register that is connected to a bank. Each of these elements is a primitive Fractal component. There are two composite components: one for the station and one for the whole system. Composite components are assemblies of primitive and/or (other) composite components. The T-s attached to components are Fractal interfaces[1]. Arrows are bindings between components: they go from a client interface to a server interface.

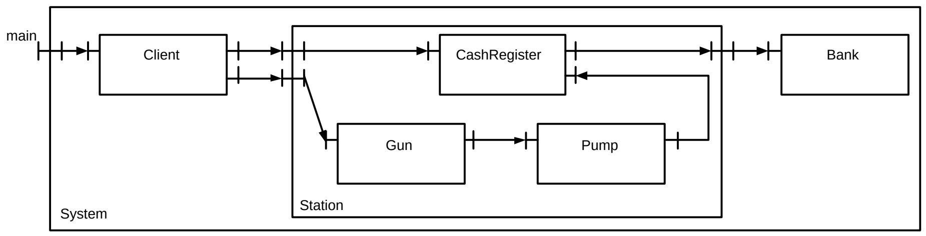


Fig. 1. A gas station with Fractal components.

Fractal provides a Java API to create, introspect and manage the components, their interfaces and their bindings. For instance, components can be started and stopped, and bindings can be created and removed dynamically. The structural definition of components is provided with an XML ADL. Figure 2 provides a piece of the architecture definition for the gas station.

3 Rationale for our Project

The software architecture presented in the previous section reifies business dependencies between components. The bindings that have been identified come

[1] Only business interfaces are specified in the example. Fractal provides the concept of control interfaces that provide a level of control on the component there are attached to.

```
<!-- A composite component definition -->
<component name="station">

   <!-- The Station component provides (server role) or
        requires (client role) interfaces -->
   <interface name="gunProvideGas" role="server" signature="station.GunProvideGas"/>
   <interface name="bankAuth" role="client" signature="station.BankAuth"/>
   <!-- other interfaces -->

   <!-- The station component contains the pump, gun & cashRegister components -->
   <component name="pump">
      <!-- ... -->
   </component>

   <!-- The Station component interfaces are bound to other interfaces
   <binding client="this.cashRegisterUserInterface"
            server="cashRegister.cashRegisterUserInterface" />
   <binding client="this.gunProvideGas" server="gun.gunProvideGas" />
   <!-- other bindings -->
</component>
<!-- .... -->
```

Fig. 2. The software architecture of the gas station with the Fractal ADL.

from the analysis of the business logic of the application. Based on this logic, some crosscutting domains may arise when the application evolves, either because some new unforeseen requirements emerge, or because all the concerns could not have been addressed at a first stage. For instance, a security domain may be needed between the CashRegister and the Bank components. This domain crosscuts the functional domains that have been identified by the business analysis (figure 3 illustrates this).

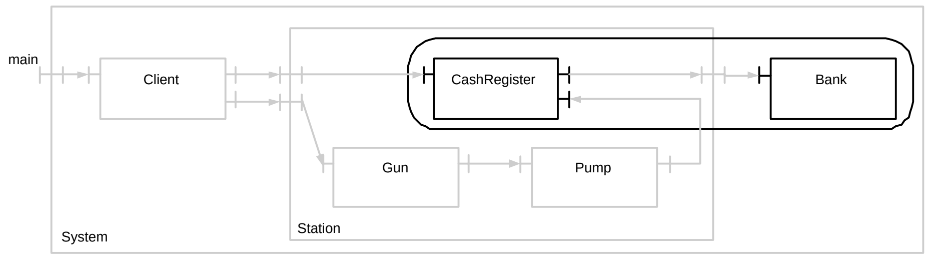


Fig. 3. Crosscutting security concern in the gas station example.

Whereas business assemblies and domains are clearly reified in the architecture description of the Fractal ADL, this is not the case with the crosscutting domains: no artifact exists in the Fractal model and ADL to express them. This is true for Fractal, but this is also true as far as we know, for any other ADL or component model.

Hence, the purpose of our project is to extend the existing Fractal model and ADL to take into account crosscutting domains.

4 Extending Fractal to support crosscutting concerns

The extension that we have implemented let us superimpose on an initial business assembly, a level of assembly that corresponds to a crosscutting domain. Hence some new bindings need to be set up. For instance, we want to redirect all outgoing calls from the CashRegister component to perform some encryption functions, and we want to redirect all incoming calls to the Bank component to perform a symmetric decryption function (figure 4 illustrates this).

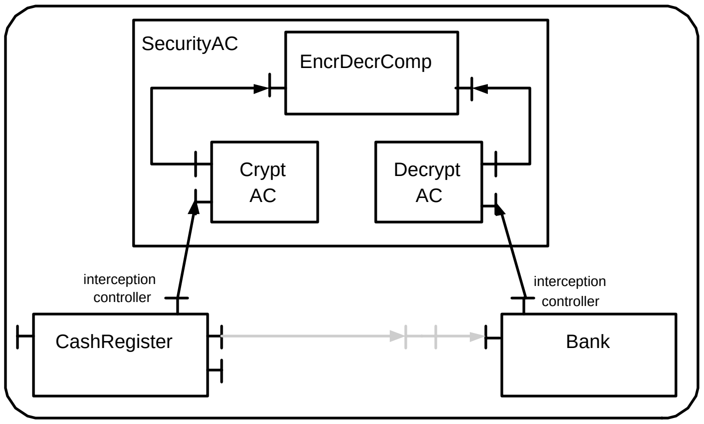


Fig. 4. Aspect component for the security concern.

At this point, we need two elements to implement our extension: we need the notion of a component that localizes the definition of the crosscutting concern, and we need a mechanism that weaves the crosscutting concern on a software architecture. Both elements are described below.

4.1 Aspect Components

An Aspect Components (AC)[2] localizes the definition of a crosscutting concern. This is the case of SecurityAC in figure 4. This AC is composed of 3

[2] The term Aspect Component comes from our previous project, JAC [6], which is a framework for dynamic aspect-oriented programming in Java.

sub-components: 2 of them, CryptAC and DecryptAC, implement the interception logic associated with the security concern, and the 3rd one, EncrDecrComp, provides the mathematical functions that are required to crypt and decrypt messages. In AspectJ terms, we could say that CryptAC and DecryptAC implement pointcut descriptors, whereas EncrDecrComp implements two pieces of advice.

The CryptAC and DecryptAC components are method interceptors. CryptAC intercepts method calls and DecryptAC intercepts method executions. Their server interface conforms to the AOP Alliance API[3]. They implement the `MethodInterceptor` interface that defines the following method.

```
public Object invoke( MethodInvocation mi ) throws Throwable;
```

The `invoke` method provides the code that must be run before and after the joinpoint. `MethodInvocation` provides a `proceed` method to execute the joinpoint. To perform the interception, both CryptAC and DecryptAC rely on an interception controller provided by the CashRegister and Bank components. This controller ensures that the outgoing and incoming calls can be reified.

To sum up our approach, a crosscutting concern is implemented with a Fractal component (composite like in our example, or simply primitive) called an aspect component (AC), that provides at least one `MethodInterceptor` interface. No other requirement is needed. The remainder of this section focuses on the way ACs can be woven on top of a software architecture.

4.2 Crosscut bindings

In our approach, weaving an AC is very similar to binding two components together (except that one of them is an AC, and that the other must provide an interceptor controller). There are two ways to establish such a binding, that we call a crosscut binding: either directly, or declaratively with a pointcut expression.

Direct crosscut binding. Starting from the references of the component to be aspectized and of the AC, a crosscut binding is directly created between the two. All methods of the component will then be intercepted by the AC. This binding is similar to establishing a meta-link relation between a base component and a meta-component.

Crosscut binding with a pointcut expression. In this case, a pointcut expression is required. It is composed of three regular expressions that operate on component names, interface names, and method names. All methods that match the three expressions are aspectized by the AC. This weaving is performed by calling a `weaveAC` method on a root business component. This method recursively traverses the hierarchy of sub-components of this root component, and finds all

[3] AOP Alliance `<http://sourceforge.net/projects/aopalliance>` is an open-source initiative to define a common API for AOP framework. The API is implemented by Spring and JAC, and soon by DynAOP.

the methods that match the pointcut expression. The technique is similar to the weaving performed by aspect compilers and frameworks.

Note that in both cases, bindings established either directly or with a pointcut expression, can later on be manipulated dynamically: they can be unbound or rebound to modify the crosscut policy of the AC. Hence, they are quite similar to bindings between business components.

5 Related Works

Some recent approaches tried to combine CBP and AOP. This section provides a quick review of some of them.

A first approach that combines CBP and AOP is JAsCo [9] that extends the Java bean model by introducing the notions of *aspect beans* and *connectors*. The *Aspect bean* describes what and where (the notions of *advice* and *pointcut* in AOP) to apply a context independant behavior, using a kind of inner class called the *hook*. The *connectors* are in charge of deploying *hooks* in a specific context of application. A great contribution of JAsCo is provided by its connector language that allows to define a more fine-grained control than AspectJ, on the order the aspects are executed. Finally, we can notice that the management of hooks and connectors is completely centralized and handled by a *connector registry* that have been recently enhanced through HotSwap and Jutta [8] in order to reduce the cost that every dynamic AOP approaches suffers from.

DAOP [7] is a dynamic distributed platform where aspects and components are first-order entities that are composed at runtime by a *middleware layer*. As JAsCo approach, the aspect and component management is centralized and all the information about the architecture and its entities is stored in the *middleware layer*. In a first phase, aspects and components are described with a specific *aspect-component language* that allows interfaces, roles and binding definitions. Then, at runtime components and aspects are concretely bound following the *middleware layer* specifications. The original contribution of the DAOP approach is to give a unique role name to every component and aspect. By this way, communications are done by giving these role names and not by object references.

Jiazzy [4] is an enhancement of the Java language for large scale binary components that are separately compiled and externally linked (*units*). Units are kind of pre-compiled Java classes container that are of two different types: *atoms* (construct from Java classes) or *compounds* (construct from other *atoms* or *compounds*). New behaviors can be added to methods or fields without editing the source code, thanks to a mechanism of *open classes* and *open signature* that are based on mixins. To sum up, Jiazzy separate concerns at the granularity of classes and offers some behavior enhancement with a mechanism of mixins that is less powerful than the AspectJ approach.

JBoss AOP [2] is a project that provides AOP capabilities to EJB applications. JBoss AOP allows to modify an application with aspects, to introduce new features in an application with a mixin mechanism, and to manage some metadata. Advices are programmed as Java classes implementing an interception API. Pointcuts are defined in XML and associate interceptors with the application. The weaving is dynamic with JBoss AOP.

6 Conclusion

ADL, component models and AOP are all concerned about software evolution. Their goal is to empower developers to give them the possibility to build complex systems. Our position in this paper is that the three approaches are complementary. However, as far as we know, they have never been put together into one unified framework. This paper proposes a first step into that direction. We have extended an existing component model and ADL, Fractal (see section 2) with some AOP support for crosscutting concerns. We are then able to describe complex component-based software architecture with aspects.

Aspects in our proposal are components called aspect components (AC), that implement a special meta-interface. Weaving an aspect is then a matter of establishing bindings between business components and ACs. There are two ways for establishing such a binding: either directly between a business component and an AC, or by recursively traversing a hierarchy of composite components and finding components that match a given pointcut expression. These two ways unify the meta-link relation of reflective programming and the weaving process of aspect-oriented programming.

This study served as a proof of concept that merging concepts from ADL, CBP and AOP can lead to a working prototype. However, many features remain to be implemented to obtain a more complete development environment: the pointcut definition language must be enhanced, some API must be defined to support crosscut introspection and some GUI tools are needed to assist developers in specifying their crosscutting assemblies. The integration between ADL, CBP and AOP has been conducted by mapping the concepts of AOP onto the ones of ADL and CBP. It remains to be seen whether the reverse is also true. Finally, as with other aspect languages or frameworks, the weaving of an aspect onto a business application is based on the pointcut expressions. The information contained in these expressions is rather light, and in all cases only concerns the structure on the underlying application. Behavioral pointcut expressions could certainly lead to a more powerful aspect programming environment.

References

1. Bruneton, E., Coupaye, T., and Stefani, J. Recursive and dynamic software composition with sharing. In Workshop on Component-Oriented Programming (WCOP) at ECOOP'02 (June 2002). `http://fractal.objectweb.org/current/fractalWCOP02.pdf`.
2. Burke, B., and al. JBoss-AOP. `http://www.jboss.org/developers/projects/jboss/aop`.
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. Aspect-oriented programming. In Proceedings of the 11th European

Conference on Object-Oriented Programming (ECOOP'97) (June 1997), vol. 1241 of Lecture Notes in Computer Science, Springer, pp. 220–242.
4. McDirmid, S., and Hsieh, W. Aspect-oriented programming with jiazzi. In Proceedings of the 2nd international conference on Aspect-oriented software development (2003), ACM Press, pp. 70–79.
5. Medvidovic, N., and Taylor, R. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (May 2000).
6. Pawlak, R., Seinturier, L., Duchien, L., and Florin, G. JAC: A flexible solution for aspect-oriented programming in java. In Proceedings of Reflection'01 (Sept. 2001), vol. 2192 of Lecture Notes in Computer Science, Springer, pp. 1–24.
7. Pinto, M., Fuentes, L., Fayad, M., and Troya, J. Separation of coordination in a dynamic aspect oriented framework. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02) (April 2002), pp. 134–140.
8. Suve, D., and Vanderperren, W. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In Proceedings of the Dynamic Aspects Workshop (DAW) at AOSD'04 (March 2004).
9. Suve, D., Vanderperren, W., and Jonckers, V. JAsCo: an aspect-oriented approach tailored for component based software development. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03) (2003), ACM Press, pp. 21–29.
10. Szyperski, C. Component Software - Beyond Object-Oriented Programming, 2nd ed. Addison-Wesley, 2002.
11. Szyperski, C., and Pfister, C. Why objects are not enough. In Proceedings of the International Component Users Conference (1996).

An AOP Implementation Framework for Extending Join Point Models

Naoyasu Ubayashi[1], Hidehiko Masuhara[2], and Tetsuo Tamai[2]

[1] Kyushu Institute of Technology, Japan
[2] University of Tokyo, Japan
{ubayashi,masuhara,tamai}@acm.org

Abstract. Mechanisms in AOP (aspect-oriented programming) can be characterized by a JPM (join point model). AOP is effective in unanticipated software evolution because crosscutting concerns can be added or removed without making invasive modifications on original programs. However, the effectiveness would be restricted if new crosscutting concerns cannot be described with existing JPMs. Mechanisms for extending JPMs are needed in order to address the problem. In this paper, an implementation framework for extending JPMs is proposed. Using the framework, we can introduce new kinds of JPMs or extend existing JPMs in the process of software evolution.

1 Introduction

Mechanisms in AOP (aspect-oriented programming) can be characterized by a JPM (join point model) consisting of the join points, a means of identifying the join points (pointcuts), and a means of raising effects at the join points (advice). Crosscutting concerns may not be modularized as aspects without an appropriate join point definition that covers the interested elements in terms of the concerns, and a pointcut language that can declaratively identifies the interested elements. Each of current AOP languages is based on a few fixed set of JPMs. Many different JPMs have been proposed, and they are still evolving so that they could better modularize various crosscutting concerns.

