Anale. Seria Informatică. Vol. II fasc. I
Annals. Computer Science Series. 2nd Tome 1st Fasc.

Aspect Oriented Programming Challenges

Dr. Ioan Despi, Lecturer,
University of New England, Armidale, Australia
Dr. Lucian Luca, Associate Professor,
"Tibiscus" University, Timisoara, Romania

ABSTRACT. Separation of concerns is an important software engineering principle, meaning the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concern (concept, goal, purpose). A new emerging paradigm is introduces and discussed – Aspect Oriented Programming, that makes possible to express those programs that OOP fails to support.

1 Introduction

Ten years ago a revolutionary movement started in the object-oriented (OO) developer community that tried to bring a new level of flexibility to today's complex object-oriented programming (OOP) paradigms. It's called A*spect-Oriented Programming* (AOP), a simple idea that promises radical results for systems that use similar objects to interact with multiple classes - called *cross-cutting concerns*. Examples of these concerns would be logging, synchronization, and error handling - for instance. OOP languages don't handle these cross-cutting concerns - or *aspects* - very well from a design standpoint, which results in tangled code. AOP was developed to separate those concerns from the rest of the program to allow for more reusability and to avoid duplicating code. Informally, if the main program is designed from a top/down point of view, aspects would be introduced from the left/right point of view and used by the program when needed.

Aspect-Oriented Software Development (AOSD) appears as a new technology that focuses on the separation of cross-cutting concerns. Although the AOSD has been working towards a solution since 1995, it still hasn't become generalized enough for most programmers to understand and use confidently. According to many voices in the AOP community, which – by the way – numbers only about 2,000 registered people worldwide, only about 10-15% (300) of programmers are experienced enough to use AOP in an OOP environment.

AOP doesn't require a new syntax or way of programming and it can be introduced into OOP applications incrementally. There are three approaches AOP takes when used by programmers - language-based, at the metadata level, and code instrumentation at program runtime. All three are used to complete what researchers call the *aspect weaver*. What's missing today is standardization of the approaches used in AOP, maybe AOP has to go to some packaging into something a novice or mid-level programmer can type and understand.

Using AOP doesn't deliver speed enhancements at compilation time or a drastic reduction in memory usage when its run. What it delivers, however, is a level of flexibility you can't find in today's larger programs. It lets programmers develop components (or aspects) of an object so changes can be made in one place, instead of throughout the hierarchies. While the technology is being used at the university Ph.D. thesis level and within the research laboratories like Microsoft and IBM, AOP doesn't get a lot of attention from the people who could provide the most growth opportunities for the platform: IT managers.

2 Concerns

A *concern* is any issue in the domain of a problem, a property of an area of interest of the system. It can be a primitive one (e.g., buffering, caching) or a complex one (e.g., dependability, safety). More, a concern can be functional (e.g., business rule) or non-functional (e.g., transaction management).

The ability to identify, encapsulate, and manipulate only those parts of the software that are relevant to a particular concept, goal or purpose is called *Separation of Concerns* (SoC) and it is a main principle in software engineering [OT01].

SoC is closely related to composition and decomposition in programming languages, introduced in [Dij76] when talking about design process. In Dijkstra's opinion, the design process is about breaking down the system into units of behaviour or function, often known as functional decomposition. By splitting a large subject into smaller units, the complexity of the system is reduced, then by composition – the assembly of these smaller parts into a system – the whole functionality is (re)achieved. The definition says nothing about *how* to decompose a system and there are a lot of attempts and approaches to do it. The only clue is that reasonable criteria should be used in decomposing systems and the *separation of concerns principle* [Par72] states that every part of a decomposed system should be responsible for a well-defined task or concern of the system. It is desirable to reason about every concern in isolation so they should have as little knowledge about the other concerns.

In many situations separation of concerns is not easy to achieve. Once software systems reach a certain complexity, the modularisation constructs provided by current programming languages and environments fall short. Current software engineering techniques generally provide a dominant decomposition mechanism that is not suitable to capture and represent all kinds of concerns that can be found in software applications. This problem is identified in [TOH99] as the *tyranny of dominant decomposition*: modern languages and methodologies permit the separation and encapsulation of only one kind of concern at a time. Examples of tyranny decomposition are functions in functional paradigm, rules in rule-based systems and classes in OO paradigm. Hence the conclusion it is impossible to encapsulate and manipulate all the diverse and heterogeneous concerns in only one of the decomposition mechanisms.

