http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 6 (6): 879-884, 2007
ISSN 1812-5638

Design Issues in Aspect Oriented Programming

[1]Deepak Dahiya and [2]Rajinder K. Sachdeva
[1]iTech US Inc., South Burlington, Vermont 05403, US
[2]IIPA, Indraprastha Estate, Postal Code 10003, Delhi, India

Abstract: The actual realization of a design language form has been revealed to be non-trivial. Most of the design languages are very much tailored towards a specific application or application domain. Moreover, with the exception of very few hardly any of these language notations have been used outside their own research environment. The development of a more generic aspect oriented design language requires a wider and more thorough look at the requirements. This study examines the design notation issues and discusses how it fulfils the requirements in principle and design and consequently would lead to a general purpose AOSD design language (AOSDDL) that will map AOSD design notations to the existing AOP languages.

Key words: AspectJ, UML, concern, aspect, component, network

INTRODUCTION

This study discusses a design language AOSDDL (Aspect Oriented Software Development Design Language) architecture that attempts to reproduce the semantic of AspectJ in the UML and the same is proposed within this research.

It has become apparent that design language research deals largely with trade-offs. For example, many of the aspect oriented design systems trade-off implementation dependency for wide tool support or limited support with general purpose flexibility. Research into aspect oriented design languages so far has shown that no single solution will meet all possible requirements of aspect oriented software development and thus, multiple systems for domains with different demands must be able to co-exist and interoperate.

The challenge in designing aspect oriented solutions therefore is to draw the optimal line between trade-offs depending on the requirements at hand. For this, it is crucial to understand fully the requirements of a given domain (Banniassad *et al*., 2005; Rashid *et al*., 2003).

Tools environment: The Eclipse Platform for Java was used to carry out the implementation and testing of the abstract notations in AspectJ. To implement graphical notations and diagrams the Together CASE tool was used. The CASE tool Together from Borland is an enterprise development platform enabling application design, development and deployment. It is extensible through an open Java API offering the possibility to develop custom software that plugs into the Together platform in the form of modules. The open API is composed of a three-tier interface that enables varying degrees of access to the infrastructure of together.

AOSDDL notations: This study specifies an approach for AO modeling to address the specification of crosscutting concerns at the architecture level in order to maintain the separation of concerns at an early stage in the software development life cycle. A key intention is to offer standard development tool support and interchange ability among various CASE tools, thus an extension to UML was developed without changing its metamodel specification to achieve standard UML conformity. Using UML as a modeling language improves developer productivity and offers high acceptance, as it is the industry-standard modeling language for the software engineering community. When using standard UML for aspect-oriented modeling, developers do modeling by using familiar tools and environments to gain all the benefits they are used to in OO design. UML is an extensible modeling language that enables domain-specific modeling which raises its suitability as a modeling language for supporting aspect-oriented modeling.

Another important goal was to gain the benefits both of code and design reuse of AO software, including the ability to reuse aspect and base elements separately. Thus, aspects and base elements should be completely kept apart and independent of the implementation technology in order to simplify the replacement of the AO language. A clear separation of the language dependent crosscutting parts eases the support of many different AO languages and concepts. This study focuses on adopting AspectJ concepts for the implementation language dependent parts of AOSDDL. For the support of other AO concepts (such as Hyper/J) is considered

Corresponding Author: Deepak Dahiya, iTech US Inc., South Burlington, Vermont 05403, US

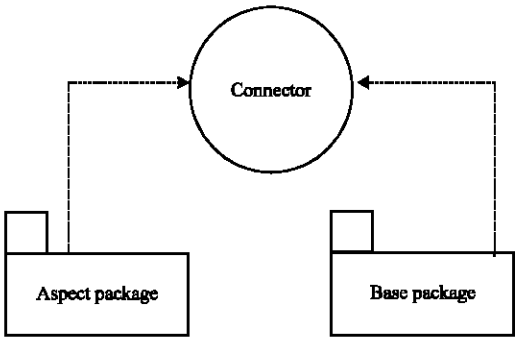

Fig. 1: Package level (De) composition

and part of some future research. AOSDDL considers the fact that crosscutting concerns tend to affect multiple classes in a system. Since a concern itself can consist of several classes and since all of these classes may be associated with the class the concern crosscuts, the module construct for a concern should be higher-level than a class. Otherwise associations modeled on class-level would supersede the logical grouping of the classes belonging to one concern. This would make the design models hard to read and lead to graphical tangling of crosscutting concerns instead of a clear separation.