AOP is effective in unanticipated software evolution because crosscutting concerns can be added or removed without making invasive modifications on original programs. However, the effectiveness would be restricted if new crosscutting concerns cannot be described with JPMs supported by current AOP languages. Mechanisms for extending JPMs are needed in order to address the problem. Masuhara and Kiczales presented a modeling framework that captures different JPMs by showing a set of interpreters[7]. We call this the M&K model. Based on the M&K model, we propose a framework, called X-ASB (eXtensible Aspect Sand Box), for implementing extensible AOP languages in which different JPMs can be provided as its extension. The term *framework* in this paper indicates provision of common implementations and exposure of programming interfaces for extending base languages. X-ASB is based on contributions of ASB[1] that is a suite of aspect-oriented interpreters such as PA (pointcuts and advice as in AspectJ) , TRAV (traversal specifications as in Demeter), and COMPOSITOR (class hierarchy composition as in Hyper/J). Advantages of X-ASB are: language developers can easily prototype new or extended JPMs in the process of software evolution; more than one JPM can be provided at the same time like combining Demeter-like traversal mechanism in AspectJ-like advice. Most of current extensible AOP languages allows the programmers to extend the elements of the JPMs in their language, not to introduce new JPMs. Our final goal is computational reflection for AOP. We consider facilities of adding new JPMs or changing existing JPMs from base level languages as reflection for AOP. The effectiveness in software evolution would be restricted if language developers must extend JPMs whenever application programmers need new kinds of JPMs. Reflective mechanisms will address this problem.

In this paper, issues on implementing AOP languages are pointed out in section 2. In section 3, X-ASB is introduced to tackle the issues. We show a JPM development process from the viewpoint of software evolution. In section 4, we show advanced topics towards computational reflection for AOP. In section 5, we discuss the effectiveness of X-ASB in software evolution. We introduce some related work in section 6, and conclude the paper in section 7.

2 Issues on implementing AOP languages

Designing and implementing a new language is not easy. Although extensible languages, such as computational reflection[6] and metaobject protocols would be useful for software evolution, providing an extensible AOP language that covers possible JPMs is not easy because the JPMs are drastically different from the viewpoint of implementation.

The M&K model shows the semantics of major JPMs by modeling the process of weaving as taking two programs and coordinating them into a single combined computation. A critical property of the model is that it describes the join points as existing in the result of the weaving process rather than being in either of the input programs. The M&K model explains each aspect-oriented mechanism as a weaver that is modeled as a tuple of 9 parameters:

$$< X, X_{JP}, A, A_{ID}, A_{EFF}, B, B_{ID}, B_{EFF}, META > .$$

A and B are the languages in which the programs p_A and p_B are written. X is the result domain of the weaving process, which is the third language of a computation. X_{JP} is the join point in X. A_{ID} and B_{ID} are the means, in the languages A and B, of identifying elements of X_{JP}. A_{EFF} and B_{EFF} are the means of effecting semantics at the identified join points. $META$ is an optional meta-language for parameterizing the weaving process. A weaving process is defined as a procedure that accepts p_A, p_B, and $META$, and produces either a computation or a new program. In terms of the M&K model, PA is modeled as follows: X: execution of combined programs; X_{JP}: method calls, field gets, and field sets; A: class, method, and field declarations; A_{ID}: method and field signatures; A_{EFF}: execute method; B: advice declarations with pointcuts; B_{ID}: pointcuts; B_{EFF}: execute advice before, after, and around method.

Although the M&K model parameterizes major JPMs, it makes no mention of implementation structures. In the current ASB implementation based on the M&K model, weavers are developed individually, and there is no common implementation among these weavers. It is impossible to add new kinds of JPMs unless the code of each weaver is re-implemented, and code regions to be modified are scattered. Although several differences among JPMs may make it difficult to actually implement a single parameterizable procedure, we believe that there is some kind of implementation structures that can be commonly applied to major JPMs. In the next section, we propose X-ASB as one of the common implementation structures.

3 X-ASB: A framework for extending JPMs

There are multiple framework layers for implementing or extending JPMs. The level 1 framework provides the common implementation for all kinds of JPMs and programming interfaces that must be implemented by JPM developers. The interfaces expose hot-spots for extending JPMs. This level gives JPM developers basic architecture for implementing JPMs. The level 2 framework provides advanced toolkit for implementing specific weavers such as PA-like weavers and TRAV-like weavers. Using the toolkit, JPM developers can implement individual weavers as well as multi-paradigm weavers that support several JPMs. For example, COMPOSITOR that supports PA-like before/after advice can be implemented. In this section, we explain the overview of the level 1 framework that is currently provided by X-ASB. We show a JPM development process using the code skeleton of the PA weaver and the extended PA weaver from the viewpoint of software evolution.

3.1 X-ASB overview

The overview of X-ASB, which is implemented in Scheme, is shown in Figure 1. The bottom half is a common implementation provided by X-ASB, and the top half is a set of programming protocol interfaces that must be implemented by JPM developers. The common implementation includes a base language interpreter, libraries provided for JPM developers, and other common implementations that are not shown here. Table 1 shows programming protocols as function names with their parameter name lists. Using the interfaces, a new kind of JPM can be added to the base language. The interfaces expose hot-spots for defining and registering join points (no.2), pointcuts (no.3), and advice (no.4, 5, 6, 7) because a JPM is characterized by these three components. The interfaces also expose hot-spots for defining a weaver body that mediates these components (no.1). Table 2 shows X-ASB libraries.

In X-ASB, JPMs can be systematically introduced or extended as follows: 1) define kinds of join points; 2) define kinds of pointcuts; and 3) define a body

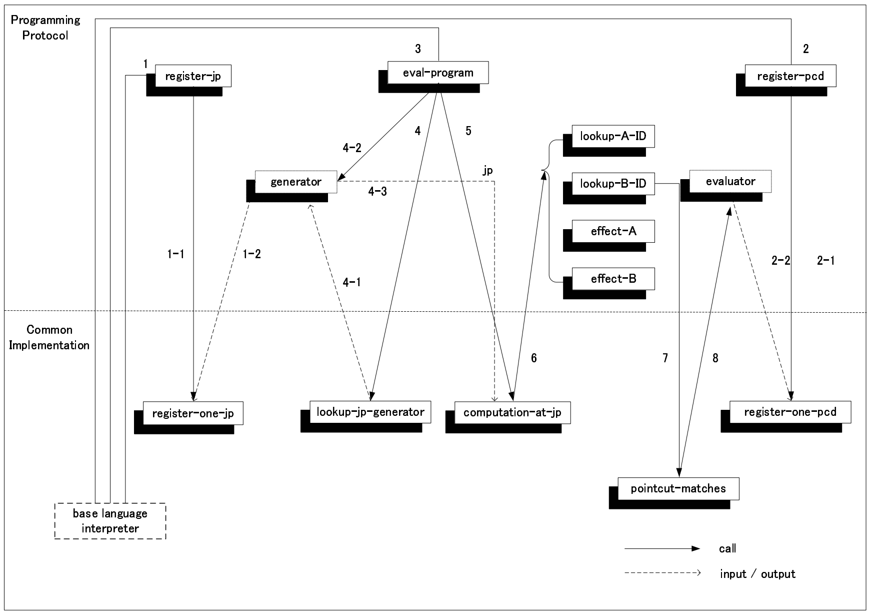


Fig. 1. X-ASB overview

of weaver, and computation at join points. The base language interpreter calls the functions `register-jp`, `register-pcd`, and `eval-program` implemented by JPM developers as follows:

```
(define weaver
  (lambda (pgm meta)
    (register-jp)
    (register-pcd)
    (eval-program pgm meta)))
```

We show a JPM development process using the code skeleton of the PA weaver that have only `method call` join point and `call` pointcut designator. The PA weaver processes, for example, the following program that calculates the factorial of n. Calls to procedure `fact` is declared as a pointcut, and `after` advice is executed at the join point corresponding to the pointcut.

```
(class sample-fact object
  (method void init () (super init))
  (method int main () (send this fact 6)) ;call method
  (method int fact ((int n))
    (if (< n 1) 1
        (* n (send this fact (- n 1)))))
   ;pointcut & advice
  (after (call fact) (write 'after) (newline)))
```

No.	Signature	Ret val	M&K
1.	`(eval-program pgm meta)`	none	X
2.	`(register-jp tag generator)`	none	X_{JP}
3.	`(register-pcd tag evaluator)`	none	X_{JP}
4.	`(lookup-A-ID jp param)`	A_{ID}	A_{ID}
5.	`(lookup-B-ID jp param)`	B_{ID}	B_{ID}
6.	`(effect-A A-ID jp param)`	none	A_{EFF}
7.	`(effect-B B-ID jp thunk param)`	none	B_{EFF}

Table 1. X-ASB programming protocol

No.	Signature	Ret val
1.	`(register-one-jp tag generator)`	none
2.	`(lookup-jp-generator tag)`	generator
3.	`(register-one-pcd tag evaluator)`	none
4.	`(pointcut-matches ptc jp)`	#t or #f
5.	`(computation-at-jp jp param)`	none
	`param: optional information`	

Table 2. X-ASB library

Step 1: define kinds of join points. First, JPM developers have to define kinds of join points and the related data structures including an AST (Abstract Syntax Tree) in PA, an object graph in TRAV, and so on. The interface `register-jp` and its parameter `generator` are used in the step 1 (see 1, 1-1, 1-2 in Figure 1). The `register-jp` interface registers a new kind of join point. Each join point is managed by the structure composed of a join point tag name and a `generator` that generates a join point. In the base language interpreter, there are several hook-points such as `call-method`, `var-set/get`, `field-set/get`, `if`, and so forth. A set of join points can be selected from these hook points. Crosscutting concerns such as loop structures can be extracted by selecting hook points concerning control expressions as join points. Crosscutting concerns such as data flows, on the other hand, can be extracted by selecting data access hook points. In general, data structures related to join points tend to be different drastically among JPMs. The `generator` parameter abstracts differences among these data structures. The `register-one-jp` library function helps JPM developers to implement the `register-jp` interface. The following is the code skeleton of the PA weaver.

```
(define register-jp
  (lambda ()
    (register-one-jp 'call-method generate-call-jp)))
(define generate-call-jp ...)
```

Step 2: define kinds of pointcuts. Next, kinds of pointcut designators must be defined as a boolean function using the `register-pcd` interface (see 2, 2-1, 2-2 in Figure 1). Each pointcut designator is managed by the structure composed of a pointcut tag name and an `evaluator` that checks whether a current join point is an element of a pointcut set. The `register-one-pcd` library function helps JPM developers to implement the `register-pcd` interface. The following is the code skeleton of the PA weaver.

```
(define register-pcd
  (lambda ()
    (register-one-pcd 'call call-pcd?)))
(define call-pcd?  ...)
     ; compare method name of pointcut designator
     ; and method name of method call join point
```

Step 3: define a body of weaver, and computation at join points. Lastly, JPM developers have to implement a body of a weaver using the `exec-program` interface that corresponds to the X parameter in the M&K model (see 3 in Figure 1). In the case of the PA weaver, the internal data structure related to join points is an AST. The `eval-program` creates an AST, walks it, and checks if the visited node is registered as a join point. At the `method call` join point, the `call-method` function related to the AST is called. Figure 2 illustrates the architecture of the PA weaver. The following is the body of the PA weaver.

```
(define eval-program
  (lambda (pgm meta)
    (walk-ast (generate-ast pgm meta))))
(define walk-ast
  (lambda (ast)
     ...
    ; computation at method call join point
    (call-method mname obj args)
     ...))
```

The computation at the `method call` join point, the `call-method` function, can be defined using the `computation-at-jp` library function, a generic (template) function as shown below (see 5, 6 in Figure 1). The `jp` parameter (see 4-3 in Figure 1) is an instance of a current join point generated by the generator (see 4-2 in Figure 1) that is registered by `register-jp` (see 1-2 in Figure 1). The registered join point generator can be searched using the `lookup-jp-generator` library function (see 4, 4-1 in Figure 1). The `lookup-A-ID`/`lookup-B-ID` and `effect-A`/`effect-B` interfaces corresponds to A_{ID}/B_{ID} and A_{EFF}/B_{EFF} in the M&K model, respectively. These interfaces must be implemented by JPM developers. Implementing the interfaces, a new kind of advice can be introduced. In `call-method`, the `call` pointcut evaluator `call-pcd?` is executed in the `pointcut-matches` (see 8 in Figure 1) invoked from the `lookup-B-ID` (see 7 in Figure 1), and the advice executor `effect-B` is executed.

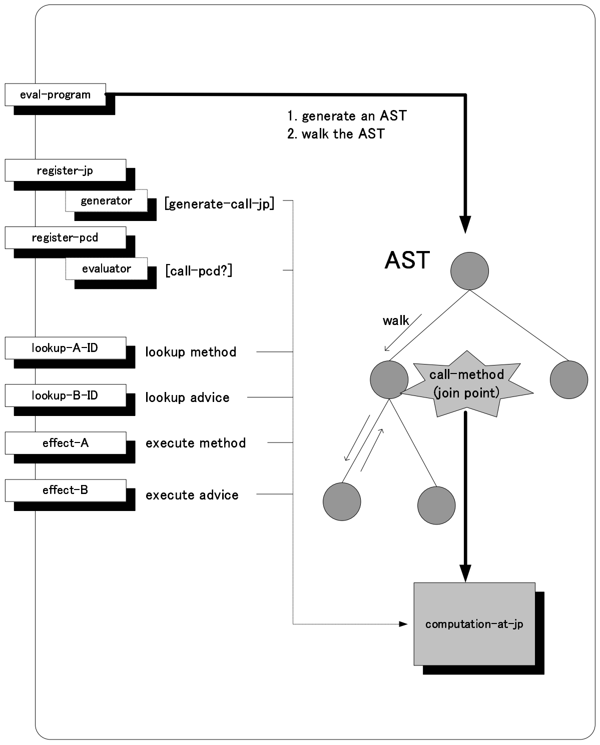


Fig. 2. PA weaver overview

```
;; X-ASB library
(define computation-at-jp
  (lambda (jp param)
    (let* ((A-ID (lookup-A-ID jp param))
           (B-ID (lookup-B-ID jp param)))
          (effect-B B-ID jp
            (lambda ()
              (effect-A A-ID jp param)) param))))

(define pointcut-matches
   ... search a pointcut evaluators corresponding
       to a join point, and execute a found evaluator.)
```

```
;; define computation at call method join point
;; using X-ASB library
(define call-method
  (lambda (mname obj args)
    (computation-at-jp
     ((lookup-jp-generator 'call-method) mname obj args)
       null)))               ; no additional parameter

(define lookup-A-ID     ...)  ; lookup method
(define lookup-B-ID     ...)  ; lookup advice
                              ; (check if the join point
                              ;  satisfies the pointcut
                              ;  conditions
                              ;  using pointcut-matches.
                              ;  return advice if true.)
(define effect-A        ...)  ; execute method
(define effect-B        ...)  ; execute-advice
```

As shown in step 1, 2, and 3, the PA weaver is constructed modularly using X-ASB. JPM developers have only to modify specific code regions such as `register-jp` and `register-pcd` when a new kind of join point and pointcut are needed. On the other hand, JPM developers must modify several code regions in order to add a new JPM element in the case of the current ASB implementation. We can separate JPM implementations using X-ASB. Separation of implementation concerns contributes to evolution of JPMs.