3 OO Tyranny

OOP allows the designer to express the essence of the design in small amounts of syntactic material. In this way, OOP removes a difficulty from the design but – unfortunately -- it cannot do more than remove accidental difficulties from the design and the complexity of the design itself is still present [Bro87].

The fundamental aspect of OOP is *modularity* through encapsulation. Modularity decomposes complex problems into manageable modules. In well decomposed systems the modules show *low coupling*, i.e. interdependencies between modules are minimised, and *high cohesion*, i.e. a module's responsibilities are highly related. Modularity is a specialisation of the separation of concerns principle in that it separates software into components according to functionality and responsibility.

OOP modularise concerns by building hierarchies of classes through classification and specialisation. The problem here is the quickly increasing complexity of the system, due to the many objects and inter-relations between these objects. This leads to *high coupling* (strong dependencies between objects) and l*ow cohesion* (weak bindings inside objects), the opposite of what you get with good modularity (that is, objects with strong inner relations and few coupling to other objects) [EKS92].

OOP was useful regarding the separation of functional concerns of a system in class hierarchies. But across these hierarchies, concerns may exist that are not possible to generalise through inheritance or polymorphism. It seems that OOP approach of modularising software systems according to a single concern does not provide enough structure for developing complex systems. If we decompose a system into a class hierarchy, some general issues cannot be dealt with in a modularised way. Concerns not represented in the current system decomposition have to be forced onto the primary decomposition. These concerns are called *cross-cutting concerns*. Concerns that cross-cut these functional decompositions do not fit equally well into the OO model and have potentially harmful impact on software engineering quality factors (e.g., adaptability, maintainability, extensibility, and reusability).

As an example, let us consider a credit card processing system: its core concern would process payments, while its system-level concerns would handle logging, transaction integrity, authentication, security, performance, aso. Hence the credit card processing system consists of several concerns (at least system-level ones) that cross-cut multiple modules. OOP techniques for implementing such concerns result in systems that are invasive to implement, tough to understand, and difficult to evolve.

The presence of these hidden concerns is revealed in two ways:

1. the concern's code is *scattered* throughout the system, that is we have the actual functionality scattered across multiple classes as redundant code.
2. the concern's code is *tangled* with other code, that is two or more collocated concerns overlap each other.

As a consequence, hidden concerns can lead to:

i. inconsistency when modifying the code
ii. poorer maintainability
iii. less readable code
iv. inflexible code, and
v. violations of code standards and development procedures.

In the mean time, a proper separation of concerns has a number of benefits, divided in the following categories [Ost03] :

i. ***Comprehensibility.*** Putting together pieces of code otherwise non-contiguous in the program increases our understanding about them without having to know the structure of the whole system.

ii. ***Reusability***[1]. The more a piece of code concentrates on a single concern, the more likely to reuse it in different contexts.

iii. ***Scalability***[2]. With a good separation of concerns we should be able to escape Brooks' law[3], because the required communication and coordination is minimised. More, from a compiler perspective, program parts with little dependencies on other program parts make it easier to perform modular checking – that is, check and compile a single part in isolation.

iv. ***Maintainability***[4]. It is easier to maintain if we can localise a concern in a single place.

The separation of concerns is a recognised problem in software engineering and a lot of work was done to solve it. At the beginning, separation of concerns was more oriented towards the implementation, dealing with concerns that are tangled in the code. Then, Software engineering community recognised the need of separation of concerns through the whole software development cycle, starting with requirements elicitation and specification, and going through design, implementation, testing and so on. The new technology that is used is called *Aspect-Oriented Software Development* (AOSD) and it allows to separately specifying the concerns of a system and some descriptions of their relationships and then it provides mechanisms to *weave* or compose them together into a coherent program [EFB01].

4 Approaches

The first question to ask here is what infrastructure is needed to support AOSD? From the programmer point of view, we must supply aspect-oriented (sub) languages that are based on the constructs and syntax that the programmer is most familiar with, as well as features to manipulate the cross-cutting characteristics of concerns. At the implementation level, the aspect specifications must weave with the code and at runtime the support for aspect specification and integration should not degrade performance.

A number of post-object technologies have been proposed, including

1 Reusability means that one piece of software is used in multiple places.
2 Scalability refers to a reasonable relation between cost and size of a software system.
3 Brooks' Law [Bro75] states that adding more people to a software project makes it later, that is -in software engineering- Time <>Size/People
4 Maintenance means to add, remove or change a particular concern.

generic programming [Aus99], *generative programming* [CE00], *computational reflection* [Mae87], and the broad category of *Advanced Separation of Concerns* (AsoC). AsoC provides a software developer with a method of software decomposition more powerful than OOP provides alone. Different AsoC techniques define different programmatic abilities to reflect upon and modify the behaviour of a system. We briefly discuss some of them in the following subsections.