Figure 1 provides an overview of the notation and its focus on using package and a connector. AOSDDL includes a base package (having the business logic), an aspect package (having the crosscutting concern) and a connector to link aspects and base elements. Being one of the most popular Aspect Oriented language, AspectJ has been used to describe and present the AOSDDL notation model. Both the aspect package and the base package are used to express any crosscutting concern that can occur and might affect the system. Further, they can contain any valid UML design construct that might be describing either the complete system or a part of the system based on Aspect Oriented Design. The aspect can be modeled as an individual entity or independently of any design it may potentially affect or be a part of. The connection between base design and aspect design is specified separately. Support of different AO technologies is therefore rather simple and straightforward, as it is only the connector's syntax that has to be changed. This connector will hide the details of the interaction between components. To model any design construct, the connector can be considered in terms of a client connector that communicates with the aspect packages via the<uses>relationship. The type of connector used to interconnect aspect components also influences the performance of component based systems. This kind of separation enables high degree of reusability of both the aspect and base elements since the connector notation (element) is the only crosscutting element. This way of focusing on UML notations and standard notation of packages as a single unit leads to design models that are easy to read, as they avoid *graphical tangling*. Additionally, the connector encapsulates the underlying implementation technology using AspectJ.

As described above, the AOSDDL notation can contain the following classes that conform to the concepts AspectJ offers for the specification of weaving rules:

- The Introduction class, which defines the rules for AspectJ's introduction mechanism.
- The Pointcut class, which defines execution points in the control flow of the program.
- The Advice class, which defines the code to be executed at the pointcuts defined in the Pointcut class.

All classes contain operations with special semantics to specify how aspect and base elements have to be recomposed. The complete syntax of the AspectJ specific connector will not be presented here, however a few examples described here, provide a macroscopic view of how the notation can be used and shows some of the most important constructs.

AOSDDL is a simple and powerful notation for aspect-oriented modeling. In order to reduce errors when mapping models to code and offer low-level architecture design support, the development of code generator is part of a future research.

Other design notations: The following issues were considered for modeling a general purpose AOSD design language (AOSDDL) with regard to a programming language namely, AspectJ and a standard Object Oriented design language namely, UML quite widely in use in the software industry:

- Mapping AOP to Aspect Oriented UML Extensions (Ho *et al*., 2002; Stein *et al*., 2002)
- Identifying Software Concerns (Bamiassad *et al*., 2002)
- Design Language Issues for Component Based software Development (Suvee *et al*., 2003)
- Mapping UML extensions through composition patterns to Aspects (Clarke and Walker, 2001)
- AspectJ Extensions for Distributed Computing (Shomrat yehudai, 2002)

In this study, we present the issue of identifying software concerns, mapping UML extensions through

composition patterns to Aspects, issues concerning component based software development and AspectJ extensions for distributing computing. The other issues are presented elsewhere. In general, they provide a broad outline that sums up the parameters for various concerns and scenarios prevalent in the software industry and how AOSDDL will address them under various forms.

Identifying software concerns: Separation Of Concerns (SOC) is a long-established principle in software engineering. It has received widespread attention in modem programming languages, with constructs such as modules, packages, classes and interfaces, which support properties such as abstraction, encapsulation and information hiding. SOC has also received attention in software architecture and design, with techniques such as composition filters and design patterns. While advances in all of these areas have had significant benefits, problems related to inadequate separation of concerns remain. This has led to work on Advanced Separation Of Concerns, (ASOC), aspect-oriented programming and multidimensional separation of concerns. These bring a number of innovative ideas to programming in particular and to software development in genera], which are now beginning to mature and coalesce under the heading of Aspect-Oriented Software Development (AOSD).

Although ASOC has been emphasized in recent work, concerns themselves have remained something of ignored. Current ASOC tools provide only limited support for explicit concern modeling, representations of concerns tend to be tied to particular tools or artifacts and concern modeling usually occurs just in the context of a particular type of development activity. such as coding or design. A global perspective on concerns, that spans the life cycle and is independent of particular development tools or artifacts, has been lacking.