3.2 Extending existing weaver

It is desirable that one can extend an existing weaver slightly when JPMs that the weaver provides are insufficient for describing new kinds of features required in the process of software evolution. It is relatively easy to deal with this problem using X-ASB. The following is an example in which the PA weaver is extended in order to support context-sensitive calling sequences. The example is a communication program in which a protocol —an order of exchanged messages— is important. This program, written in the base language of X-ASB, separates a situation in which a protocol error might occur by defining a new kind of pointcut construct. Suppose that the order of message sequences is `<m1, m2, m3>`. The pointcut definition `(calling-sequence (not (list 'm1 'm2 'm3)))` catches the crosscutting concern that violates the order.

```
(class sample-protocol-error-detection object
   (method int m1 () (...))
   (method int m2 () (...))
   (method int m2 () (...))
   (after (calling-sequence (not (list 'm1 'm2 'm3)))
          (write 'invalid-calling-sequence) (newline)))
```

This pointcut designator can be added to the existing PA weaver as follows.

```
(define register-pcd
  (lambda ()
    (register-one-pcd 'calling-sequence
                      calling-sequence-pcd?)))
(define calling-sequence-pcd? ...)
```

4 Towards reflection

X-ASB exposes two kinds of programming interfaces for adding JPMs to the base language. The first is a set of programming interfaces provided for language developers that implement weavers as shown in section 3. The second is a set of programming interfaces provided for programmers that develop user applications and want to add JPMs specific to these applications. In the implementation style illustrated in subsection 3.2, only language developers can extend the PA weaver. It would be better for application programmers to be able to add new aspect-oriented features using X-ASB programming interfaces within the base language, as follows.

```
(class sample-calling-sequence object
  (method void register-pcd ()
    (meta register-one-pcd 'calling-sequence
                           calling-sequence-pcd?)
    (super register-pcd))
  (method boolean calling-sequence-pcd?  ...)
```

Application programmers may override the `register-pcd` method defined in the `object` class. To call procedures defined in the framework provided by X-ASB, programmers may use the `meta` call. Although the power of the extension is still limited, this brings to mind the reflective programming. The base language programming interfaces in X-ASB correspond to metaobject protocols in reflective OOP languages. Using reflective mechanisms, application programmers can extend JPMs in order to deal with unanticipated application-specific requirements in the process of software evolution.

5 Effectiveness in software evolution

As mentioned in previous sections, X-ASB is effective in unanticipated software evolution. We may face new kinds of crosscutting concerns, which cannot be handled by existing JPMs, in the process of software evolution. New kinds of JPMs will be needed when we face the following situations: 1) we want to extend existing JPMs slightly in order to adapt to application-specific purposes; 2) we want to use more than one JPM simultaneously. As an example of the first case, we showed a JPM development process of the extended PA weaver in section 3. The extended PA weaver was developed modularly to support context-sensitive calling sequences that were not provided by the original PA weaver. As an example of the second case, it may be necessary to combine PA-like JPMs with TRAV-like JPMs. This can be realized using the level 2 framework of X-ASB. Using X-ASB, we can extend new kinds of JPMs according to software evolution.

6 Related work

Shonle, Lieberherr, and Shah propose an extensible domain-specific AOP language XAspect that adopts plug-in mechanisms[8]. Adding a new plug-in module, we can use a new kind of aspect-oriented facility. CME (Concern Manipulation Environment)[3], the successor of Hyper/J, adopts an approach similar to XAspect.

Although pointcut languages play important roles in AOP paradigms, current AOP languages do not provide sufficient kinds of pointcut constructs. In an effort to address this problem, several previous investigations have attempted to enrich the pointcut constructs. Kiczales emphasizes the necessity of new kinds of pointcut constructs such as `pcflow` (predictive control flow) and `dflow` (data flow)[5]. Gybels and Brichau point out problems of current pointcut languages from the viewpoint of the software evolution, and propose robust pattern-based pointcut constructs using logic programming facilities[4]. These approaches introduce new pointcut constructs in order to deal with new kinds of crosscutting concerns. However, adopting these approaches, we need to add another pointcut construct to existing AOP languages whenever we face another kind of problem. As a consequence, the syntax of AOP languages would become very complex. Chiba and Nakagawa propose `Josh`[2] in which programmers can define a new pointcut construct as a boolean function. Using X-ASB, we can add not only new pointcut constructs but also new kinds of join points and advice.

7 Conclusion

The paper proposed mechanisms for extending JPMs in order to support unanticipated software evolution. Using X-ASB, we can introduce new kinds of JPMs when we face new kinds of crosscutting concerns that cannot be handled by existing JPMs.

8 Acknowledgement

This research has been conducted under Kumiki Project, supported as a Grant-in-Aid for Scientific Research (13224087) by the Ministry of Education, Science, Sports and Culture.

References

1. ASB(Aspect SandBox), http://www.cs.ubc.ca/labs/spl/projects/asb.html.
2. Chiba, S. and Nakagawa, K.: Josh: An Open AspectJ-like Language, In *Proceedings of Aspect-Oriented Software Development (AOSD 2004)*, pp.102-111, 2004.
3. Concern Manipulation Environment (CME): A Flexible, Extensible, Interoperable Environment for AOSD, http://www.research.ibm.com/cme/.
4. Gybels, K. and Brichau, J.: Arranging Language Features for More Robust Pattern-based Crosscuts, In *Proceedings of Aspect-Oriented Software Development (AOSD 2003)*, pp.60-69, 2003.
5. Kiczales, G.: The Fun Has Just Begun , *Keynote talk at Aspect-Oriented Software Development (AOSD 2003)*, 2003.
6. Maes, P.: Concepts and Experiments in Computational Reflection, In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87)*, pp.147-155, 1987.
7. Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2003)*, pp.2-28, 2003.
8. Shonle, M., Lieberherr, K., and Shah, A.: XAspects: An Extensible System for Domain-specific Aspect Languages, *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), Domain-Driven Development papers*, pp.28-37, 2003.

Evolving Pointcut Definition to Get Software Evolution

Walter Cazzola[1], Sonia Pini[2], and Massimo Ancona[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it
[2] Department of Informatics and Computer Science
Università degli Studi di Genova, Italy
{pini|ancona}@disi.unige.it

Abstract. In this paper, we have briefly analyzed the aspect-oriented approach with respect to the software evolution topic. The aim of this analysis is to highlight the aspect-oriented potentiality for software evolution and its limits. From our analysis, we can state that actual pointcut definition mechanisms are not enough expressive to pick out from design information where software evolution should be applied. We will also give some suggestions about how to improve the pointcut definition mechanism.

Keywords: AOP, Software Evolution, Design Information, UML, Pointcut Definition

1 Software Evolution: What is it?

Nowadays a topical issue in the software engineering research area consists of producing software systems able to adapt themselves to environment changes by adding new and/or modifying existing functionality. This characteristic is called *software evolution*.

The term *evolution* may, generally, be interpreted and studied from several distinct points of view. In general software evolution implies to reengineering the design and the code of software systems. Software evolution and maintenance can be categorized into [9]: *corrective*, *adaptive*, *perfective*, and *preventative*. The criteria that govern this taxonomy are well identified by the motivations that render necessary the evolution, e.g., adaptive software evolution is necessary when new functionality are required.

Nonstopping applications with long life span are typical applications that have to be able to dynamically adapt themselves to sudden and unexpected changes to their environment. Therefore, the support for run-time adaptive software evolution is a key aspect of these systems.

Design information provides all the necessary data for governing software evolution and is often used for manually evolving systems that can be stopped. Object oriented methodologies for software development, as UML [1], describe the system's behavior, architecture and components; all functions in the system are captured by a use-case model and the dynamic behavior of each use-case is described by scenarios and interaction diagrams. Therefore, the automatic reengineering of the design information of a non-stopping system should represent the perfect tool for dynamically adapting such kind of of systems.

Unfortunately, design data are difficult to manage automatically but especially it is difficult to automatically generate working code from the design and inject it in the running system. In this case, the evolution can be carry out by defining some mechanisms that face the occurred events, manipulate the UML diagrams and then inject such a changes directly and automatically in the code. As discussed in [2, 3] the diagram manipulation is feasible by working on their XMI representation and by using a set of reconfigurable rules for planning the adaptation but the code injection is still far from being achieved.

Software evolution that involves a generic system is usually carried out stopping the system and manually, or with the aid of specific tools, changing the system code according with the required evolution. On the other hand, a similar approach is not feasible when the system subjects to the evolution cannot be stopped, e.g., because provides a critical service as a monitoring system.

Independently of the mechanism adopted for planning the evolution, the evolution of a nonstopping system requires a mechanism that permits of concreting the evolution on the running system. In particular this mechanism should be able of i) picking the code interested by the evolution out of the whole system code, ii) carrying out the patches required by the planned evolution on the located code.

Both computational reflection [8] and aspect-oriented programming [5] provide mechanisms (introspection and intercession the former and aspect weaving the latter) that allow of modifying the behavior and the structure of an application, also of a non-stopping application. Reflective mechanisms mainly focus their efforts on dynamically modifying the system on a per object-basis whereas the AOP mechanisms better address functionality that crosscut the whole implementation of the application. Evolution is a typical functionality that may crosscut the code of many objects in the system.

2 Why could AOP be useful for Software Evolution?

Aspect oriented programming (AOP) [5] is a designing and programming technique that takes another step towards increasing the kinds of design concerns that can be cleanly captured within source code. Its main goal consists of providing systematic means for the identification, modularization, representation and composition of crosscutting concerns such as security, mobility and real-time constraints. Moreover, the captured aspects (both functional and nonfunctional) are separated into well-defined modules that can be successively composed in the original or in a new application.

Where the tools of OOP are inheritance, encapsulation, and polymorphism, the components of AOP are *join points*, *pointcut*, and *advice*. Join points represent well-defined points in a program's execution, such as method calls, field get and set methods. Pointcut is a construct that picks out a set of join points based on defined criteria, such as method names and so on. Pointcuts serve to define an advice. An advice picks out additional code to be executed before, after, or around join points. Typical implementations of object-based AOP frameworks insert hooks at the join points. An advice is executed when the execution reach the corresponding (i.e., picked out by a pointcut) join point. AspectJ [7] is one of the most relevant frameworks supporting the AOP methodology.

AOP can be classified in *static* and *dynamic* AOP. The systems, as AspectJ, compliant to the static approach permit of weaving aspects at compile or load-time. On the other hand, the dynamic AOP approach allows of dynamically plugging and unplugging aspects at run-time widening the applicability spectrum of the AOP methodology. The dynamic AOP approach requires a support middleware at run-time, called *execution monitor*; the system raises a callback to that middleware to notifies that a hooks has been encountered. The middleware takes also care of executing the advice.

Many frameworks support or may be used for implementing dynamic AOP, e.g., JAC [13], DJ [12], Prose [14], Wool [15] and JMangler [6].

From AOP characteristics, it is fairly evident that AOP is on the way of providing the necessary tools for instrumenting the code of a nonstopping system, especially when advices can be run at run-time. Pointcut should be used to pick out a region of the code involved by the evolution, whereas the advice should define the code evolution at the corresponding pointcut. Weaving such an aspect on the running system should either inject new code or manipulate the existing code, allowing the system dynamic evolution.

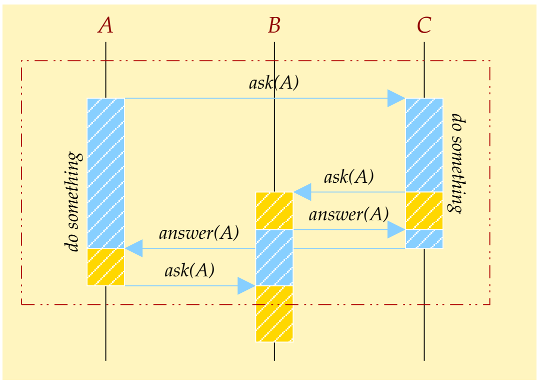
Unfortunately, in the evolution case, pointcut definition is an hard job because the portion of code interested by the adaptation can be scattered around in the code and not confined in a well-defined area that can be taken back to a method call. As pointed out by Tom Tourwé et al. in [17], this problem is due to the poor expressiveness of the pointcut definition languages provided by the actual AOP frameworks.

The developer has to identify and to specify in the correct way the pointcut. To pick out the pointcuts, the developer can use, what we call *linguistic pattern matching*. Nowadays, the pointcut definition languages permit to locate where an advice should be applied by describing the pointcut as a mix of references to linguistic constructs, such as method call or access to variables, and of generic references to the position, such as before or after; the result, for example, looks something like `after the call of method m`. Therefore, it is difficult to define generic, reusable and comprehensible pointcuts that are not tailored on a specific application. Moreover a similar approach is not feasible when the pointcut should involve code that spans among several classes, as in the case of a pointcut describing a collaboration among objects.

3 Towards a Pointcut Definition Driven by UML Diagrams

Several mechanisms for achieving software adaptability have been proposed [18, 11]. The approach that we believe the most promising consists of integrating a reflective architecture as the one proposed in [3] with an AOP framework. The reflective middleware has to take care of deciding the extent of the evolution and which code is affected by such an evolution. Whereas the AOP framework has to dynamically weaving the planned evolution on the join points picked out by the reflective architecture. Both frameworks should perform their duty manipulating the design information of the system prone to be adapted.

It is relatively simple to plan the system evolution by manipulating its UML diagrams and similarly it is quite simple to detect the extent of the code modification and which objects are affected by them from the design information. On the other hand, to use design information to pick out a set of pointcut is not so simple because what it is concisely described by a diagram or a portion of a diagram might be implemented by many instructions disseminated in several part of the code or, analogously, what it is abstracted in several entities by the design diagrams can be implemented as a single entity by the system code. Besides, as discussed in [17], computational patterns, easily recognizable in a sequence or in a collaboration diagram, are not trappable by actual pointcut definition languages.



The above reported portion of sequence diagram describes a quite frequent collaboration among three entities, A, B and C; A asks to C to intercede with B on its behalf, then, after the successful intercession, A will interact with B. This schema could be used to describe a control access protocol with an external authenticator. Notwithstanding that, it is quite simple to understand the computational pattern described by this sequence diagram is not so simple to pick out which code realize it, especially on a pattern matching bases. In fact, the diagram just describe the order and which operation MUST be done, but nothing is said about which code do that and about what A is doing while it is waiting an answer from C. Therefore, the computational pattern expressed in the squared portion of the reported sequence diagram cannot be picked out by the actual pointcut definition languages. As stated in [17] something can be done refactoring the code in order to localize and to encapsulate the code of each entity in a method, but still nothing can be done to pick out the collaboration among three entities.

Problems related to pointcut definition have been raised by several researchers [17, 4], in all their works they propose to use a more expressive pointcut definition language mainly based on logic deduction and pattern matching. Notwithstanding the powerfulness of their proposals, they cannot deal with the straightforwardness and the abstraction provided by a UML diagram. A pointcut defined in term of UML diagrams picks out portion of code otherwise not identifiable, as explained above. Sillito et al., in [16], highlighted the importance of using *use case* diagrams in the pointcut definition, our idea is quite similar but we do not want to define a novel pointcut definition language, as AspectU, that needs a special interpreter or to be mapped on an existing AOP language, as AspectJ. Rather we would like to extend an existing pointcut language and act on the weaving mechanism to support UML-based join points.

