MOVING FROM AOP TO AOSD DESIGN LANGUAGE

Deepak Dahiya **Rajinder K. Sachdeva**

Abstract

This paper recapitulates the work and summarizes the various stages of the research work carried out on development of Aspect Oriented Software Development Language (AOSDDL). It introduces the concept of "aspect oriented programming" and outlines the general path of research that has been taken. An analysis of the evolution of object oriented design methodology shows that the original object or class architecture was not designed for the requirements of today's enterprise wide distributed environment. This paper outlines how the novel paradigm proposed by aspect oriented design language could advance the current design architecture and overcome its main design flaws. A discussion of the applications of aspect oriented programming and its advantages highlights the potential beneficiaries of this new design methodology, namely third party tool developers, software developers, software vendors and most importantly the end users.At the end, the paper describes the main research challenges that are targeted by this research effort Further, a series of conclusion remarks summarizes what has been learnt from this work, and how these experiences contribute to the wider field of research.

1 Introduction

In the early days of computer science, developers wrote programs by means of direct machine-level coding[1]. Unfortunately, programmers spent more time thinking about a particular machine's instruction set than the problem at hand. Slowly, we migrated to higher-level languages that allowed some abstraction of the underlying machine. Then came structured languages, we could now decompose our problems in terms of the procedures necessary to perform our tasks. However, as complexity grew, we needed better techniques. Object-oriented programming (OOP) let us view a system as a set of collaborating objects. Classes allow us to hide implementation details beneath interfaces. Polymorphism provided a common behavior and interface for related concepts, and allowed more specialized components to change a particular behavior without needing access to the implementation of base concepts.

Programming methodologies and languages define the way we communicate with machines. Each new methodology presents new ways to decompose problems: machine code, machine-independent code, procedures, classes, and so on. Each new methodology allowed a more natural mapping of system requirements to programming constructs. Evolution of these programming methodologies let us create systems with ever increasing complexity. The converse of this fact may be equally true: we allowed the existence of ever more complex systems because these techniques permitted us to deal with that complexity.

There is a well documented problem in the software engineering field relating to a structural mismatch between the specification of requirements for software systems and the specification of object-oriented software systems. The structural mismatch happens because the units of interest during the requirements phase (for example, feature, service, capability, function etc.) are different to the units of interest during object-oriented design and implementation (for example, object, class, method, etc.)[2]. The structural mismatch results in support for a single requirement being scattered across the design units and a single design unit supporting multiple requirements - this in turn results in reduced comprehensibility, traceability and reuse of design models. Currently, OOP serves as the methodology of choice for most new software development projects. Indeed, OOP has shown its strength when it comes to modeling common behavior. However, OOP does not adequately address behaviors that span over many -- often unrelated -- modules. Separation of concerns is a basic engineering principle that is also at the core of object-oriented analysis and design methods in the context of UML [3]. Separation of concerns can provide many benefits: additive, rather than invasive, change; improved comprehension and reduction of complexity; adaptability, customizability, and reuse.

In contrast, AOP [4] methodology fills this void. AOP quite possibly represents the next big step in the evolution of programming methodologies. However, for aspect-oriented software development (AOSD) [5] to live up to being a software engineering paradigm, there must be support for the separation of crosscutting concerns across the development lifecycle including

traceability from one lifecycle phase to another. Concerns that have a crosscutting impact on software (such as distribution, persistence, etc.) present well-documented difficulties for software development. Since these difficulties are present throughout the development lifecycle, they must be addressed across its entirety.

Although a lot has been done to study the aspect oriented design approach in enterprise systems for architecture and its implementation, work on a general-purpose design language for aspect-oriented software development is attracting a lot of attention. The development of aspect oriented requirements gathering approach, design notation and environment for development of enterprise systems needs to be further refined in the context of software applications and industry.

This discussion has shown a range of design methodologies related to object oriented and aspect oriented software development that augment the current software industry scene and practices. Ongoing efforts in this area suggest that this trend of incorporating aspect elements inside any object oriented software design is far from over.

The majority of these designs are implemented as individual ad-hoc extensions – all with the goal of improving the software design to account for today's requirements such as logging, caching, persistence and distribution. However, the fundamental problem, namely that the programming methodology provides no architectural support for flexible extensibility, remains.