Concern modeling schema frame work: During software development, concerns arise at all stages of the life cycle, from requirements specification, through design, coding and testing, to maintenance and evolution. Concerns also span multiple phases of the life cycle, relate to multiple instances and types of artifacts and crosscut phases and artifacts in different ways. Finally, concerns are dynamic and relative, that is, that the concerns relevant to a particular software unit will change over time and that they also depend on the perspective or purpose of the user or stakeholder who considers the software.

This implies that a general-purpose concern modeling framework that may be a part of the design language should support the representation of arbitrary concerns, the representation of composite concerns, support the association of concerns to arbitrary software units, work products, or system elements, language independent, methodology compatible, applicability across the software development life cycle, support the representation of arbitrary relationships among concerns and widely supported by various software engineering phases.

Applications: Concern-space modeling has many potential applications in software development. It provides a form of documentation for basic information about concerns and their relationships. This kind of model can afford a global perspective that draws on, combines and relates concerns from multiple work products and life cycle stages.

This kind of concern modeling framework that contains physical concerns (representing work products) and mapping relationships (that relate logical concerns to physical concerns) can serve as a semantic hyperindex that allows concerns to be traced into work products and development tasks. This supports traceability of concerns into and across work products and stages and it makes it possible to see how concerns arise, are propagated and possibly dropped across stages and iterations of the life cycle.

Mapping relationships further allow us to assess the impact on the physical level of changes on the logical level. For example, if we no longer care about robustness and lose that motivation for a concern such as logging, then we may be able to safely drop the software units that implement that concern. However, we may also find that a unit considered for deletion also contributes to other purposes (as logging may also support auditing) and so should be retained.

Another application is in organizing code (or other units) for purposes of concern-driven program composition.

Mapping UML extensions through composition patterns to aspects: Requirements such as distribution or tracing have an impact on multiple classes in a system and are described, in general, as cross-cutting requirements, or aspects. Scattering and tangling make object-oriented software difficult to understand, extend and reuse. Though software design is an important activity within the software life cycle with well-documented benefits, those benefits are reduced when cross-cutting requirements are present. One approach to mitigate these problems is by separating the design of cross-cutting requirements into composition patterns.

Composition patterns require extensions to the UML and are based on a combination of the subject oriented model for composing separate, overlapping designs and

UML templates. We also show how composition patterns map to one programming model that provides a solution for separation of cross-cutting requirements in code-aspect-oriented programming. This mapping serves to illustrate that separation of aspects may be maintained throughout the software life cycle.

A composition pattern is a design model that specifies the design of a cross-cutting requirement independently from any design it may potentially cross-cut and how that design may be re-used wherever it may be required. Composition patterns are based on a combination of the subject-oriented model for decomposing and composing separate, potentially overlapping designs and UML templates.

Mapping to AspectJ: At the conceptual level, composition pattern (Rashid *et al*., 2003) design and aspect-oriented programming also have the same goals. Composition patterns provide a means for separating and designing reusable cross-cutting behaviour and aspectoriented programming provides a means for separating and programming reusable cross-cutting behaviour. The advantages of this are two-fold. First, from a design perspective, mapping the composition pattern constructs to constructs from a programming environment ensure that the clear separation of cross-cutting behaviour is maintained in the programming phase, making design changes easier to incorporate into code. Secondly, from the programming perspective, the existence of a design approach that supports separation of cross-cutting behaviour makes the design phase more relevant to this kind of programming, lending the standard benefits of software design to the approach.

Design issues for component based software development: Component Based Software Development (CBSD) and more recently, Aspect-Oriented Software Development (AOSD) have been proposed to tackle problems experienced during the software engineering process. When applying CBSD, a full-fledged software-system is developed by assembling a set of premanufactured components. Each component is a black-box entity, which can be deployed independently and is able to deliver specific services. The deployment of this paradigm drastically improves the speed of development and the quality of the produced software. AOSD on the other hand, tries to improve the separation of concerns in current software engineering methodologies, by providing an extra separation dimension along which the properties of a software-system can be described.