Our proposal consists of using UML diagrams or portion of UML diagrams to describe where the advice should be woven. In this way, pointcuts are not tailored on the program to adapt but they are more general and represent patterns applicable to several computational flows. Of course, we will use a textual representation instead of a graphical one based on the XMI standard [10] to define our pointcuts. Moreover, we will exploit meta-data code annotations as supported by .NET or Java (version 1.5) to introduce XMI code, that will play the role of the hooks, in the code to be adapted. Annotations have the benefit to be supported by standard programming environments and to be skipped during the normal execution, i.e., in this case, when no aspect is woven on that annotation; therefore they should not add extra penalties during the execution.

4 Conclusions and Future Work

In this position paper, we have analyzed aspect-oriented development techniques in relation with the software evolution problem. In particular, we have focused our analysis on the approach to software evolution that we believe the most promising: software evolution driven by design information. From our examination results that with actual mechanism for pointcut definition is hard to pick out the code described by UML diagrams that with a higher abstraction level. Similar issues related to pointcut definition have also been raised by other researchers [17, 4]. Our proposal consists of marking the code with the corresponding UML diagrams (hooks for the weaving mechanism) and of using such diagrams in the definition of the pointcuts and therefore in helping to pick out where evolution should take place. In the next, we will extent the pointcut definition language of an existing AOP framework, as AspectJ, with the possibility of using UML diagrams as pointcut as well. Analogously, we will enable the weaving mechanism to act in conjunction with join points specified by such a kind of pointcuts.

References

1. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
2. Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Pavel Hruby and Kristian Elof Søresen, editors, *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02)*, pages 35–54, Højstrupgård, Helsingør, Denmark, on 20th-22nd of September 2002. Microsoft Business Solutions.
3. Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrich, John-Jules Meyer, and Mark D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science, pages 69–84. Springer-Verlag, Heidelberg, Germany, February 2004.
4. Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 60–69, Boston, Massachusetts, April 2003.

5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.
6. Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - A Powerful Back-End for Aspect-Oriented Programming. In Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-oriented Software Development*, chapter 9. Prentice Hall, 2004.
7. Ramnivas Laddad. *AspectJ in Action: Pratical Aspect-Oriented Programming*. Manning Pubblications Company, 2003.
8. Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.
9. Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. Towards a Taxonomy of Software Evolution. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution (USE 2003)*, Warsaw, Poland, April 2003.
10. OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.
11. Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE'98)*, pages 177–186, Kyoto, Japan, April 1998.
12. Doug Orleans and Karl Lieberherr. DJ: Dynamic Adaptive Programming in Java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, LNCS 2192, pages 73–80, Kyoto, Japan, September 2001. Springer.
13. Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, LNCS 2192, pages 1–24, Kyoto, Japan, September 2001. Springer.
14. Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just in Time Aspects: Efficient Dynamic Weaving for Java. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 100–109, Boston, Massachusetts, April 2003.
15. Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A Selective, Just-in-Time Aspect Weaver. In *Proceedings of the 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE'03)*, LNCS 2830, pages 189–208, Erfurt, Germany, September 2003. Springer.
16. Jonathan Sillito, Christopher Dutchyn, Andrew D. Eisenberg, and Kris De Volder. Use Case Level Pointcuts. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, Oslo, Norway, June 2004.
17. Tom Tourwé, Andy Kellens, Wim Vanderperren, and Frederik Vannieuwenhuyse. Inductively Generated Pointcuts to Support Refactoring to Aspects. In *Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'04)*, Lancaster, UK, March 2004.
18. Emiliano Tramontana. Managing Evolution Using Cooperative Designs and a Reflective Architecture. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 59–78. Springer-Verlag, Heidelberg, Germany, June 2000.

Part (IV): Parametric Aspects and Generic Aspect Languages

Chairman: Hidehiko Masuhara, University of Tokyo, Japan.

Parametric Aspects: A Proposal

Jordi Alvarez

Universitat Oberta de Catalunya
Avinguda Tibidabo 39–43, 08035 Barcelona, Spain
jalvarezc@uoc.edu

Abstract. Aspect-Oriented Software Development (AOSD) provides better design solutions to problems where Object Oriented Development produces tangled and scattered designs. Nevertheless, there are still several problems for which AOSD is not helpful. An example is the implementation of some design patterns. While it has been shown that some of them can be implemented in a domain-independent way by the use of AOSD [1], there is still a group of them for which current AOSD techniques are of little use. This paper proposes to extend Aspect Oriented Languages with parametric aspects. This extension can integrate seamlessly into Aspect Oriented Languages like AspectJ, and allows to provide a better design solution for problems for which current AOSD techniques are of little help, improving software reuse and reducing its complexity, thus facilitating the software evolution process. Two representative examples are used in order to expose the proposed extension: the implementation of the abstract factory pattern in a domain-independent way, and that of a simple Enterprise Java Bean.

1 Introduction

AOSD solves some of the problems that arise in OO Development [2, 3]. In problems where OO produces a tangled and scattered design, Aspect Orientation can achieve a cleaner and more compact design. Nevertheless, the ideas behind AOSD are in continuous evolution and have not yet been completely developed. Software engineering properties promoted by the use of AOSD can still be further improved in several directions. This paper proposes an extension that allows to increase software reuse and to decrease software complexity. Software complexity has been identified as one of the main drawbacks for a successful software evolution process [4]. Thus, we are concerned here with the capability to produce software systems more suitable for software evolution.

The implementation of GoF design patterns [5] with AO techniques (using the AspectJ language [6]) has been boarded in [1]. A subset of 12 design patterns from [5] has been implemented in an abstract way (completely independent of the domain). This means that pattern implementation can be reused for different applications of the design pattern to different domains. The rest of the patterns in [5] cannot be completely implemented in a domain independent way with current AOSD techniques. It is argued that they are "too abstract" in order to provide a domain-independent implementation for them [7]. Nevertheless, these problems for which current AO Languages are not able to provide qualitatively cleaner and more compact design, still show some regularities and common behaviour. When software needs to be evolved, these regularities should be taken into account in the evolution process; and their complexity has an affect on the resulting complexity of the evolution process.

This paper explores how ideas in parametric types (or genericity) can be extended to AO languages. A proper combination of both can be powerful enough to capture some of the regularities that AO languages alone were not able to capture. Our AO extension allows to decouple these regularities from the part of the software system that corresponds to the application domain, coding them only once in an abstract enough way. As a result, the complexity of the latter can be significantly reduced. As this is mainly the part of software systems that is subject to software evolution, also the complexity that the software evolution process shall deal with, can be importantly reduced.

Parametric types (or generics) were introduced to the main stream of the software industry with the C++ language [8]. Also, several proposals to add parametric types to the Java language exist [9–13], and the next Java release: Java SDK 1.5, will allow them [14, 15]. The main idea behind parametric types is to introduce parameters into type definitions. In this way, regular types can be defined in two phases: (1) a "generic" type is defined, in which one or more formal parameters take part, and (2) a regular type can be created as an instance of the generic type just by replacing (at compile time) the formal parameters by other types or constants.

Our approach borrows this two-phase definition mechanism for aspects. This does not only adds genericity 'a la C++' to AO Languages, but also takes benefit from AO ideas in order to provide a more powerful genericity that perfectly fits into the AO approach. The paper also uses two very representative examples in order to show how this extension helps indeed to produce much more compact designs.

2 Parametric Aspects Definition

This section describes which constructs do we propose to add to aspect languages in order to be able to define parametric aspects. Although the proposal is valid for the aspect oriented approach in general, an extension of AspectJ syntax has been developed and is used in the provided examples.

Throughout the text, the *AbstractFactory* GoF design pattern [5] will be used as an example. This design pattern is one of the patterns for which current AOSD techniques cannot provide a domain-independent implementation. It will be shown how the parametric aspect extension proposed here allows to provide a domain-independent implementation of a restricted version of the *Abstract-Factory* pattern, allowing to reuse it from domain to domain. In this way, if the application domain changes and the software needs to be evolved, the design pattern implementation will not take part in the performed changes, remaining unchanged.

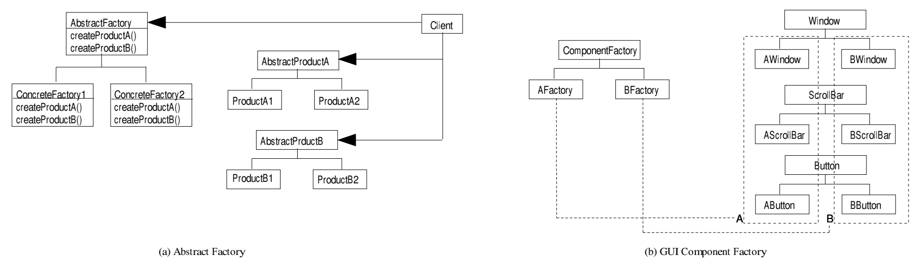


Fig. 1. Representation of the *AbstractFactory* pattern and a simplified implementation of it for the domain of GUI components.

Figure 1 part (a) shows a representation of the *AbstractFactory* pattern. The goal of this pattern is to detach the creational behaviour for classes in a hierarchy (abstract products) from the hierarchy itself. This allows to provide different creational strategies (factories) without the need to modify the hierarchy. Part (b) corresponds to the application of the *AbstractFactory* pattern to the domain of GUI components.

The same as parametric types, parametric aspects have a set of formal parameters attached to its definition. In what follows, we refer to these parameters as *roles*, in a similar way to the roles taking part in design patterns. Roles in parametric aspects may correspond to types (classes), methods, attributes, and constants. Note that this is different to parametric types, where parameters correspond only to types or constants.

The piece of code in next page shows the implementation of a restricted version of the *AbstractFactory* pattern through a parametric aspect. The following constructs are part of the syntax extension to AspectJ:

– The **roles** keyword is used to specify the roles (parameters) taking part in the parametric aspect. Roles may correspond to classes/types, methods, attributes, constants, and class sets. From now on we will refer to these elements with the common name of language objects.
– The **generic** keyword allows to partially define classes, methods, and other language objects. These "generic" definitions contain references to roles, and are not properly defined as regular language objects until the parametric aspect is instantiated, and the roles are replaced by regular language objects (see next section). So, these "generic" definitions are indeed skeletons that will be completed when the aspect is instantiated.
– One skeleton might correspond to multiple definitions when the parametric aspect is instantiated. When this is possible, the keyword **multiple** is

used. `AbstractFactoryAspect` (see next page) contains one class skeleton for `<ProductSet>Factory`. When the parametric aspect is instantiated, this only definition produces one class definition for every class-set playing the role `ProductSet` (see also next section).
– Class sets are defined through the **class-set** keyword. Class sets will not have a counterpart in the final program code. Nevertheless, the possibility to have this kind of roles, and to use them in skeleton definitions will be shown to be very useful.
– A special expression language is also provided in order to use roles within generic definitions (skeletons). The expressions are evaluated at weave time and enclosed between '<' and '>' symbols. Simple expressions allow just to refer to roles; and more complex expressions can conveniently traverse the class hierarchy (see `create<AbstractProduct>` method definition, and next section for the explanation). These expressions will be referred from now on as *parametric expressions*. Class, method, and attribute names can be a combination of parametric expressions and text prefixes and suffixes. This allows to define very powerful skeletons.

```
abstract aspect AbstractFactoryAspect {
  roles AbstractFactory, AbstractProduct, ProductSet;

  generic class <AbstractFactory> {
    multiple public abstract <AbstractProduct> create<AbstractProduct>();
  }

  multiple generic class <AbstractProduct>;

  multiple generic class-set <ProductSet>;

  multiple generic class <ProductSet>Factory extends <AbstractFactory> {
    multiple public <AbstractProduct> create<AbstractProduct>() {
      return new <subclasses(AbstractProduct) & members(ProductSet)>();
    }
  }
```

The definition of `AbstractFactoryAspect` contains several class-skeleton definitions: `AbstractFactory`, `AbstractProduct`, and `<ProductSet>Factory`. Additionally, `<AbstractFactory>`, and `<ProductSet>Factory` contain also method-skeleton definitions for the factory creational methods. All these definitions may have a total or partial definition inside the parametric aspect. These definitions can be further completed when the aspect is instantiated. This allows language objects playing a given role to be forced to respect a given structure.

3 Aspect Instantiation

Parametric aspects will always be abstract aspects. They can be instantiated using the **extends** keyword, resulting in regular aspects that can be used by developers as any other concrete aspect in the system. In order to achieve this, the aspect that instantiates the parametric aspect must provide replacement for all the roles appearing in it.

All roles must be accordingly replaced by language objects (regular types, methods, attributes, constants, or class sets). For each one of these language objects, we say that the language object *plays* the corresponding role.

The piece of code below shows how `AbstractFactoryAspect` can be instantiated for the GUI component domain. The resulting `ComponentFactoryAspect` only needs to specify the replacement objects for the roles that have been declared in `AbstractFactoryAspect`. In other situations, additional behaviour could be provided by the concrete aspect in the usual way.

```
aspect ComponentFactoryAspect extends AbstractFactoryAspect {
    class ComponentFactory plays AbstractFactory;
    class Window plays AbstractProduct;
    class ScrollBar plays AbstractProduct;
    class Button plays AbstractProduct;
    class-set A plays ProductSet { AWindow, AScrollBar, AButton; }
    class-set B plays ProductSet { BWindow, BScrollBar, BButton; }
}
```

`ComponentFactoryAspect` shows how, once the *AbstractFactory design pattern* is defined as a parametric aspect, applying the pattern to a concrete domain can easily be done by defining another aspect that fills the gaps corresponding to the roles for the application domain.

`ComponentFactoryAspect` replaces the role `AbstractFactory` by the `ComponentFactory` class. We say that `ComponentFactory` plays the `AbstractFactory` role. There are three different classes (`Window`, `ScrollBar`, and `Button`) that play the `AbstractProduct` role. Two class-sets:`A` (containing `AWindow`, `AScrollBar`, and `AButton`) and `B` (containing `BWindow`, `BScrollBar`, and `BButton`) play the `ProductSet` role.

Then, if the application domain changes (for example, a new factory or a new set of products is needed), and our software system needs to be adapted to it, only new roles and their specific behaviour must be defined. The abstract factory design pattern implementation has been completely decoupled from the application domain code, and can remain unchanged, favouring the software evolution process requested by the domain change.

Skeleton definitions in the abstract parametric aspect correspond to language objects. Every one of these language objects has a *unique identifier*. The unique identifier of a class is its name and package. The unique identifier of a method is its name plus the type of its parameters plus the unique identifier of the class that contains it. The unique identifier of a skeleton definition in the parametric aspect can contain parametric expressions. In order to produce regular language object definitions from the skeletons, the parametric expressions that take part in their unique identifier are evaluated according to the role assignments established in the concrete aspect. When several language objects play the same role, all the possible combinations are generated, and one (class/method/...) definition is generated for every combination.