4.1 Composition Filters

Composition filters is may be the oldest composition method [Ber94]. The Composition Filters (CF) model is an evolution of the object model by means of the Sina language. The main idea is to wrap every object with Filters, i.e., entry code, which can catch and manipulate messages. Filter can swallow a message, they can delegate (to implement inheritance, or extend), or can synchronize with other objects (Synchronisation protocols), or can log, or can modify (override behaviour, adapt). To describe a range of data abstractions, the authors used the term of *interface predicate,* then instead of extending the language with numerous new language constructs, the framework of composition filter (CF) was introduced which integrates all these desired constructs and interface predicates into a single, unified model [BA01]. A composition-filter object consists of two parts: an interface and an implementation part. The interface is in charge with messages passed between objects. It consists of one or more input and output filters, optional internal and external objects and method header declarations. Filters are controlled by conditions. Filter names, method headers and conditions names can be made visible to the clients, but the implementation part is hidden. Each incoming message must pass a bank of input filters, each of which could cause the message to be blocked, diverted to another object, or modified in some way. Any message sent from the object must pass through a bank of output filters, with similar functionality. Filters work in a similar way as meta object protocol change (MOP change). Filters modify the MOP method *Class.acceptMessage* and filters can implement aspects (e.g., debugging) and views.

4.2 Meta-Object Programming

Some of the original inspiration for AOP comes from research in dynamic, reflective object-oriented languages and meta-object protocols.

Computational reflection [Smi84] enables a program to access its internal structure and behaviour and to programmatically manipulate that structure, thereby modifying its behaviour. The process of getting access to reflective data is called *reification*.

Meta-Object Protocol (MOP) [KRB91] provide the ability for a program to reason about itself. MOPs offer a refined form of reflection that focuses on modifying and reacting to object behaviour at runtime or structurally reflecting upon code at compile time. At the origin, the Meta-Object Protocol is a protocol layer in Common Lisp which contains a set of default rules about how methods are added, how classes inherit from super-classes, aso, that is, how the CLOS object system works. These protocols are built into the object system, and are enforced automatically, making the application development process much more efficient, as the protocols do not have to be manually invoked by the programmer wherever they are needed throughout the application. The typical features of a MOP include programmatic control of dynamic dispatch and subclassing of metaobjects such as classes and methods. More, the default rules of the MOP can also be modified, enabling the programmer to actually customize the object system to suit the application.

Using the MOP, the developer can model any object system he wants, even the object system of another language such as Java. MOPs for OO systems provide meta-objects attached to other objects or control or data flow structures and provide the ability to intercept base operations in the code and jump to the relevant meta-level meta-object. Java provides a kind of reflection capability. A Java program can ask for the class of a given object, find the methods of that class, and then invoke one of those methods. The ***Class and Method*** *classes in Java* are considered meta-classes and their instances are considered metaobjects. A metaobject protocol defines execution of an application in terms of behaviour implemented by metaclasses. Unfortunately, Java's reflection capabilities are not complete; it's a kind of read-only property: a program can query the methods of a class, but it cannot change them, whereas a full MOP allows modification of any meta-information that can be reified. More, Java does not allow subclassing of metaclasses *Class* and *Method*. Using the terminology of [KRB91], one can say that Java provides introspection but not intercession. Other OO languages, such as C++, provide even less in the way of computational reflection.

4.3 Adaptive Programming

Adaptive Programming (AP) [Lie96] is a novel programming style that has been invented and developed by the Demeter Research Group at The

Northeastern University in Boston. AP is a methodology rather than a concrete technology. It provides a high-level interface to conventional OOP, aiming at the production of better evolvable and maintainable code. AP decouple the class graph and behaviour of OO software along two dimensions, by using the Law of Demeter. It decomposes behaviour from underlying class graph, so modifications can be made to either dimension, without having negative side-effects on the other. Unfortunately, the two dimensions are not completely separated, they overlap and so the two dimensions of AP are not orthogonal. There must be some dependency between them because algorithms are always bound to data structures. Generally speaking, one can say that AP implements ideas of multi-dimensional separation of concerns [TOHS99].