The research work therefore investigates traceability between developing a standard and general purpose AOSD design language with existing UML features and extensions to map AOSD design notations to AOP language. The aim is to provide a uniform design interface to add new extensions (for example, logging, caching, security etc) with a view towards eventually developing a standard design language for a broad range of AOSD approaches – independent of the programming language in hand.

2 Aspect Oriented Programming And Design

A gap exists between requirements and design on one hand, and between design and code on the other hand. Aspect oriented programming (AOP) extended to the modeling level where aspects could be explicitly specified during the design process will make it possible to weave these aspects into a final implementation model. Another step could be extension of AOP to the entire software development cycle. Each aspect of design and implementation should be declared during the design phase so that there is a clear traceability from requirements through source code thus using UML as the design language to provide an aspect-oriented design environment.

The separation and encapsulation of crosscutting concerns has been promoted as a means of addressing these difficulties; the standard object-oriented paradigm does not suffice. In order to overcome the difficulties for crosscutting concerns throughout the lifecycle, an approach is required that provides a means to separate and encapsulate both the design and the code of crosscutting behaviour. It is important to work towards a general purpose AOSD design language that meets certain goals including the following:

- *Implementation language independent*: The final form of AOP language may vary from that of any current one. Thus, any design language that simply mimics the constructs of a particular AOP language is liable to fail to achieve implementation language independence.
- *Design-level composability*: Design level composability is a desirable property for two reasons. First designers may check the result of composition prior to implementation, for validation purposes. Second, some projects will continue to require the use of a non-aspect-oriented implementation language because of pragmatic constraints, such as the presence of legacy code written in languages without aspect-oriented extensions; these projects could still benefit from separating the design of crosscutting concerns.
- *Compatibility with existing design approaches*: An AOSD design-level language should also build existing design languages such as UML, to provide a bridge from old techniques to new, so that software engineering realities such as incremental adoption and legacy support are possible.

The construction of complex, evolving software systems requires a high-level design model. This model should be made explicit, particularly the part of it that specifies the principles and guidelines that are to govern the structure of the system. In reality, however, implementators tend to overlook the documented design models and guidelines, causing the implemented system to diverge

from its model. Reasoning about a system whose models and implementation diverge is error prone – the knowledge we gain from these models is not of the system itself, but of some fictious system, the system we intended to build. The system's comprehensibility is impeded, and so using software engineering techniques goes against our intended goals – quality, maintainability and cost minimization. The essence of the problem of implementing higher-level principles and guidelines lies in their globality. These principles cannot be localized in a single module, they must be observed everywhere in the system, which means that they crosscut the system's architecture.

3 Why do we need Aspect Oriented Design in Software Development?

The identification of the mapping and influence of a requirement level aspect promotes traceability of broadly scoped requirements and constraints throughout system development, maintenance and evolution. The improved modularization and traceability obtained through early separation of crosscutting concerns can play a central role in building systems resilient to unanticipated changes hence meeting the adaptability needs of volatile domains such as banking, telecommunications and e-commerce. These crosscutting concerns are responsible for producing tangled representations that are difficult to understand and maintain. Examples of such concerns at the requirements level are compatibility, availability and security requirements that cannot be encapsulated by a use case and are typically spread across several of them.

With increasing support for aspects at the design and implementation level, the inclusion of aspects as fundamental modeling primitives at the requirements level and identification of their mappings also helps to ensure homogeneity in an aspect oriented software development project.

The main drive behind aspect oriented design language research is the idea of developing design constructs (elements) that exhibit a degree of flexibility and customizability that is only known from programmable end systems. While new design language constructs based on aspect oriented programming are being designed they are still tied to a particular platform whereby the vendor provides both the software tool and the design language tool as a complete package with additional proprietary tools. Thus, new design language aspect constructs can only be tested or utilized to individual specific requirements after the vendor has released a software upgrade. The development of new functionality is typically preceded by a long and awkward standardization process. These different paradigms have created an increasing gap between the functions and capabilities of these constructs in an aspect oriented development environment.

Reconsidering the system architecture of object oriented software applications is therefore a crucial step in aspect oriented software development.

4 Aspect Oriented Software Development Design Language

AspectJ [6, 7, 8] is a popular and well established AOP language that provides support for specifying and composing crosscutting code into a core system. It supports the AOP paradigm by providing a special unit, called "aspect", which encapsulates crosscutting code. Other compositional implementation languages and mechanisms also exist [9, 10]. At the design level, an AOSD design language with extensions to UML [1, 11, 12, and 13] in its capabilities relating to decomposition and modularization is required that would map to a particular AOSD implementation. Further, a standard AOSD design language must be capable of supporting many of these aspect programming languages. A graphical notation helps developers to design and comprehend aspect-oriented programs. Further, it would facilitate the perception of aspect-orientation. A design notation helps developers to assess the crosscutting effects of aspects on their base classes. Its application carries over the advantages of aspect-orientation to the design level and facilitates adaption and reuse of existing design constructs.