Then, as `ComponentFactoryAspect` declares two `ProductSet`, two class definitions would be generated for `<ProductSet>Factory`: `AFactory`, and `BFactory`. The same happens with method definitions: as there are three classes that play the `AbstractProduct` role, three method definitions would be generated for the `create<AbstractProduct>` method skeletons in `AbstractFactoryAspect`. This would result in three methods for each one of the classes: `ComponentFactory`, `AFactory`, and `BFactory`. The set of classes and method definitions that would be generated at weave time is shown below:

```
class ComponentFactory {
  public abstract Window createWindow();
  public abstract ScrollBar createScrollBar();
  public abstract Button createButton();
}
class AFactory extends ComponentFactory {
  public Window createWindow()       { return new AWindow(); }
  public ScrollBar createScrollBar() { return new AScrollBar(); }
  public Button createButton()       { return new AButton(); }
}
class BFactory extends ComponentFactory {
  public Window createWindow()       { return new BWindow(); }
  public ScrollBar createScrollBar() { return new BScrollBar(); }
  public Button createButton()       { return new BButton(); }
}
```

In order to be able to generate the bodies of the methods in `AFactory` and `BFactory` from just one method skeleton in `AbstractFactoryPattern`, the *parametric expression language* must be powerful enough. With this goal, a language is provided that allows to deal with class sets, obtain the superclasses and the subclasses of a given class, and make intersections and unions of different class sets. This behaviour is obtained with the use of a few functors (**members**, **subclasses**, **superclasses**, **subclasses***, and **superclasses***) in combination with set-related operators (`&` and `|`). The complete syntax is not shown here for lack of space. This reduced set of features allows to conviniently traverse the class hierarchy at weave time.

```
<subclasses(AbstractProduct) &
              members(ProductSet)>
```

The above parametric expression (extracted from `create<AbstractProduct>` method skeleton in `AbstractFactoryAspect`) computes the intersection of the subclasses of the class playing the `AbstractProduct` role and the classes in the class-set playing the `ProductSet` role. That is, it computes the concrete product to be created for a given factory and an abstract product.

As parametric expressions are evaluated at weave time, they cannot deal with modifications introduced into the class hierarchy through the use of runtime reflection. Additionally, the weaving process itself may update also the hierarchy, which poses a recursion problem as the evaluation of parametric expressions before the weaving process might differ from their evaluation after it. The simplest solution is taken and only the hierarchy before the weaving process is taken into account.

4 Related Work

The introduction of parametric type ideas into AOSD has also been boarded recently by other researchers. [16] proposes parametric introductions, which allow to add the classes to which introductions apply as parameters to the introduction definitions themselves. This allows, for example, to easily implement the *singleton* design pattern. The main difference with our approach is that [16] has as parameters the classes being instrumentalized, whereas in our approach parameters are one more element in the aspect itself, and might refer to newly created language objects (classes, methods . . .). On the other hand, the use of the parametric expression language provides a very powerful way to define behavior.

Family Polimorphism [17] addresses a similar problem, although it is not related to the AOSD approach. Also, [18] combines aspects and frame technology, with the goal of adding parametrisation and also deeper generalisation capabilities to AOSD systems.

5 Conclussion and Further Work

This paper proposes to extend AOP languages in order to support parametric aspects. The extension allows to define behaviour in a domain-independent way for problems for which the existing AO languages are only able to provide an overall domain-dependent solution. As a consequence, the complexity of the part of software systems that are dependent on the domain, and the complexity that the software evolution process must deal with, can be significantly reduced.

An extension of AspectJ with the syntax shown in this paper is currently under development. Additionally, the scope of application of the proposed extension is being evaluated for other design patterns and also other different problems.

A Appendix. A Second Example: EJB Development

EJB development is a tedious task. Many interfaces with redundant code need to be developed. This task can be lightened by development environments and tools. Nevertheless, no solution is provided from the programming language point of view. The use of parametric aspects could help in avoiding the development of redundant code, as the code below shows.

```
abstract aspect EJB {
    roles Bean, beanService;
    generic class <Bean> {
        generic multiple public <beanService>;
    }
    interface <Bean>Remote extends javax.ejb.EJBObject {
        public <beanService> throws java.rmi.RemoteException;
    }
    interface <Bean>RemoteHome extends javax.ejb.EJBHome {
        <Bean>Remote create() throws java.rmi.RemoteException,
                             javax.ejb.CreateException;
    }
    interface <Bean>Local extends javax.ejb.EJBLocalObject {
        public <beanService>;
    }
    interface <Bean>LocalHome extends javax.ejb.EJBLocalHome {
        <Bean>Local create() throws javax.ejb.CreateException;
    }
}

aspect HelloEJB extends EJB {
    class Hello plays Bean {
        String hello(String name) plays beanService {
            return "Hello "+name+"!";
        }
    }
}
```

The `EJB` aspect definition declares two roles and provides skeletons for all the interfaces needed to define an EJB. Then, in order to define a simple EJB, we only need to extend the `EJB` aspect with a concrete aspect (in the example above, the `HelloEJB` aspect), and define the class that plays the `Bean` role, and the method that plays the `beanService` role.

References

1. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proceedings of OOPSLA'02. (2002) 161–173
2. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Proceedings of ECOOP'97. Number 1241 in Lecture Notes in Computer Science, Springer Verlag (1997) 220–242
3. AOSA: Aspect Oriented Software Development (2003) `http://www.aosd.net`.
4. Lehman, M.M., Ramil, J.F.: Rules and tools for software evolution planning and management. Annals of Software Engeneering **11** (2001) 15–44
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, M.: An overview of AspectJ. In: Proceedings of ECOOP'01. Volume 2072 of Lecture Notes in Computer Science., Springer Verlag (2001) 327–353
7. Noda, N., Kishi, T.: Implementing design patterns using advanced separation of concerns. In: Workshop on Advanced Separation of Concerns in Object Oriented Systems - OOPSLA'01. (2001)

8. Stroustrup, B.: The C++ Programming Language. second edn. Addison Wesley (1991)
9. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In Chambers, C., ed.: Proceedings of OOPSLA'98, Vancouver, BC, ACM SIGPLAN Notices (1998) 183–200
10. Odersky, M., Wadler, P.: Pizza into Java: Treanslating theory into practice. In Jordan, N.D., ed.: Proceedings of POPL'97, Paris, France, ACM, ACM Press (1997) 146–159
11. Myers, A., Bank, J., Liskov, B.: Parameterized types for Java. In Jordan, N.D., ed.: Proceedings of POPL'97, Paris, France, ACM, ACM Press (1997)
12. Agesen, O., Freund, S.N., Mitchell, J.C.: Adding type parameterization to the Java programming language. In Bloom, T., ed.: Proceedings OOPSLA'97, Atlanta, Georgia, ACM Press (1997)
13. Cartwright, R., Steele, G.L.: Compatible genericity with runtime-types for the Java programming language. In Chambers, C., ed.: Proceedings of OOPSLA'98, Vancouver, BC, ACM SIGPLAN Notices (1998)
14. Bracha, S.L.G.: JSR 14: Add generic types to the Java programming language (2001) http://jcp.org/en/jsr/detail?id=14.
15. Torgensen, M., Hansen, C.P., Ernst, E., von der Ah, P., Bracha, G., Gafer, N.: Adding wildcards to the Java programming language. In: Proceedings of the 19th ACM Symposium on Applied Computing. (2004)
16. Hanenberg, S., Unland, R.: Parametric introductions. In Akşit, M., ed.: Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 80–89
17. Ernst, E.: Family polymorphism. In Knudsen, L., ed.: Proceedings of ECOOP'01. Volume 2072 of LNCS., Springer Verlag (2001) 303–326
18. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Support product line evolution with framed aspects. In: Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Lancaster, England (2004)

Dynamic Framed Aspects for Dynamic Software Evolution

Philip Greenwood, Neil Loughran, Lynne Blair, Awais Rashid

Computing Department, Lancaster University, Lancaster LA1 4YR, UK
greenwop|loughran|lb|awais@comp.lancs.ac.uk

Abstract. Software evolution is an inevitable process when developing a system of any notable size and is the most costly stage in the life cycle of a system. Automating parts of this process will reduce the resources required to carry out this stage of development. We aim to develop a framework that achieves this automated evolution by using Dynamic AOP to encapsulate these evolutionary changes and allow them to be applied dynamically at runtime. However, a problem with this is being able to reuse these aspects in different systems and scenarios. We propose the use of framed aspects to parameterise the aspects to generalise them so they can then be customised for a specific use.

1. Introduction

Software evolution is an inevitable stage when developing any type of software. Evolution is defined in [16] as "the process of changing a system to maintain its ability to survive". There are three types of maintenance that can be applied to a system:

- Corrective Maintenance
- Adaptive Maintenance
- Perfective Maintenance

Large amounts of time and money are spent on software evolution and so it is desirable to reduce the amount of effort required to perform this stage of development. In this paper we will outline a framework that will allow a system to evolve certain parts of itself automatically and so reduce the effort to maintain the system. The framework will concentrate on performing perfective maintenance which is generally considered to be maintenance that implements new functional or non-functional requirements.

The use of dynamic Aspect-Orient Programming (AOP [10]) has been proposed to develop an autonomic system in [6]. Autonomic systems [7] are those which are able to perform certain levels of maintenance upon themselves and so can evolve dynamically. The properties that dynamic AOP possesses will allow these changes to be well encapsulated and applied at run-time without needing the system to be taken off-line.

Problems in this domain, regarding the reuse of the aspects and creating the aspects to be applicable in a variety of different scenarios will arise. Framed aspects will allow the parameterisation of each aspect and allow it to be customised to a variety of systems and scenarios.

Framed aspects is a new approach that allows for the easy parameterisation of aspects. The use of framed aspects has been proposed in previous work [11]. The context set out in [11] was to allow the easy development of software product lines. The framed aspects were used to extract common cross-cutting concerns from a particular system family and were then parameterised to allow the easy customisation of the aspect for a particular version of the system. This process improved the reuse of the aspect and reduced development time for later versions. This paper proposes a different application of framed aspects which will require extensions being made to their behaviour.

The aim of this paper is to describe the use of Dynamic Framed Aspects – framed aspects that can be created and applied dynamically to a running system. The paper will illustrate how such a framework could be implemented and will highlight certain key issues.

The structure of the paper is as follows. Section 2 describes in more detail the implementation and use of framed aspects. Section 3 then looks at the use of dynamic AOP to implement a dynamically evolvable system and highlights some of the problems that can arise. Section 4 examines in more detail how dynamic framed aspects can be used to overcome the problems encountered. Section 5 will then describe other related work and how this work will differ from them. Finally, section 6 concludes this report and summarises it.

2. Framed Aspects

Framed aspects are the amalgamation of frame technology [3] with AOP. Frame technology, which has its origins in the late 1970s, provides a mechanism for creating reusable components by way of meta-variables, code templates, conditional compilation, parameterisation, generation and a specification from a developer. Generalising code and assets in this manner allows them to be reused in different contexts making frames ideal for use in the generation of code libraries and software product lines. [18] presents a language independent XML based implementation of frame technology and has been used in the creation of product lines as diverse as city guide systems [19] and UML documents [8]. Typical examples of commands in frames are <*set*> (sets a variable), <*select*> (selects an option), <*adapt*> (refines a module with new functionality) and <*while*> (creates a loop around repeating code).

2.1 Problem with frames

Frames by themselves cannot encapsulate crosscutting concerns effectively, thus future evolutions can lead to changes across frames, effectively limiting the longevity of systems and components and giving rise to architectural erosion [17]. Frame technology requires that variation points and compositions are done explicitly in the code; this can lead to hard to read code. Utilising frame technology with a suitable AO language minimises this disruption by allowing system features, which are often crosscutting, to be encapsulated within a single module. This encapsulation will ease the evolution of the crosscutting concerns, and combining this with the configuration and generalisation properties that frames provide will ease the evolution process further still.

2.2 Using framed aspects

The Lancaster Frame Processor, a simplified adaptation of frame technology which is strongly influenced by XVCL, has been designed to be used with AO from the ground up and allows features to be composed together in a non invasive manner. [11] describes how a simple cache component can be generalised using frames in order to make it reusable.

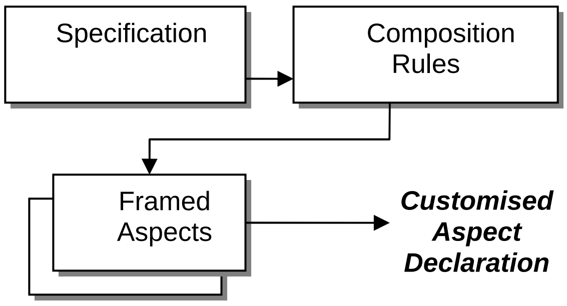


Fig. 1. Framed aspect composition

- Framed Aspect – generalised aspect code (via conditional compilation, parameterisation etc.).
- Composition Rules **-** contains possible legal aspect feature compositions, combinations, constraints. The rules are also responsible for controlling how these are bound together.
- Specification **- c**ontains the developer's customisation specifications. Frame commands will consist of setting of meta variables and selecting various options. The developer will usually take an incomplete template specification and fill in the options and variables s/he wishes to set.

In the framed aspect approach requirements elicited from the analysis phase are modelled into a feature graph based on FODA [9]. From the feature graph it is possible to delineate frame boundaries by following rules based on mandatory, alternative and optional features of a system or component and from this we can create the aspect code. Variation points or 'hotspots' are identified and the aspectual code is generalised with a suitable frame construct. Constraints and valid/invalid combinations of features are modelled in the composition rules module while the specification module supplies a custom specification from the developer. [12] explains the framed aspect process and methodology in more detail.

3. Dynamic Software Evolution and Dynamic AOP

This section will outline the use of dynamic AOP to implement a system capable of dynamic evolution, describing the properties this type of system needs to posses and why dynamic AOP is suitable for their implementation. First, we will describe how dynamic software evolution can be implemented and the benefits it will bring.

3.1 Dynamic Software Evolution

As already stated, software evolution is an inevitable task that has to be performed when developing a system of any notable size. The aim of the proposed framework is to ease some of the burden that this task requires.

In order to do this we aim to automate some parts of the evolution process by allowing the system itself to decide which available evolution steps are required to keep the system operating as intended. As mentioned previously these evolution steps will be encapsulated using dynamic AOP, with framing techniques used to parameterise them.

Based on information collected at run-time (collected by monitoring modules regarding the current environmental conditions) the system will be able to make decisions about which evolution changes, if any, are required. The information gathered will also be used to customise the aspect to suit the current system and conditions.

Automating this process will provide many benefits to system operators, from improved system availability to lowered running costs. Since such systems can maintain themselves dynamically, they do not have to be taken off-line for re-configurations to be applied. Also, as human contact is reduced, system availability is improved as human error is the most common cause of system failure [4]. Furthermore, as fewer maintenance staff are required, the running costs are lower.

Obviously there are some limitations as to what the proposed framework will be able to achieve. For example, the evolutionary changes must be already be thought of and pre-programmed but the key point is that it will be the system that decides when and where the changes should be applied. If the correct selection of evolutionary changes exists in the framework they should be applicable to a number of different systems and scenarios. This will reduce development time and improve reuse.