The Law of Demeter [LO99] is a generally accepted design-style rule. It was found by Holland during research on the Demeter project in 1987. The law states:

Each unit should have only limited knowledge about other units: only units 'closely' related to the current unit.

In other words, the law promotes the principle of least knowledge, that is units shall not make assumptions about the whole environment but only about their immediate neighbourhood. A class should not rely upon the structure of other classes in the system except for an immediate set of neighbours. The main idea is to avoid chains of invocations, such as $C.O_1.O_2.O_3$ in the program code, because (in this case) the class C would make unnecessary assumptions about the class graph. For instance, C assumes that the class returned by the operation O_1 declares and operation O_2. The code may break as soon as O_1 returns another class. Instead, the Law of Demeter demands O_1 to propagate the message to O_2, which will propagate it further to O_3 [Lie96]. The law improves adaptiveness but the side effect is the time and space overhead required by the wrapper methods that propagate the message. The overall structure of the class hierarchy is discovered by the runtime system in the form of a UML-like class graph, and programmatic traversal of this structure is permitted using the *adaptive visitor* design pattern and a traversal strategy [LP97]. The traversal strategies specify the collaborating classes of operations, they define traversals on class graph, without enumerating all the classes, but by setting some minimal constraints that a concrete class graph must meet so that the traversal graph can be built. The actual behaviour is realised by adaptive visitors, which encapsulate the functionality of entire operations. The combination of a traversal strategy and an adaptive visitor is also known as a *propagation pattern* and all the above principles of AP operate under the term *structure-shy programming.* In general, an AP-enabled software is much adaptive (able to adapt to a new class graph) if has been developed with these principles in mind.

4.4 Subject-Oriented Programming

Subject-oriented Programming (SOP) is a program composition technology from IBM initiated by H. Ossher and W. Harrison [HO93]. The work is based on the observation that many objects in a software system play different roles during their lifetimes. In tool and application integration, it is common for different tools and applications to associate different states and behaviours with a same object. Their example is a *Tree* object, playing roles as a home to a *Bird* object, a *Logger* object, an *Accountant* object, aso.

A *subject* is a collection of state and behaviour specifications pertinent to a particular application or tool. A subject is definitional, it does not contain itself any state, it is a collection of class fragments whose class graph models its domain in its own subjective way. A subject may be a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a complete application. *Subject composition* combines class hierarchies to produce new subjects that incorporate functionality from existing subjects.

Subject-oriented programming involves dividing a system into subjects and writing rules to compose them correctly. Different subjects can separately define and operate upon shared objects, without any subject needing to know the details associated with those objects by other subjects. *Subject activation* provides an executing instance of a subject, including the actual data manipulated by a particular subject. *Composition rule* specifies in detail how the components are to be combined. An object identifier (*oid*) is the globally known unique identification of the object as it appears in the context of one or more subjects of interest. Object means the state and behaviour associated with an object identifier by that subject and there is no global concept of class.

SOP is a language-independent technology. IBM have built support for subject-oriented programming in C++ as an extension of the IBM VisualAge for C++ Version 4 (VAC++) compiler and environment, in Java as Hyper/J, that supports multi-dimensional separation of concerns, and in ENVY/IBM Smalltalk.

4.5 Intentional Programming

Intentional Programming (IP) provides a natural, modular way for a programmer to manufacture a software system. IP enables programs to be written and viewed in a variety of specific notations. IP allows the programmer to specify a language-neutral intent through an easy to manipulate interface, and then IP can generate code in any target language. The technology was coined by Charles Simonyi's team while he was working at Microsoft Research [Sim95]. IP belongs more to Model-Driven Architectures (MDAs) and code generation tools than to AOP. AOP looks at programs and seeks cross-cutting aspects which localise the expression of that aspect (with all the benefits that such localisation affords), whereas IP looks at the specifications (the stakeholders concerns) and tries to faithfully represent those concerns in an XML tree-like form.

4.6 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) addresses the problem of the dominant decomposition by implementing the *base* program (addressing the dominant concern) and several *aspect* programs (each addressing a different cross-cutting concern) separately and then *weaving* them all together into a single executable program. Each aspect deals with a particular concern and is implemented by a special-purpose type.

4.6.1 AOP Concepts

The two major features that define an aspect-oriented system are *quantification* and *obliviousness* [FF00]. Quantification refers to the fact that a piece of code may affect another, completely separated piece of code somewhere else in the system. One can distinguish between static (source code) and dynamic (runtime) quantification. More, static quantifications are divided in black box (made only over the public interface) and clear box (made over the parsed structure of the underlying code). Dynamic quantifications are made over run-time events, like calling a subprogram or raising an exception. Obliviousness refers to the fact that the affected piece of code has not been specially prepared to receive this modification.