Currently available AOSD-research mainly focuses on Object Oriented Software Development (OOSD). CBSD however, also suffers from the problems that arise with the tyranny of the dominant decomposition. Similar to OOSD, aspects such as synchronization and logging are encountered, which crosscut several components from which the system is composed. Consequently, the ideas behind AOSD should also be integrated into CBSD. The other way around, namely the integration of CBSD within AOSD, is a valuable concept as well. CBSD puts a lot of stress on the plug-and-play characteristic of components; for example, it should be possible to extract a component from a particular composition and replace it with another one. Introducing a similar plug-and-play concept in AOSD, would make aspects reusable and their deployment easy and flexible.

Combining the AOSD and CBSD principles is a valuable contribution to both paradigms. However, currently available AOSD and CBSD research cannot be straightforwardly integrated, this because of several restrictions which are imposed by the existing approaches:

- The deployment of an aspect within a software-system is at this moment rather static. In AspectJ for example, an aspect looses its identity when it is integrated within the base-implementation of a software system. This makes it very difficult to extract an aspect from a particular composition and to replace it afterwards with a totally different aspect. This plug-and-play property is vital in some environments where the dynamic characteristic of components is considered an essential requirement.
- Most AOSD approaches describe their aspects with a specific context in mind. Therefore, it is impossible to reuse aspects. This is not acceptable within CBSD, since every component of a software-system should be independently deployable.
- The communication between the various components from which an application is composed, is in most cases specific to the employed component model. Java Beans for instance, makes use of an event-model. Currently available AOSD-technologies however, are not suited to deal with these specific kinds of interactions.

To integrate the ideas of AOSD into CBSD, we need a new aspect-oriented implementation language, designed especially for CBSD. This language should enable the development of software along another separation dimension, on top of the Java class hierarchy. It stays as close as possible to the regular Java syntax and introduces two concepts: aspect beans and connectors. An aspect bean is a regular Java bean that is able to

declare one or more logically related hooks, as a special kind of inner classes. Hooks are genetic and reusable entities and can be considered as a combination of the AspectJ's pointcut and advice. Since aspect beans are described independent from a specific context, they can be reused and applied upon a variety of components. The initialization of a hook with a specific context is done by making use of connectors.

To make such a language operational, we need a new component model that already incorporates the necessary traps to enable dynamic aspect application and removal. Another advantage of this new component model will be that component developers are still able to guarantee QOS for their components. However, the dynamicity and flexibility gained by using this new component model comes with a price in the form of large performance overhead compared to static languages, like for example AspectJ. As a consequence, this approach can be limited in use where limited resources is an issue.

AspectJ extensions for distributed computing: Current programming systems do not provide mechanisms for modularizing crosscutting concerns in distributed systems and thus they are major sources of low readability and maintainability of the software. Issues like transactions, security and fault tolerance are typical crosscutting concerns in distributed systems.

Many crosscutting concerns also arise during unit testing of distributed systems. The code for unit testing includes typical crosscutting concerns that AspectJ can deal with. AspectJ is a widely used language for Aspect-Oriented Programming (AOP) in Java. Unfortunately, if we use AspectJ to modularize testing code for distributed software, the code (aspect) can be somewhat modular but it often consists of several sub-components distributed on different hosts. They must be manually deployed on each host and the code of these sub-components must include explicit network processing among the sub-components for exchanging data since they cannot have shared variables or fields. These facts complicate the code of the aspect and degrade the benefits of using aspect oriented programming.

Implications on network processing: AspectJ is a useful programming language for developing distributed software. It enables modular implementation even if some crosscutting concerns are included in the implementation. However, the developers of distributed software must consider the deployment of the executable code. Even if some concerns can be implemented as a single component (aspect) at the code level, it might need to be deployed on different hosts and it would therefore consist of several sub components or sub-processes running on each host. Since Java (or AspectJ) does not provide variables or fields that can be shared among multiple hosts, the implementation of such a concern would include complicated network processing for exchanging data among the sub components.

Programming frameworks such as Java RMI do not solve this problem of complication. Although they make details of network processing implicit and transparent from the programmers' viewpoint, the programmers still must consider distribution and they are forced to implement the concern as a collection of several distributed sub-components exchanging data through remote method calls. The programmers cannot implement such a concern as a simple, non distributed monolithic component without concerns about network processing. This is never desirable with respect to aspect orientation since it means that the programmers must be concerned about distribution when implementing a different concern.

Contributions: Here we summarize the main contributions and achievements of the research carried out as part of this research.