Large scale changes, such as when entire business goals change will have to be managed and applied manually as this is beyond the scope of our framework. We will be focussing on small scale perfective maintenance such as switching between algorithms to maintain optimum performance and introducing various functional and non-functional concerns as they are required.

3.2 Dynamic AOP

AOP aims to improve the areas where Object-Oriented Programming (OOP) fails by allowing concerns that would normally crosscut a number of objects to be cleanly encapsulated in a single element. AOP introduces three concepts: aspects, advice and joinpoints. Advice is used to implement the crosscutting concern, joinpoints specify the points in the base-code where the advice should be applied and aspects are used to encapsulate the advice and joinpoints.

Dynamic AOP techniques allow aspects to be woven while the system is being executed (either at class-load time or run-time). This provides a variety of useful features such as being able to introduce entirely new aspects and removing aspects already woven. However, certain problems can also arise such as lower performance, compatibility issues and security issues.

The majority of concerns that will require evolution will tend to be crosscutting and so dynamic AOP will be suitable for this kind of implementation. The following properties of dynamic AOP have been identified as being fundamental for implementing a system able to evolve automatically and dynamically, detail of these can be found in [6]:

- Apply adaptations dynamically
- Easily remove adaptations
- Encapsulate adaptations
- Implement fine-grained changes
- Apply adaptation to various points in the system

There are currently numerous dynamic AOP techniques such as AspectWerkz [2], JAC [13] and Prose [15]. The majority of these techniques support all of the above properties but to varying degrees. Whichever technique we choose, we will inevitably have to extend it in order to overcome a number of issues; see [6] for a comprehensive list of these problems. The issues that this paper aims to address are:

- Customising aspect behaviour – developing a mechanism for customising an aspect to suit the current needs of the system. The run-time and environment conditions of the system will vary greatly over time; the aspect should be adaptable to meet these needs.
- Re-use – as well as being customisable to suit the current conditions, the aspects should be applicable to a variety of systems. Each system may have methods which perform similar actions but the differing method and fields names may prevent an aspect which is suitable to alter the actions of these methods from being applied. The use of framed aspects will overcome these problems.

4. Dynamic Framed Aspects

From the earlier description of framed aspects it is clear that they allow the customisation of the behaviour of the aspects and also improve the reuse of the aspects they implement.

However, the framed aspect specification is currently statically defined and the concrete aspects are created from the frames before the aspects are woven to the system, normally during the compilation phase. This process prevents changes being made to the framed aspects dynamically.

In order to implement dynamic framed aspects we propose the use of a dynamic AOP language to create the aspects and a mechanism to dynamically create/update the aspects depending on the current environmental conditions.

For our initial prototype we propose to use AspectWerkz as the dynamic AOP technique. The reason why AspectWerkz is chosen here is because it is the most familiar to us at this time. Prose was discounted due its limitations in the types of aspects it can create and its overall flexibility.

Parameterising aspects in AspectWerkz should not pose any problems as the structure they use is the same as standard Java classes, and Java classes have been successfully parameterised in the past using framed aspects. The architecture we envisage will be implemented is shown in figure 2.

AspectWerkz Runtime
The AspectWerkz runtime is responsible for weaving and unweaving the aspects with the application base-code. The AspectWerkz runtime must be able to receive aspects sent to it from the framed aspect server.

Dynamic Application
The dynamic application is that which is being controlled by the AspectWerkz runtime and the application monitor. Using the proposed architecture any application should be able to be made dynamic without requiring much modification to the source code.

Application Monitor
This module is responsible for monitoring the current context of the running application and then translating this into an aspect request that will be sent to the framed aspect server. Initially only physical attributes of the system will be monitored such as memory usage, network usage, hard-disk space and processor usage. More detailed context information can be gathered later, such as parameter values passed to methods, field values, etc.

Framed Aspect Server
The framed aspect server will receive aspect requests from the application monitor. From these requests the server will retrieve the framed aspect from a repository and then customise it to suit the request received. The concrete aspect will then be sent to the AspectWerkz server to be woven to the dynamic application.

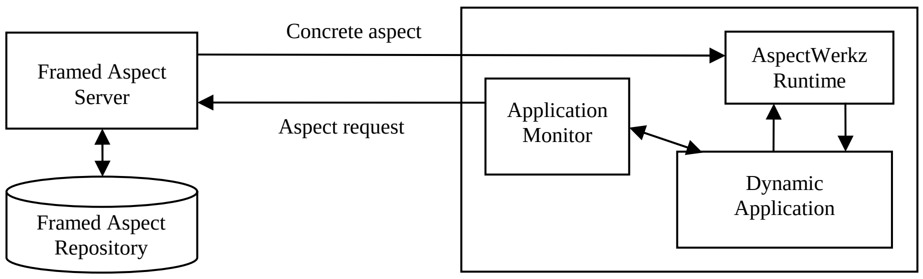


Fig. 2. The proposed architecture

Framed Aspect Repository
This will be used to store all the framed aspects. It is proposed that this will be implemented as a database structure to support querying and fast retrieval. It should also be possible to add new framed aspects to introduce new behaviour to the dynamic applications that was not available when the system was first started.

Aspect Request
The aspect request sent to the framed aspect server will contain a set of parameters detailing the current conditions of the system and the running environment. This data will be used to select an appropriate framed aspect and to create a concrete version of it.

4.1 Problems

This section will briefly introduce some of the initial problems that we anticipate when developing such a framework.

Applying Framed Aspects to the Base-System
One of the initial problems that we need to overcome is finding a way to mark where each aspect can be applied in the system. Each framed aspect implements a different crosscutting concern and these concerns may not be applicable at all points in the system. For example, a caching aspect does not need to be applied to a method that only prints a message to the user.

The solution we propose for this problem is to allow the programmer to specify a configuration file which will allow them to specify all the potential aspects that are relevant to the system and all the places where they could be applied.

However, this could result in complications and a long configuration file being specified. To reduce this problem, aspects could be grouped according to the types of changes they apply. This will allow the programmer to list types of aspect rather than each individual one. Also a GUI would aid the user in the creation of this file.

Framed Aspects Structure

For this solution to work successfully, each framed aspect will have to be created to follow certain conventions so that they can be applied to a variety of different systems and to make their structure predictable. Currently, framed aspects are largely unstructured and can be used to parameterise any part of the aspect. A more constrained structure is required so that the dynamic application can be created to accommodate all the potential aspects that may be applied to it. If the system has already been created then it may need refactoring to be made suitable for the aspects to be used.

Monitoring

The way the framed aspects have been parameterised will affect the information that is to be collected. We have to be sure that the information collected from the application monitor will be able to 'fill-in the gaps' of the framed aspects. From the information passed to the framed aspect repository a decision will have be made about which is the best framed aspect to use. It is not vital for this decision to be correct as the monitor will be able to use the data it has collected to check the aspect has had the desired effect; a new aspect can be requested if it has not. The above-mentioned configuration file, used to specify where the aspects need to be applied, will also be used to determine where and what needs monitoring.

4.2 Example

Suppose a client-server application is having performance problems. The monitoring module on the client detects a particular method that is experiencing network congestion when communicating with the server. From the configuration file, the system knows that introducing a cache may solve the performance problems. The monitor sends a request for a caching aspect, this specification includes what needs to be cached and the size the cache should be (this depends on the amount of resources available). This information is sent so that an aspect can be created with the correct data types and so that it does not use too many resources caching data. The aspect server will then create and compile the aspect and send it back to the client where it will be woven with the base-system.

This example, although simple, demonstrates how such a system would operate and how it could evolve its behaviour depending on the current system properties. Obviously in reality the aspects would have more than two parameters that need to be collected and sent to the server, but the principles are the same as in this simple example.

5. Related Work

Hot Swap [1] is an autonomic system developed by IBM. A monitoring component is used to examine the environment of the Hot Swap system and then requests adaptations to be made. The limitation of Hot Swap is only whole components can be replaced; AOP will allow us to make much more fine-grained changes. Additionally the replacement components used in Hot Swap need to be entirely pre-programmed, whereas the aspects in our solution can be dynamically re-programmed to suit the current environmental needs of the system.

An alternative way system behaviour could be modified is to use a technique called adaptive code. Adaptive code can be implemented in a variety of ways, from using parameters that are modified at run-time to alter the behaviour, to implementing a number of alternative algorithms to perform a certain task and the most appropriate can be selected at run-time (see [5]). The biggest drawback of using adaptive code is that it is not possible to insert new code or new monitors at run-time.

In the literature, the implementation of a system which possessed a certain degree of dynamic behaviour has also been achieved through the use of Prose [15]. MIDAS [14] was created to allow the distribution of aspects in a mobile environment. Whenever a node entered a particular location, aspects were distributed to it so it would behave correctly for that particular location. This is a limited solution in that the system is only location aware; we aim to create a framework that can be used in a variety of scenarios.

6. Conclusions

This paper has proposed a framework which will allow a system to be developed that will be able to evolve using a combination of framed aspects and dynamic AOP. When combined, the properties these technologies posses will allow aspects to be developed with a high degree of flexibility and they will be able to be applied dynamically to the system. In previous work the problem of reuse and customising the aspects to suit a particular scenario was highlighted as an issue to be resolved; the use of framed aspects will achieve this. This work is still in the early stages of development but the benefits of using framed aspects in this way are clear and will ease the development and implementation of dynamically evolvable systems.

References

[1] Appavoo, K. et al, “Enabling Autonomic Behaviour in Systems Software with Hot Swapping”, IBM Systems Journal Vol. 42 No. 1, 2003.

[2] AspectWerkz, “AspectWerkz Dynamic AOP for Java Overview”, http://aspectwerkz.codehaus.org/, 2004.

[3] Basset, P. “Framing Software Reuse – Lessons from Real World”, Yourdon Press Prentice Hall, 1997.
[4] Ganek, A.G., T. A. Corbi, “The Dawning of the Autonomic Computing Era”, IBM Systems Journal Vol. 42 No. 1, 2003.
[5] Glass, G., Cao, P., “Adaptive Page Replacement Based on Memory Reference Behaviour”, Measurement and Modelling of Computer Systems pp. 115-126, 1997.
[6] Greenwood, P., Blair, L., “Using Dynamic AOP to Implement an Autonomic System”, Dynamic Aspects Workshop AOSD 04, 2004.
[7] Horn, P., “Autonomic Computing: IBM’s Perspective on the State of Information Technology”, 2003.
[8] Jarzabek, S. and Zhang, H. “XML-based Method and Tool for Handling Variant Requirements in Domain Models”, 5th IEEE International Symposium on Requirements Engineering pp.166-173, 2001.
[9] Kang, K. C. et al, “Feature Oriented Domain Analysis (FODA) Feasibility Study”, Technical Report CMU/SEI-90-TR-21 Software Engineering Institute Carnegie Mellon University, 1990.
[10] Kiczales, G., et al, “Aspect Oriented Programming”, Proceedings ECOOP '97, 1997.
[11] Loughran, N. et al, “Supporting Product Lines Evolution with Framed Aspects”, ACP4IS Workshop AOSD 04, 2004.
[12] Loughran, N., Rashid, A., “Framed Aspects: Supporting Configurability and Variability for AOP”, ICSR-8, 2004.
[13] Pawlak, R. et al, “JAC: A Flexible Solution for Aspect-Oriented Programming in Java”, Reflection 2001, 2001.
[14] Popovici, A., Frei, A., Alonso, G., “A Dynamic Middleware Platform for Mobile Computing”, Middleware 2003, 2003.
[15] Popovici, A., Gross, T., Alonso, G., “Dynamic Weaving for Aspect-Oriented Programming”, AOSD 2002, 2002.
[16] Sommerville, I., “Software Engineering 5th Edition”, Addison Wesley, 1997.
[17] Van Gurp, J., Bosch, J., “Design Erosion: Problems and Causes”, Journal of Systems and Software Vol. 61 Issue 2, 2002.
[18] Wong, T.W. et al, “XML Implementation of Frame Processor”, Symposium on Software Reusability SSR 01 pp. 164-172, 2001.
[19] Zhang, W., et al, “Reengineering a PC-based System into the Mobile Device Product Line”, International Workshop on Principles of Software Evolution (IWPSE),2003.

Evolvable Pattern Implementations need Generic Aspects

Günter Kniesel[1], Tobias Rho[1]
and Stefan Hanenberg[2]

[1] Dept. of Computer Science III, University of Bonn
Römerstr. 164, D-53117 Bonn, Germany
{gk,rho}@cs.uni-bonn.de
[2] Dept. of Comp. Science and Inf. Systems, University of Duisburg-Essen
Schützenbahn 70, 45117 Essen, Germany
shanenbe@cs.uni-essen.de

Abstract. Design patterns are a standard means to create large software systems. However, with standard object-oriented techniques, typical implementations of such patterns are not themselves reusable software entities. Hence, providing typical implementation of such patterns and connecting them to a piece of software needs to be done by hand which is an error-prone process. Aspect languages have the potential to change this situation, due to their ability to encapsulate elements that crosscut different modules. Still, existing aspect languages can only express a small number of typical patterns implementations in a generally reusable way. In this paper, we point out the limitations of known aspect-oriented languages, define the notion of a generic aspect language (generAL) and argue that generic aspects are a natural way to achieve reusable design pattern implementations. We sketch the main features of one particular generic aspect language, LogicAJ, and show how it enables generic implementations of recurring design pattern implementations. In particular it permits to switch easily from one pattern implementation variant to another, which significantly eases the evolution of software.

1 Introduction

Design patterns [4] are a standard means of creating large software systems. Catalogues like for example [4, 1, 13] permit developers to benefit from the successful application of certain design elements for a given problem. This increases the quality of software, its comprehensibility and maintainability. The application of a design pattern usually results for a given programming language in a number of typical implementations. However, with standard object-oriented techniques, such implementations are not themselves reusable software entities. Hence, applying a certain design pattern typically means hand-crafting a number of software elements and embedding them into the application, which is a error-prose process. For example, even the typical implementation of a relatively simple design pattern like Singleton in Java is not trivial (cf. [5], pp. 127-133). Furthermore, pattern implementations typically require the developer to perform invasive changes of a number of classes in the system.

Typical implementations of design patterns provide a number of anticipated variation points: for example the Decorator design pattern [4] permits to add easily new functionality to an already decorated object. However, an unanticipated evolution of the underlying application is not that easy, because the application's evolution needs to be synchronized with the evolution of the design pattern implementations. This kind of needed synchronization is known as the problem of *co-evolution of design and implementation* [16]. Furthermore, changing the implementation of a certain design pattern to an alternative implementation is not easy: for example changing from a class-based Adapter implementation [4] to an object-based adapter implementation requires a number of invasive changes in the code. Such changes are also needed if the developer decides to the replace a certain design pattern. For example, the replacement of a Decorator implementation with a class-based adapter requires a number of changes in the code.

Aspect-oriented software development [9] is a promising approach to tackle the previous problems. The application of aspect-oriented languages promises modular implementation of typical design patterns, thus reducing the need for invasive non-local changes. However, it turns out that current aspect-oriented languages do not live up to this promise. For example [6] shows that only a number of typical design pattern implementations from the catalogue in [4] can be implemented in a reusable manner in the aspect-oriented language *AspectJ* [10]. [7] discusses two typical examples of design pattern implementations that cannot be implemented in a reusable way in known aspect languages and proposes *Sally* that supports genericity for aspect-oriented languages.