Recall [KLM97] *concerns* are properties or areas of interest in the system and they can be implemented as *components* (if can be cleanly encapsulated in a generalised procedure) or as *aspects* (if cannot be cleanly encapsulated in a generalised procedure). Concerns crosscut if the methods related to those concerns intersect, both inside a class or over several classes. AOP provides a way to encapsulate concerns. A location which is affected by one or more crosscutting concerns is called *join point*. A programmer can add here new actions, before or after the original code execution, on the static or dynamic structure of the program. An aspect is a modular unit of crosscutting concerns. Each aspect can be expressed in a separate and natural form, and can be automatically combined together into a final executable form by an *aspect weaver*. As a result, a single aspect can contribute to the implementation of a number of procedures, modules, or objects, increasing reusability of the code. Now is obvious that every AOP language should have three critical elements for separating crosscutting concerns: (i) a join point model, (ii) a means to identify join points, and (iii) an implementation tool for the join points [EAK01].

4.6.2 AOP Programming Languages

Heron is a new open source, general purpose, strongly typed, imperative programming language with built-in support for object oriented (OOP), interface oriented (IOP) aspect oriented (AOP) and generic programming techniques (http://www.heron-language.com). The Heron syntax is a mix of Pascal, Java and C++. Heron is designed primarily as a compiled language but the Heron language specification includes an easily interpreted subset called HeronScript.

AspectJ is a simple general-purpose extension to Java that provides for implementation of crosscutting concerns (http://www.aspectj.com). AspectJ identifies join points in the execution flow of a Java program, as nodes in a simple runtime object call graph. These nodes are the points at which an object receives a method call and points at which an attribute of an object is referenced. The edges are control flow relations between the nodes. *Pointcuts* are a means to make reference to a set of join points and to manipulate certain values captured in those join points. *Advices* are modules that encapsulate the crosscutting implementations upon pointcuts. *Aspects* are units of modular crosscutting implementation.

JBoss (http://www.jboss.org) is one of the most popular Java application servers in the industry and is the de facto Java application server development standard. It was developed as open source software, and passes over four million downloads. JBoss includes full support for J2EE-based APIs, that is Java objects can leverage features such as transactions and security that are usually reserved for J2EE objects. These features are provided as a collection of predefined aspects, and can be applied to application objects via dynamic weaving, without requiring the application itself to be recompiled. In other words, the JBoss AOP framework allows developers to write plain Java objects and add or remove some aspects (locking, encryption, and logging) later at runtime. JBoss is available at.

AspectWerkz (http://aspectwerkz.codehaus.org/) is a dynamic, fast, and free AOP framework for Java that offers both offline and online aspect weaving mechanisms. The offline mode allows aspects to be integrated ("woven") into application code before the application is deployed. The online mode weaves the aspects at class load-time in a transparent way. Both implement dynamic AOP concepts that allow developers and system operators to add, remove, and modify aspects during the execution of an application. One can choose between an XML-based notation or on a Java framework.

5 Conclusions

Aspect-oriented programming claim to contribute to solve the problems mentioned for OOP so one can ask if it will replace the methodologies we know today. AOP is a very important paradigm that gives the programmer the opportunity to move distracting support functionality away from the primary code. There is no risk for inconsistency since a modification of the aspect code will be applied everywhere the aspect is used. Maintainability is improved as changes only have to be performed in one place. In this way, AOP will probably add a new standard to programming, but it will not replace anything we use today, in the same way as OOP did not replaced procedural and functional programming. They are a part of OOP and so will OOP be in AOP.

AOP complements OOP by facilitating another type of modularity that puts together the implementation of a crosscutting concern into a single unit. The main benefits of AOP come from its ability to modularise implementations of crosscutting concerns. AOP strives to overcome the problems caused by code tangling and code scattering, improving thus productivity, reusability and the evolution of code. Because every concern is addressed separately with minimal coupling, the result is a system with less duplicate code, a modularised implementation even in the presence of other crosscutting concerns. This output is much easier to understand and maintain.

The main problem with AOP is that it adds a new dimension of possibilities and another one of complexity, hence the danger to keep it as simple as possible such that ordinary programmers can use it. AOP complexity comes from the new mechanisms and tools used in the implementations. Because AOP is actually a new paradigm, its application comes with the same sort of problems as when going from procedural programming to OOP. For instance, AOP is not very well tested and documented, and there is a lack of specific development tools.