5 Research Challenges

The advantages of a flexible and extensible aspect oriented design language are expected to benefit the software community at various levels.

The main aim of this work is to investigate flexible and extensible mechanisms that enable dynamic introduction of new functionality into an existing operational design. This endeavor is pursued from the endpoint of the programmer and the design team as both has a great interest in implementation and / or processing of individual elements.

The key challenge of this research work therefore is to design a novel design language architecture that provides the basis for flexible extensibility of design functionality. In order to verify the practicality of this architecture, prototyping an application according to the new design elements will be a major part of this undertaking.

The challenges of the architectural design language are as follows:

- *Generic platform (not tied to a specific application)*
  The design goal is to develop a generic programmable design language platform to support the diversity of today's and future design specifications. The idea is to replace the numerous ad hoc approaches to provide specific design elements inside the language that allows users (such as programmers or systems analyst) to extend the design capabilities in a uniform way.
  Unlike most existing design language architectures, which are tied to a specific application domain, the goal here is to start with a requirement analysis of a wide range of software applications and design specifications in order to consider the multitude of requirements in the architectural design.

- *Modular component-based architecture*
  Another key objective is to design a design language architecture that is truly component-based taking advantage of component features such as modularity, extensibility, and reusability. The design elements can hence be programmed into aspects or classes called components. These components will typically provide a new specification or simply extend an existing specification.
  The component architecture allows complex technical and design specifications to be split into simply and easy-to-develop functional components. This 'divide and conquer' approach eases the design and development of specifications. Moreover, it improves the granularity of design specification extensibility and reusability of components among specifications.

- *Compatibility and transparency*
  The introduction of aspect oriented programming in current design methodologies, such as object-oriented, depends largely on how easily it can be integrated with existing technologies. It is therefore a major objective to design the design language architecture in a way that enables seamless transitioning towards the aspect based programming paradigm. Most early design proposals, for example, did not consider the crosscutting concerns, a vital requirement, and hence, ended up with solutions that rely on a design consisting only of objects and classes. Such software systems are obviously very hard to introduce in a distributed environment where security, caching and logging are major concerns. Consequently, an important goal here is to design an aspect based architecture that allows transparent, and hence seamless, application of design elements to the software components. No change to the domain specific functional components, systems and applications, or the intermediate modules that are not directly involved should be required. Such transparent solutions have the advantage that a partial transitioning from object oriented design to aspect oriented design – where the common but the more important concerns reside are most effective – is possible.

- *Commercial feasibility*
  Another important factor for the success of aspect oriented design language is its commercial viability. Many great technologies have failed in the past simply due to a weak business model. As a result, this work focuses on a solution that has evident beneficiaries and a likely commercial perspective.

  The challenge is to develop an active design language that enables third party development of aspect based software applications. Breaking the tight coupling between the design language and the software development environments decouples the role of the systems analyst from the software vendor and thus opens up a new competitive market for third party aspect oriented design software. This is particularly promising as unhindered competition typically maximizes the cost-performance ratio of products and specifications.

6 Work Outline Summary

This paper introduced the concepts of aspect oriented programming and software development. It outlines how the new methodology has emerged from traditional object oriented methodologies as a result of the growing demands of today's software practitioners and applications. Furthermore, it provides the motivation for this line of research along with the main research challenges of this study. The remainder of this work is outlined as follows:

Initial work dealt with a comprehensive overview of the current state-of-the art in the field by introducing related work that is or has been under investigation at other research institutions and universities. A special focus is placed on research into aspect oriented software design methodologies and enabling technologies. This work concludes with an overview of current work on aspect oriented applications and design language specifications.

Next, work continued with the requirements analysis for aspect oriented systems. The requirements are derived from past experiences in object oriented and aspect oriented programming paradigms of working in the software industry and academics and a thorough study of related work as well as other influencing factors, for example commercial aspects such as the deployment of new technologies. From these general requirements a subset of requirements that form the basis for the design of the AOSDDL design language architecture and implementation is drawn.