In this paper we argue for the need of a much stronger degree of genericity to fulfill the promise of evolvable pattern implementations. In support of our thesis we present the generic aspect language LogicAJ [15], and show how its genericity mechanisms enable reusable and easy to evolve implementations of design pattern variants.

2 Evolvable Pattern Implementations – Problem

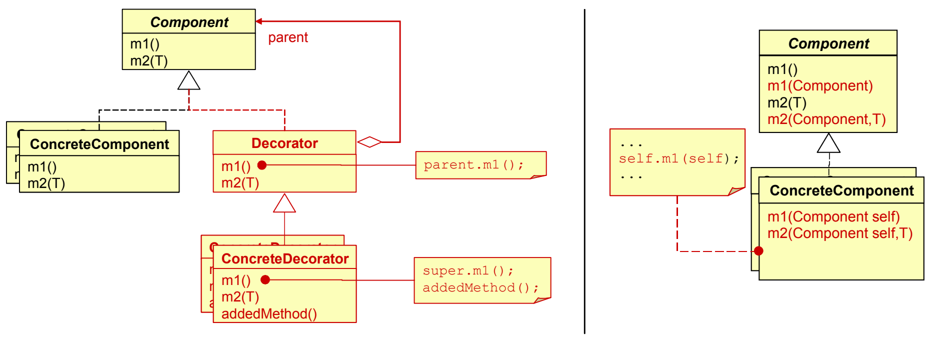
The problem that we address is the evolution of "patterned" designs and implementations. In this section we introduce two variants of the decorator pattern and discuss the problems that arise if a design starts with the first variant and must later be evolved into the second one. A solution based on the design and implementation of pattern variants as generic aspects in LogicAJ is presented in the rest of the paper.

2.1 Example: Variation of Decorator-Based Designs

The decorator pattern provides a flexible alternative to subclassing. Additional functionality can be added to an object dynamically. **Fig. 1** shows the UML diagram for the basic variant of the decorator[1]. There are four participant roles in the decorator

[1] The variant of the decorator pattern listed in [4] uses inheritance to achieve subtyping between the Decorator and the Component at the expense of undesired inheritance of state. Here, we

pattern: *Component* defines an interface for the objects to which additional functionality should be attached. *ConcreteComponent* classes implement such objects. The *Decorator* aggregates a *Component* instance and implements the *Component* interface by forwarding messages to this instance. The *ConcreteDecorator* adds functionality to *Component*.



```
parent.m1();
```

```
super.m1();
addedMethod();
```

```
...
self.m1(self);
...
```

Fig. 1 a) Basic Decorator Pattern according to [4] b) Subsequent evolution of Component. In the following sections the parts in red will be implemented by a generic aspect.

Now assume that, in an application built using the decorator pattern, new requirements imply the additional need to override the behaviour of a component on a per-instance basis. This means, that if some other object has own implementations of the component's methods that implementations should be used. One possible solution is to use *back-reference*s. These can be implemented as an additional parameter (cf. [11, 8]): every method inside the component gets a new parameter `self` and every (explicit or implicit) occurrence of `this` is replaced by `self`[2]. The corresponding design of *Component* is illustrated in Fig. 1b.

However, this design decision implies a synchronization of the design pattern implementation with the new design of *Component*. Fig. **2** illustrates how the whole decorator hierarchy needs to be adapted. Comparison of Fig. **1**a and Fig. **2** shows that the move from the basic decorator to the one with back references involves extensive changes in the design and implementation:

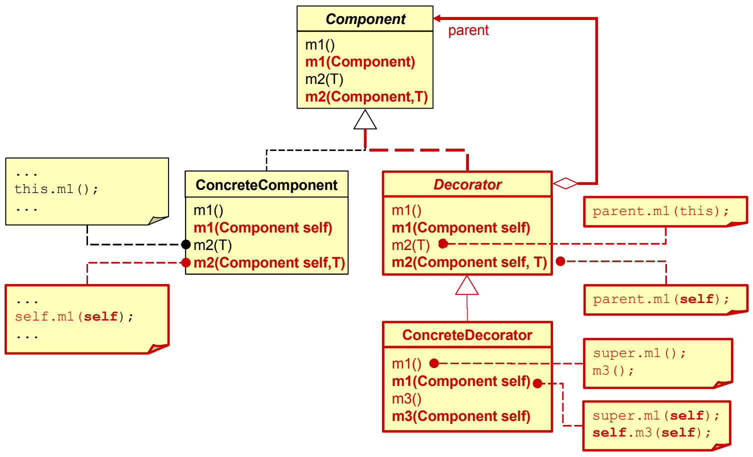
1. Every method in the *Component* interface must be extended by an additional parameter, *self*, for the back reference to the forwarding object. We call such methods *delegatee methods*.
2. Delegatee methods must be included in all subtypes of *Component*. In their body, all messages to *this* must be replaced by messages to *self*.

use a more general variant, in which we only assume that Decorator is a subtype of Component. In Java, this is achieved by implementation of the Component interface. In other languages it would require inheritance from a purely abstract class.

[2] For a thorough discussion of the issues involved in a simulation of object-based overriding via back references see [12].

3. Invocations of original methods must be replaced by invocations of delegatee methods *throughout the program*.
4. For the sake of clients that cannot be adapted, every original method must be preserved.
5. Forwarding methods in *Decorator* now must pass the correct value of the back reference (either *this* – in an original method, or *self* – in a delegatee method).

The result of this simple unanticipated evolution is a number of invasive changes in a number of different modules: large effort is necessary to keep existing design pattern implementations in sync with other design decisions. In practice, it is almost impossible to guarantee such a correct synchronization.



```
...
this.m1();
...
```

```
...
self.m1(self);
...
```

```
parent.m1(this);
```

```
parent.m1(self);
```

```
super.m1();
m3();
```

```
super.m1(self);
self.m3(self);
```

Fig. 2 Decorator Pattern with back reference implemented as an additional method parameter. The parts in red are generated by the aspects described in Section …

3 Towards a Solution – Patterns and Aspects Together

3.1 Patterns describe generic collaborations

Design patterns are an effective way to reuse design knowledge. In a highly standardized way, patterns describe design problems, their context, the forces that influence potential solutions, a generic solution idea and possible variations of its implementation. These descriptions are typically provided in two forms, as concrete examples and as generic abstractions of designs and related object interactions. The generic descriptions of solutions are highly parametric. The names of classes, fields, and methods presented in design pattern “solution” sections are never meant literally but as indicators of *roles* that the respective entities play within the pattern. Correspondingly, object interaction diagrams and related method code refer to such role names.

In order to apply a pattern, a designer must first identify the entities of an application that correspond to the roles mentioned in the pattern. Then he may instantiate the solution, replacing roles by names of concrete entities.

Thus, pattern solutions can be viewed as *generic descriptions of collaborations that are parameterized by the roles of the entities mentioned in the pattern*. Correspondingly, a particular application of a pattern in a given software system is an instantiation of the generic collaboration.

3.2 Aspects describe collaborations

Aspects come in different shapes and sizes. Put differently, there is a wide variety of concepts and systems that call themselves aspect-oriented. Their commonality is that all provide ways to express “crosscutting” structure and behaviour in one single module called an aspect, filter, hyperslice, etc. Structure and behaviour encapsulated in an aspect can range from omnipresent homogeneous behaviour (like logging) to very specific behaviour limited to a particular set of cooperating classes.

The latter case is the one relevant to patterns. Aspect languages have the potential to express the essence of pattern implementations, due to their ability to encapsulate collaborations. For instance, Hannemann and Kiczales [6] show how 12 of the patterns from [4] can be expressed in AspectJ in a reusable way.

However, 11 other GoF-patterns have no reusable implementation in AspectJ, including often used ones like FactoryMethod, Abstract Factory, Builder, Bridge, Adapter, Decorator, Proxy and State. Moreover, some of the reusable versions are limited in different ways. For instance, Hanenberg and Unland [7] show that even for simple patterns, like Singleton, the reusable AspectJ implementation does not achieve the same effects as a manual implementation of the pattern.
In addition, many solutions are not generally reusable but specific to a particular instantiation of an aspect. Part of the aspect code must be repeated for different instantiations.

Hanenberg and Unland identify the inability to express *context-dependent* introductions as the reason for the problems. As a remedy, they propose “parametric introductions”, implemented in the aspect language Sally. They show how parametric introductions improve the implementation of patterns like Singleton and enable reusable, partial implementations of patterns like Decorator, which could not be handled at all with AspectJ.

4 Generic Aspects to the Rescue

In this paper, we start from the observation that the concept of parametric introductions is basically a first step towards a generic aspect language but still too restricted to provide a *general* solution for reusable pattern implementations. This leads us to the conclusion that genericity must be supported uniformly across an aspect language, not just for member introductions.

We define the notion of a *generic aspect language* and sketch the main features of one particular generic aspect language, *LogicAJ*. Then we show how a pattern that could not be expressed in non-generic aspect languages can be implemented easily. The example of the Decorator pattern serves to illustrate basic functionality missing from previous approaches, when it comes to reusable and easy to evolve implementations. Since the same basic functionality is required for other patterns as well, our arguments can be generalized.

4.1 Generic Aspect Languages

The common cause of the above-mentioned limitations of AspectJ-based pattern implementations is that AspectJ does not provide genericity. Typically, the AspectJ solutions refer to *fixed* names for concrete entities of the base program, where a reusable pattern implementation would require *role* names that can be bound to concrete entities when the pattern (resp. aspect) is instantiated. Therefore, the step to reusable pattern implementations is the step to generic aspect languages.

A *generic aspect language* allows aspects to use logic variables that can range over syntactic entities of the host language[3]. Depending on the host language, these can be syntactic elements of Java, C++, C#, etc. or abstractions of messages or events.

Generic aspect languages can provide different *degrees of genericity*, depending on the range of host language entities that they can match. A *fully generic* aspect language would provide logic variables hat can range over *all* syntactic entities of the host language. In a Java-based fully generic aspect language, for instance, logic variables could match anything from packages and types down to individual statements, modifiers and throws declarations.

Logic variables have two properties that are essential in our context. First, their values cannot be manipulated explicitly. Instead, they are bound to values by the evaluation of particular conditions. Conditions that can bind logic variable values are contained, for instance, in the joinpoint expressions of AspectJ, hyperslice expressions of HyperJ, and filter expressions of Compose*..

Second, every mention of the same logic variable name within a scope represents the same value. Thus it is possible to refer later to a value matched earlier. In particular, it is possible to create new code based on previous matches. In this respect, logic variables are more expressive than "*" pattern matching (e.g in AspectJ), where two occurrences of "*" do not represent the same value.

The following presentation of LogicAJ illustrates the general principles introduced so far.

[3] Every aspect language has (at least) one *host language*. This is the language in which the modules to which the aspects refer are written. For instance, the host language of HyperJ, AspectJ, Sally, and LogicAJ is Java. Aspect languages like AspectC++ and AspectC# have C++ and C# as their target languages. The composition filter model and its incarnation in the aspect language Compose* is applicable to many different host languages. Event based AOP is an even more generic model, whose host language is an abstract language of events that can be mapped to any particular object-oriented programming language.

4.2 LogicAJ

LogicAJ is an extension of AspectJ [10] by the above concepts. In LogicAJ, logic variables can range over Java packages, types, fields, and methods, including method signatures and method bodies. Put differently, logic variables can be used in any place where in AspectJ aspects it is legal to use packages, types, fields, and methods. Syntactically, logic variables are denoted by names starting with a question mark "?"[4].

Generic aspects are particularly useful for expressing crosscutting changes that follow a common structure but differ in the names of created or modified entities. A simple example is the use of mock objects. This is a common technique for narrowing down the potential sources of a failure during testing. Its essence is the replacement of some of the tested classes by mock classes, which provide a fixed expected behavior during tests.

```
usrMng = new UserManager(...);
...
dbMng = new DBManager(...);
```

```
usrMng = new UserManagerMock(...);
...
dbMng = new DBManagerMock(...);
```

Below we show a *LogicAJ* implementation of mock objects. The aspect replaces each constructor call with a call to the respective constructor of the associated mock class, if the mock class exists.

```
aspect MockAspect {
      Object around(?mock, ?args, ?class) :
            // Intercept constructor invocations.
            // Bind ?class to the name of the instantiated class
            // and ?args to the argument list of the invocation
            call(?class.new(..)) && args(?args) &&
            // Check if a class with name ?class+"Mock" exists
            concat(?class, "Mock", ?mock) && class(?mock)
      {     // return instance of mock class
            // (includes weave time check for constructor existence)
            return new ?mock(?args);
      }
}
```

The example illustrates the syntax of logic variables, their binding by the evaluation of conditions, and their use in the assembly of generic advice code. The pointcut part of the advice uses three predicates, *call*, *args* and *concat*. The *call* predicate is basically the *call* pointcut of AspectJ. The *args* predicate is an extension of the *args* pointcut of AspectJ in that the logic variable ?*args* passed as an argument to the pointcut can match an entire argument list. Here it matches all arguments of any constructor invocation. The semantics of the *concat* predicate is that the third argument is the concatenation of the first and the second. It is used here to create names of mock classes by appending the suffix "Mock".

Logic variables also enable generic introductions. With generic introductions, the members to be introduced and the types into which they should be introduced can be determined by the evaluation of predicates. It is also possible to introduce new types. Generic introductions basically generalize the concept of advice in AspectJ. This is

[4] LogicAJ shares this syntax with Sally [7] and TyRuBa [2, 3].

reflected in their syntax, which is largely advice syntax except for the keyword "introduction" instead of "advice". We will see various examples of generic introductions in the next section.

5 Evolvable Decorator Pattern Implementation – Solution

In this section we show how the two variants of the decorator pattern introduced in Section 2 are modelled in the generic aspect language LogicAJ in a modular and reusable way. As we proceed, we identify different classes of limitations of previous approaches and show in each case how they are overcome in LogicAJ. The examples underpin our conclusion that "evolvable pattern implementations need generic aspects" and motivate our definition of generic aspect languages.

5.1 Basic Decorator Pattern in LogicAJ

This section models a reusable basic variant of the decorator pattern implementation illustrated in **Fig. 1**. In the following, we assume that an interface playing the *Component* role and classes playing the *ConcreteComponent* role exist[5], are implemented in plain Java and are used as base classes for the aspects. The class playing the role of *Decorator* is generated by an abstract aspect AbstractDecorator. Classes playing the *ConcreteDecorator* role are created by concrete aspects derived from AbstractDecorator.

5.1.1 The AbstractDecorator Aspect

This aspect has a double function, as a repository of shared pointcut definitions and as the place where the classes playing the role of decorator are created (**Fig. 4**).

The decorator pattern can be instantiated multiply in an application. Every instantiation is specific for a particular *Component* type. In order to express this dependency, the class playing the role of the *Decorator* in a particular pattern instantiation is generated based on the participant *Component*. Its name is determined by adding the postfix Decorator to the name of the interface that plays the *Component* role. This dependency is abstracted in the decorator pointcut. The abstract pointcut component must be implemented in a concrete aspect, in order to trigger application of the aspect to a particular part of the program.