The overall goal of this study, namely to design a aspect oriented design language that enables flexible extensibility of requirements and design functionality, has been successfully fulfilled in the form of AOSDDL structure. The validation of the architectural design with respect to its feasibility and practicality has been accomplished through prototype implementations of the AOSDDL architecture.

- Natural extension to UML
- CASE Tool Support
- Extension of Architectural framework for design constructs
- Enforcing Architectural Regularities
- Commercial Viability
- Implementation Support
- Software Development

FUTURE SCOPE OF WORK

Besides the ongoing development efforts to complete the AOSDDL prototype implementation (Clarke and Walker, 2002), further work in this area focuses on using and extending the AOSDDL notation architecture and prototype platform in order to build and experiment with design language specifications.

The code generators, tool integration and notation deployment and are few examples of ongoing research that take advantage of the AOSDDL architecture and platform.

CONCLUSION

This study has examined the design notation issue of software concerns for aspect oriented software development and design language.

Identifying concerns helps in early understanding of requirements and when requirements need to be mapped onto elements of a software solution, identifying aspects may become much more worthwhile generally concerns embody knowledge about a software system and its components and this information can support many software development tasks, such as rationale capture, impact analysis, change propagation and software composition and decomposition.

To integrate the ideas of AOSD into CBSD, we need a new aspect-oriented implementation language, designed especially for CBSD. To make such a language operational, we need a new component model that already incorporates the necessary traps to enable dynamic aspect application and removal. However, the dynamicity and flexibility gained by using this new component model comes with a price in the form of large performance overhead compared to static languages, like for example AspectJ. As a consequence, this approach can be limited in use where limited resources is an issue. Software design is an important activity in the development life cycle but its benefits are often not realized. Scattering and tangling of cross-cutting behaviour with other elements causes problems of comprehensibility, traceability, evolvability and reusability. Attempts have been made to address this problem in the programming domain but the problem has not been addressed effectively at earlier stages in the life cycle. Composition patterns presents an approach to addressing this problem at the design stage.

AspectJ extensions for identifying join points in the execution of a program running on a remote host can simplify the description of aspects with respect to network processing if the aspects implement a crosscutting concern spanning over multiple hosts.

The primary focus of this design language is to provide a highly flexible and extensible set of notations suitable for aspect oriented software development in all real world scenarios, suitable as a research platform for aspects that can form the basis for further research into aspect oriented systems and software engineering in general.

There are still many issues to be solved until efficient AO development support comparable to current OO support is established. There is still a lot of challenging research to be done in the future until the paradigm is widely accepted and developers are aware of the benefits AOSD offers.

REFERENCES

Baniassad, E.L.A., G. Murphy, C. Schwanninger and M. Kircher, 2002. Managing crosscutting concerns during software evolution tasks: An inquisitive study, ACM Proceedings on Aspect Oriented Software Development, pp: 120-126.

Banniassad, E. and S. Clarke, 2005. Aspect Oriented Analysis and Design: The Theme Approach. Addison-Wesley.

Clarke, S. and R.J. Walker, 2002. Towards a standard design language for AOSD. ACM Proceedings on Aspect oriented Software Development, pp: 113-119.

Clarke, S. and Robert J. Walker, 2001. Composition patterns: An approach to designing reusable aspects. IEEE proc. 23rd Int. Conf. Software Eng., pp: 5-14.

Ho, W., J. Jezequel, F. Pennaneac'h and N. Plouzeau, 2002. A toolkit for weaving aspect oriented UML Designs. ACM Proceedings on Aspect Oriented Software Development, pp: 99-105.

Rashid, A., A. Moreira and J. Araujo, 2003. Modularisation and composition of aspectual requirements. ACM Proceedings on Aspect Oriented Software Development, pp: 11-20.

Shomrat, M. and A. Yehudai, 2002. Obvious or not? Regulating architectural decisions using aspect-oriented programming. ACM Proceedings on Aspect Oriented Software Development, pp: 3-9.

Stein, D., S. Hanenberg and R. Unland, 2002. A UML-based aspect orientation design notation for Aspect. ACM Proceedings on Aspect Oriented Software Development, pp: 106-112.

Suvee, D., W. Vanderperren and V. Jonckers, 2003. JAsCo: An aspect-oriented approach tailored for component based software development. ACM Proceedings on Aspect Oriented Software Development, pp: 21-29.