After analyzing the requirements, AOSDDL design language notations are defined. This central part of the research work describes in detail how AOSDDL operates and how the component based design architecture enables handling of crosscutting concerns through flexible integration and extensibility of design functionality. In addition to the basic language design, special focus is placed on the following key aspects: components, distribution and weaving.

Accordingly, as a next step was the ongoing implementation efforts of developing prototype design constructs of the AOSDDL design language architecture will be described. Due to the considerable extent of the AOSDDL architecture, this work initially focused primarily on validating the key aspects of the design through a 'proof-of-concept' implementation.

It continues with a qualitative and quantitative evaluation of AOSDDL and its prototype implementation. It evaluates how the AOSDDL architecture satisfies the objectives and requirements identified in the previous phase based on a case study and several example applications.

Finally, the research work concludes by drawing together the main arguments of this work and summarizing the contributions that have been made. It also describes future work that could be carried out based on this line of research.

7 Contributions

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

The overall goal of this work, 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

8 Future Scope of Work

Besides the ongoing development efforts to complete the AOSDDL prototype implementation[10], 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.

9 Conclusion

Several conclusions can be drawn from the development of AOSDDL:

Enforcing Architectural Regularities

The problems encountered were not as a result of an incorrect AOP design concept or idea in general but a consequence of its particular implementation. AspectJ being the only implementation available that is widely in use and is still undergoing changes. The language was not designed for the purpose of regulating architectural decisions and thus lacks sufficient tools to accommodate this task. The various design considerations regarding distributed architecture are possible with design constructs of AOP but it is their realization that caused difficulties.

AOSDDL Features

- An approach for high level architecture design, called AOSDDL, has been developed to enable separation of concerns at the design level of an AO development process. Within this approach it is assumed that the requirements have already been defined and specified during previous development stages.

- Since AOSDDL is UML conform, any CASE tool that supports UML modeling can be used.

- Aspects and base elements are completely kept apart; they are connected via a special language-specific connector element that encapsulates the underlying implementation technology. Any desired AO technology can be supported; it is

just the connector's syntax and semantics that have to be specified.

- Both, aspects and base elements, can be reused separately as the connector is the only crosscutting, language-dependent part. This sort of encapsulation offers a logical grouping of all classes belonging to one concern and eases the readability of design models as avoiding graphical tangling.

- To offer low-level architecture design support, a code generator needs to be developed to improve productivity and reduce errors when mapping model to code.

The work can be seen as a first step towards a simple and powerful modeling approach that fosters support from existing CASE tools since it is based on standard UML. AOSDDL in combination with the code generator should make AOSD more usable and more efficient for software development. The assumptions about the usefulness of the notation and the AO code generation have to be proven in the near future when using it in business development projects.

10 References

[1] Aspect Oriented Programming. http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html , 2003

[2] Object Management Group (OMG). Unified Modeling Language Specification. Version 2.0, Mar. 2003.

[3] Rambaugh, Jacobson Booch, UML Reference Manual. Addison-Wesley, 1998.

[4] Aspect-oriented programming: http://aosd.net

[5] Siobhan Clarke and Robert J. Walker. " Towards a Standard Design Language for AOSD," ACM Proceedings on Aspect Oriented Software Development, (April 2002), pp. 113-119.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M.Kersten, J. Palm and W. Griswold. " An overview of AspectJ," ECOOP Proceedings (2001), pp. 327-353.

[7] Palo Alto Research Center. http://www.parc.com/, 2003

[8] The AspectJ Team. The AspectJ programming Guide. http://www.eclipse.org/, 2006

[9] IBM Research. http://www.research.ibm.com/, 2003

[10] IBM alphaWorks. http://www.alphaworks.ibm.com/tech/hyperj, 2003.

[11] Wai-Ming Ho, Jean-Marc Jezequel, Francois Pennaneac'h and Noel Plouzeau. "A Toolkit for Weaving Aspect Oriented UML Designs, " ACM Proceedings on Aspect Oriented Software Development, (April 2002), pp. 99-105.

[12] Awais Rashid, Ana Moreira and Joao Araujo. "Modularisation and Composition of Aspectual Requirements," ACM Proceedings on Aspect Oriented Software Development, (2003), pp. 11- 20.

[13] Mika Katara, Shmuel Katz. " Architectural Views of Aspects, " ACM Proceedings on Aspect Oriented Software Development, (2003), pp. 1- 10.