AOP introduces a new paradigm, a new way of handling crosscutting concerns, a problem that is hard to solve in OOP. The future works for AOP.

References

[Aus99] **Austern, M.H.** – *Generic Programming and the STL.* Addison-Wesley, 1999

[BA01] **Bergmans, L.; Aksit, M.** - *Composing multiple concerns using composition filters.* Comm. Of the ACM (CACM) 44(10):51:57, October 2001

[Ber94] **Bergmans, L.** - *The Composition Filters Object Model.* Dept of Computer Science, University of Twente, The Netherlands, 1994

[Bro75] **Brooks, F.P.** - *The Mythical Man-Month.* Addison Wesley 1975

[Bro87] **Brooks, F.P.** – *No Silver Bullet: Essence and Accidents of Software Engineering.* Computer 20(4):10:19, April 1987

[CE00] **Czarnecki, K.; Eisenecker, U.** - *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley 2000

[Dij76] **Dijkstra, E.W.** – *A Discipline of Programming.* Prentice Hall, 1976

[EAK01] **Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.**-*Discussing Aspects of AOP.* CACM 44(10):33-38, Oct 2001

[EFB01] **Elrad, T.; Filman, R.E.; Bader, A.** - *Aspect-oriented programming: Introduction.* CACM 44(10):29-32, 2001

[EKS92] **Eder, J.; Kappel, G.; Schrefl, M.** – *Coupling and cohesion in object-oriented systems.* In Conf. on Information and Knowledge Management, Baltimore, USA, 1992

[FF00] **Filman, R.E.; Friedman, D.P.** – *Aspect-oriented program-ming is quantification and obliviousness.* In Proc. Of the ACM OOPSLA 2000 Workshop on Advanced Separation of Concerns. Minneapolis, Oct. 2000

[HO93] **Harrison. W.; Ossher, H.** – *Subject-Oriented Programming: a critique of pure objects.* In Proc. Of the 8th annual conf. OOPSLA, Washington DC, pp. 411-428, 1993

[KLM97] **Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopez, C.; Loingtier, J. M.; Irwin, J.** – *Aspect Oriented Programming.* In Aksit M, Matsuoka S (eds) – Proceedings European Conference on Object Oriented Programming, vol. 1241, pp 220-242, Springer Verlag, 1997

[KRB91] **Kiczales, G.; des Rivieres, J.; Bobrow, D. G.** – *The Art of Metaobject Protocol.* MIT Press, Cambridge, MA, 1991

[Lie96] **Lieberherr, K.** – *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, MA, 1996

[LO99] **Lieberherr, K.; Holland, I.** – *Assuring Good Style for Object Oriented Programs.* IEEE Software, pp 38-48, 1989

[LP97] **Lieberherr, K., Patt-Shamir, B.** - *Traversals of Object-Oriented Structures. Specification and Efficient Implementation.* Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, 1997

[Mae87] **Maes, P.** – *Concepts and experiments in computational reflection.* In Conf. Proc. On OOPSLA, Orlanda, FL, pp. 147-155, ACM Press, 1987

[Ost03] **Ostermannn, K.** – *Modules for Hierarchical and Crosscutting Models.* PhD Thesis, Technischen Universitat Darmstadt, Germany, April 2003

[OT01] **Ossher, H.; Tarr, P.** – *Using multidimensional separation of concerns to (re)shape evolving software.* CACM 44(10):43-50, 2001

[Par72] **Parnas, D.L.** - *On the criteria to be used in decomposing systems into modules.* CACM 15(5):330-336, 1972

[Sim95] **Simonyi, C.** - *The death of computer languages, the birth of Intensional Programming.* Technical Report MSR-TR-95-52, Microsoft Research, 1995

[Smi84] **Smith, B.** – *Reflection and semantics in Lisp.* In Conference Record of POPL84: The 11th ACM SIGPLAN-SIGACT

Symposium on Principles of Programming Languages, pp 23-35, 1984

[Sul01] **Sullivan, G.T.** – *Aspect Oriented Programming Using Reflection.* CACM 44(10):95-97, Oct. 2001

[TOHS99] **Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S.M.**–*N Degrees of separation: Multi-dimensional separation of concerns.* Proc. Of the 1999 Intl. Conf. On Software Engineering, pp. 107-119, IEEE Computer Society Press, ACM Press, 1999