[5] The Component interface could be automatically created by an extract interface refactoring on the Concrete Component classes.

```
abstract aspect AbstractDecorator {

 abstract pointcut component(?component);

 pointcut decorator(?decorator) :
    component(?component) &&
    concat(?component, Decorator, ?decorator);

 introduce(?component, ?decorator) : ... // see Fig. 4

 introduce(?decorator, ?rettype, ?params, ?name) : ... // see Fig. 5
}
```

Fig. 3 `AbstractDecorator` aspect. Participant roles are abstracted as pointcuts. A *Decorator* class specific for a particular *Component* type is generated by two generic introductions.

5.1.2 Generic Type Introduction for *Decorator* Role

The AbstractDecorator aspect creates a class playing the role of the *Decorator*. This is done via a generic type introduction as shown in **Fig. 4**. Based on the above pointcuts, the name of the *Component* and *Decorator* is determined and bound to the logic variables ?*component* and ?*decorator*. Then the ?*decorator* class is generated and declared to be a subtype of ?*component*. It contains an instance variable parent of type ?*component* to which messages can be forwarded.

```
introduce(?component, ?decorator) :
    component(?component) && decorator(?decorator)
{
   abstract class ?decorator implements ?component {
       protected ?component parent;
   }
}
```

Fig. 4 Generic type introduction

Note that the generic type introduction, that is, the ability of creating types in an aspect and defining their name depending on the current context[6], is essential for our example. We know of no other aspect language that provides this functionality.

5.1.3 Generic introduction of forwarding methods

The next step is the creation of one forwarding method in the *Decorator* class for every method in the *Component* type[7]. This is done via a generic introduction. In its condition part, it checks for methods that exist in the Component type. Then it creates methods with exactly the same signature in the *Decorator* class. Every created method forwards its invocation to the object reachable via the parent reference. The

[6] In this case, the context is determined by the value of ?*component*.

[7] For simplicity, we often identify roles with the classes that play the roles, when the meaning is clear from the context. For instance, we simply say *Decorator* class instead of class that plays the *Decorator* role. For the same reason, we use the role names as class names in all diagrams.

method creation process is repeated for all values of logic variables that make the condition part (the pointcut) true. Thus in the end, the *Decorator* class will have one forwarding method for every method from the *Component* interface.

```
introduce(?decorator, ?rettype, ?methodName, ?params) :
  component(?component) &&
  decorator(?decorator) &&
  method(public ?rettype ?compnent.?methodName(?params))
{
  ?rettype ?decorator.?methodName(?params) {
     parent.?methodName(?params);
  }

}
```

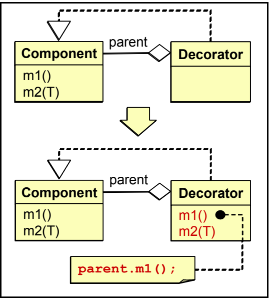


Fig. 5 Generic introduction of forwarding methods.

Without generic introductions it is not possible to write *one* introduction that creates *different* methods depending on the context. In AspectJ, for instance, one needs to know the complete signature of the methods to be introduced when writing the aspect. Thus, [6] concludes that no reusable implementation of decorator is possible with AspectJ. The concept of parametric introductions in [7] is very similar to generic introductions. It allows creation of methods with heterogeneous signatures and homogeneous implementations in different contexts. However, the ability to create context-dependent signatures *and* implementations, used above, seems to be unique to LogicAJ.

5.1.4 Instantiation of the pattern

The variant of the decorator pattern implemented in the AbstractDecorator aspect is instanttiated by the creation of a concrete subaspect that supplies the missing pointcut definition and the implementation of *ConcreteDecorator* classes (**Fig. 6**).

```
aspect MyComponentDecorator extends AbstractDecorator {

  pointcut component(?component) : equals(?component, MyComponent);

  introduce(?decorator) : decorator(?decorator)
  {
     public class ConcreteDecorator1 extends ?decorator {
        public void m () { /* possibly calling super.m()  */ }
        public void m3() { /* possibly calling super.m3() */ }
     }
     // ... more ConcreteDecorator classes ...
  }
}
```

Fig. 6 Instantiation of the decorator aspect for `MyComponent` triggers the creation of the `My-ComponentDecorator` class and its subclasses

Since the implementation of the *ConcreteDecorator* classes consists almost entirely of plain Java code, one might wonder why we define them in an aspect. The reason is that in a base class that does not declare the extends relationship to the

?*decorator*, invocations of super would not compile. Bases classes that declare the extends relation, however, would be tightly dependent of the naming convention for decorators. The solution in **Fig. 6** preserves separate compilation and encapsulates the knowledge about decorators in the aspect hierarchy.

5.2 Decorators With Back References in LogicAJ

In section 2 we have shown that the manual evolution of an application that uses the above implementation variant of the decorator pattern can be extremely costly and error-prone. Now we show that the same evolution step can be achieved in LogicAJ incrementally, by the definition of further subaspects of AbstractDecorator. The main work is performed in the AbstractDecoratorBR aspect. It includes:

1. Creation of delegatee methods in *Component* and all its subtypes (via a generic method introduction).
2. Replacement of messages to *this* by messages to *self* in all delegatee methods (via a generic around advice).
3. Replacement of invocations of original methods by invocations of delegatee methods throughout the program (via a generic around advice).
4. Passing of the correct value of the back reference in *Decorator*: either *this* – in an original method, or *self* – in a delegatee method (via a generic around advice).

5.2.1 Creation of delegatee methods

For every subtype ?*sub* of *Component* and each method in that subtype, the generic introduction in **Fig. 7** adds to ?*sub* a method with exactly the same body but an additional first parameter, self, of *Component* type.

```
introduce (?comp, ?sub, ?rettype, ?name, ?params, ?body) :
     component(?component) &&
     subtype(?component, ?sub) &&
     method(?rettype ?sub.?name(?params) ?body )
{
     method(?rettype ?sub.?name(?component self, ?params))
         ?body
}
```

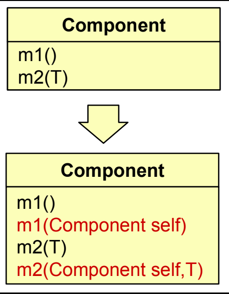


Fig. 7 Introduction of delegatee methods in all subtypes of *Component*

Note that in the above example it is essential that logic variables can also range over unnamed entities, in this case over method bodies. We need the ability to pass a value for the ?*body* variable from the method pointcut to the advice in order to copy the existing method body into the delegatee method.

5.2.2 Replacement of messages to `this`

Consistent use of the additional self parameter in the copied method body is enforced by the generic advice shown in **Fig. 8**. For all subtypes ?*sub* of *Component* it determines within delegatee methods all calls of original methods from *Component* that are invoked on this. These calls are replaced by calls on self, thus enabling execution of the respective method in the self object.

The advice uses two auxiliary pointcut definitions, withinDelegateeMethodInType and callOfComponentMethodOnThis. The first one defines that we are in a delegatee method if the method's first parameter has type *Component* and there is another method in the same type with the signature resulting from deleting the delegatee method's first parameter[9]. The second one selects all invocations of original methods from *Component* and filters those that are invoked on this, using the new pointcut receiverIsThis(). It checks whether the message receiver at the current joint point is the enclosing instance. The pointcut binds the name and arguments of the filtered method calls to the logic variables ?*name* and ?*args***.** These are used in the advice to generate the new code.

[9] In the real implementation one would use additional criteria, e.g. a particular naming scheme of delegatee methods, in order to avoid false matches.

```
around(?component self, ?name, ?args) :
   component(?component) &&
   subtype(?component, ?sub) &&
   withinDelegateeMethodInType(?sub, self) &&
   callOfComponentMethodOnThis(?sub,?name,?args)
{
   self.?name(?args);
}
```

```
m1(Component self) {
    this.m1();
    ...
}
```

```
m1(Component self) {
    self.m1(self);
    ...
}
```

```
pointcut withinDelegateeMethodInType(?type, ?type self) :
     delegateeMethod(?type, ?rettype, ?name, [?comp|?params]) );
     withincode(?rettype ?type.?name(?comp self, ..)) ;

pointcut callOfComponentMethodOnThis(?inType, ?name, ?args) :
     originalMethod(?rettype, ?name, ?params) &&
     call(public ?rettype ?inType.?name(?params)) &&
     receiverIsThis() &&
     args(?args) ;

pointcut delegateeMethod( ?sub, ?rettype, ?name, [?comp|?params]) :
     component(?comp) &&
     subtype(?component, ?sub) &&
     method(?rettype ?sub.?name(?params)) &&
     method(?rettype ?sub.?name(?comp self, ?params) );

pointcut originalMethod(?rettype, ?name, ?params) :
     component(?comp) &&
     method(?rettype ?comp.?name(?params)) &&
     method(?rettype ?comp.?name(?comp self, ?params) );
```

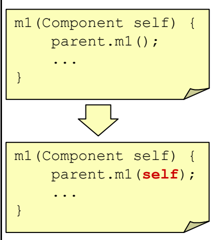
Fig. 8 Implementation of "self delegation" by replacing messages to "this".

5.2.3 Forwarding with passing of back reference

In order to ensure that the self back reference has the proper value, we must extend the code of the class that plays the *Decorator* role. Each of its forwarding methods must pass on the current value of self to the parent object. In **Fig. 9** the forwarding pointcut identifies all places where a method invokes itself on the parent object. The advice adds the self parameter to the forwarding invocation.

```
around(?comp parent, ?comp self, ?name, ?args) :
   forwarding(?comp parent, ?comp self, ?name, ?args)
{
   parent.?name(self, ?args);
}

pointcut forwarding(?comp parent, ?comp self, ?name, ?args):
   component(?comp) &&
   decorator(?decor) &&
      withincode(?rettype ?decor.?name(?comp self, ..)) &&
   call(public ?rettype ?decor.?name(?args)) &&
   target(parent) ;
```



```
m1(Component self) {
   parent.m1();
   ...
}
```

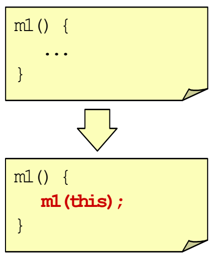
```
m1(Component self) {
   parent.m1(self);
   ...
}
```

Fig. 9 Forwarding with passing of back reference

5.2.4 Use of delegatee methods

Fig. 10 shows the redirection of the normal method execution to the execution of delegatee methods. The entire body of every original method is replaced by an invocation of the method's delegatee version, with `this` as the value of the first argument. This redirection is applied to every subtype of *Component*.

```
around(?decorator inst, ?rettype, ?name, ?args) :
   decorator(?decorator) &&
   subtype(?decorator, ?sub) &&
   originalMethod(?rettype,?name,?params) && //see Fig. 8
   execution(public ?rettype ?sub.?name(?params)) &&
   this(inst) &&
   args(?args);
{
   inst.?name(inst, ?args);
}
```



```
m1() {
   ...
}
```

```
m1() {
   m1(this);
}
```

Fig. 10 Redirection of normal method invocations to delegatee methods

The examples shown in **Fig. 8** to **Fig. 10** illustrate various cases in which availability of generic advice was essential in order to express the intended semantics.

5.3 Results

The example studied in this section shows that the adaptation of a decorator-based implementation to new requirements can be performed completely at the level of aspects. The only change that we need in addition to the implementation of the AbstractDecoratorBR aspect demonstrated above, is to turn the concrete subaspects of AbstractDecorator into subaspects of AbstractDecoratorBR. This ensures that after the next weaving, the back reference based implementation of the decorator pattern will be consistently available in the application instead of the basic one. No single line of code has to be changed manually in the base classes to achieve this result.

Looking back to our small case study from a language designer's point of view, we note that we needed the joint expressiveness of generic type introductions, generic member introductions, and generic advice. It was essential that logic variables ranged over all named entities of the host language (from types to arguments) plus some unnamed entities (in the case of method bodies, see Fig. 7).

We conclude that non-tangled, reusable implementations of patterns and other highly parametric concepts require genericity at almost *every* level of a language. Whether full genericity (not yet supported in LogicAJ) is desirable, and more generally, what is the "right" degree of genericity, are two of the open questions that we would like to discuss with the workshop participants.

6 Conclusions

Object-oriented design patterns are a standard means to create more dynamic and easier to evolve systems. However, with standard object-oriented techniques, pattern *implementations* are not themselves reusable software entities. Therefore, the evolution of a "patterned" design *beyond* its anticipated variation points can be arbitrarily difficult. Synchronizing the pattern implementation with changes in the design of the application, switching to another implementation variant for a particular pattern, and combination of multiple patterns within one application typically require extensive changes in a design and code base.

In this paper we have shown that *generic aspect languages* are a promising solution. Their characteristic is the use of *logic variables* for program entities (packages, types, fields, methods, parameter lists, argument lists, method bodies, etc.), which enables expressing generic transformations of a program which can be performed subject to generic conditions.

In particular, we described LogicAJ, a generic aspect language design for Java that provides logic variables ranging from types and packages down to the level of individual method invocations, method arguments and method bodies. Using the decorator pattern as an example we have demonstrated the expressive power of the resulting language design and in particular, how it fosters reusable and evolvable implementations.

7 References

[1] Buschmann,F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stahl, M: *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, 1996.

[2] De Volder, K.; D'Hondt, T.: *Aspect-Oriented Logic Meta Programming*, Pierre Cointe (Ed.): 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), Saint-Malo, France, July 19-21, 1999, LNCS 1616, Springer-Verlag, 1999, pp. 250-272.

[3] De Volder, K.: *Type-Oriented Logical Meta Programming*, PhD thesis, Vrije Universiteit Brussel, Belgium, 1998.

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

[5] Grand, M: *Patterns in Java*, Vol. 1, John Wiley & Sons, 1998.

[6] Hannemann, J.; Kiczales, G.: *Design Pattern Implementation in Java and AspectJ*, Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, November 2002.

[7] Hanenberg, S.; Unland, R.: *Parametric Introductions*, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, Boston, MA, March 17 - 21, ACM, 2003. pp. 80-89.

[8] Harrison, W.; Ossher, H.; Tarr, P.: *Using Delegation for Software and Subject Composition*. Research Report RC 20946 (922722), IBM Research Division, T.J. Watson Research Center, Aug 1997.

[9] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwing, J.: *Aspect-Oriented Programming*. Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997, pp. 220-242.

[10] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold William G.: *An Overview of AspectJ*, Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Springer-Verlag, 2001, pp. 327-353.

[11] Kniesel, G.: *Delegation for Java – API or Language Extension?*, Technical Report, IAI-TR-98-4, ISSN 0944-8535, University of Bonn, Germany, May, 1998.

[12] Kniesel, G.: *Dynamic Object-Based Inheritance with Subtyping*, PhD Thesis, Computer Science Department III, University of Bonn, Germany, July, 2000.

[13] Rising, L.: *The Pattern Almanac 2000*, Addison Wesley, 2000.

[14] Tip, F; Kiezun, A.; Bäumer, D.: *Refactoring for generalization using type constraints*. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA. pp. 13-26.

[15] Windeln, T.: *LogicAJ -- eine Erweiterung von AspectJ um logische Meta-Programmierung*, Diploma thesis, CS Dept. III, University of Bonn, Germany, August, 2003.

[16] Wuyts, R.: *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*, Phd Thesis, Vrije Universiteit Brussel, Belgium, 2001.