

Obvious or Not? Regulating Architectural Decisions Using Aspect-Oriented Programming *

Mati Shomrat and Amiram Yehudai[†]
Computer Science Department, Tel-Aviv University
Tel-Aviv 69978, ISRAEL
PH: +972-3-6409299, FAX: +972-3-6409357
{matis,amiramy}@post.tau.ac.il

ABSTRACT

The construction of complex, evolving software systems requires a high-level design model. However, this model tends not to be enforced on the system, leaving room for the implementors to diverge from it, thus differentiating the designed system from the actual implemented one. The essence of the problem of enforcing such models lies in their globality. The principles and guidelines conveyed by these models cannot be localized in a single module, they must be observed everywhere in the system. A mechanism for enforcement needs to have a global view of the system and to report breaches in the model at the time they occur.

Aspect-Oriented Programming has been proposed as a new software engineering approach. Unlike contemporary software engineering methods, which are module centered, Aspect Oriented Programming provides mechanisms for the definition of cross-module interactions. We explore the possibility of using Aspect-Oriented Programming in general and the AspectJ programming language in particular for the enforcement of design models.

1. INTRODUCTION

In programming, much like in a democratic regime, one can do whatever she wants, unless specifically forbidden. The construction of complex, evolving software systems requires a high-level design model. This model should be made explicit[5], particularly the part of it that specifies the principles and guidelines that are to govern the structure of the system. Recent years have seen a lot of work done in the field of modelling: unification of modelling principles to form UML[2], and extensive work on design patterns[4], to mention just a few. In reality, however, implementors tend

*This paper is based on the first author's MSc Thesis, supervised by the second author. Supported in part by the Deutsch Institute.

[†]Currently at The Academic College of Tel-Aviv-Yaffo.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD 2002, Enschede, The Netherlands

Copyright 2002 ACM 1-58113-469-X/02/0004 ...\$5.00.

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 fictitious 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.

Two major approaches have been suggested to bridge the gap between high-level design models and the system itself: user invoked and environment invoked. The first was described by Sefika, Sane and Campbell[13]:

"the use of codified design principles must be supplemented by checks to ensure that the actual implementation adheres to its design constraints and guidelines."

The second approach was described by N.H. Minsky[11]:

"the gap between the architectural model and the implemented system can be bridged effectively if the model is not just stated, but is enforced."

As noted in[9] 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.

From the discussion above it is clear that a tool which is capable of enforcing design decision should have at least the following two characteristics: a) An overall view of the system. b) Any violation of a rule should be detected as early as possible and prevented.

Aspect Oriented Programming[6] (AOP) is a programming technique for modularizing concerns that crosscut the basic functionality of systems. Aspects provide a means to clearly capture design decisions[1].

In [6] G. Kiczales provides a definition to differentiate aspects from components:

"aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in a systematic way"

We claim that by viewing design decisions as aspects of the system, and using an AOP language as our verification¹ mechanism, we are actually enforcing design rules by means of a compiler. Once a regularity has been formed as an aspect it becomes part of the system, and will be weaved into it. No other human interaction is needed apart from specifying a regularity as an aspect. Therefore, if AOP proves to be able to bridge the gap, what we get is a tool that falls within the realms of enforcing architectural principles, and not just of verifying their existence.

On first glance, it seems that AOP is perfect as a design enforcement mechanism, but is it really that obvious? The literature in the field of "regulating architectural decisions" ([16, 14, 15]) discusses various design regularities that have been shown to be of importance, and this work examines AOP's applicability for some of them. The regularities that we have chosen to address are universal in the sense that they represent the kinds of architectural constraints that are most frequently occurring in systems. We rely strongly on N.H. Minsky's work on the enforcement of regularities[10, 9, 17, 12]: Apart from being the only other work on **enforcement** known to the authors, it is the most extensive work in this field, ranging from constraints on the development process itself, to those that can only be checked at run-time. This is in contrast to most other works which focus on the verification of statically checkable constraints.

The rest of the paper goes as follows: Section 2 describes our attempts to implement various types of design regularities, section 3 discusses our findings. The remaining sections present future work and conclusions. We assume the reader has knowledge of AOP and the AspectJ programming language.

2. ENFORCING ARCHITECTURAL REGULARITIES

The AOP literature is abundant with extensively documented examples of what aspects can be used for.

- **Design by Contract:** AOP can be used as a mean for the implementation of the *Design by Contract*[8] design methodology, as shown in the AspectJ tutorial. For example, pre / post conditions are checked using *before* and *after* (respectively) on a method execution join point.
- **Exception Handling:** The design regularity of "all exceptions of a certain type should be handled the same way" is discussed in [7].
- **Observer Design Pattern:** The enforcement of the *Observer* design pattern [4], is illustrated in the examples supplied by the AspectJ team. The example of this behavioral pattern uses the *introduction* mechanism as well as method call receptions.

What all of these examples have in common is the fact that the regularities they define are of a dynamic nature and are enforced upon a monolithic system.

It is clear how these examples can be used for enforcing architectural principles, but when attempting to implement

¹In this paper we take the meaning of the term *verification* to be: verifying that architectural decisions have been implemented correctly.

design restrictions with AspectJ, one quickly reaches realms which are not covered by the literature. Following are examples which illustrate these realms.

2.1 Distributed Architecture

Modern systems tend to be constructed of heterogeneous subsystems (agents) interacting with one another. Their architecture is a distributed one rather than centralized architecture of legacy systems. In order to accomplish its goal, an agent interacts with other agents under a coordination policy. These policies must be maintained by all agents interacting under it, hence they represent a crosscut in the distributed system's architecture.

Since the current version of the AspectJ² programming language requires all source code to be available to the language's compiler, we had to constrain the general model. The general model is comprised of "black box" agents, which are designed and implemented by various people, in different programming languages and operating on different platforms. The constraints we put on the model require all source code to be written in a single programming language, AspectJ, and to be available at compilation time.

As an example of such distributed policy, we present a congestion control policy enforced in a server/client environment, in which the server is a central data base, responding to queries by clients. The policy is designed to control possible congestion of the server due to large volume of requests. The system is described in [12] as follows:

Let S be a server ...

1. Every client of S has a quantum of time dt assigned to it, which is to be the minimal delay between any two requests sent by this agent to the server.
2. The server can set the delay of an agent to any desired value.
3. If an agent attempts to send a message to S sooner than permitted by his delay, this message is to be blocked.

We bind an enforcement mechanism to each of the agents in the distributed system. Aspects are used to implement this enforcement mechanism. The mechanism is implemented in layers, making use of the aspects' ability to inherit from other aspects. The topmost level is a general infrastructure for the enforcement mechanism specifying the architecture of a distributed system and the events (sending and receiving of messages by agents) in the system which are to be monitored. This aspect, represented in Figure 1, does not detail a specific policy to be enforced, rather the distributed architecture on which a policy will act. By using an aspect, not only the policy to be enforced is clarified, the domain of the problem is formalized and made explicit.

The second layer is a specific policy. Figure 2 is an example of a congestion control policy. While in the abstract model we were referring to general agents, here we have two types of agents - 'Client' and 'Server' (lines 4-8). Each agent has a set of attributes associated with it, these are described using an inner class. The ruling of the policy is based on these attributes. The monitoring of events is done using **around** advice on an abstract pointcut that specifies the

²The examples are implemented in AspectJ1.0

```

1. public abstract aspect DistributedEnforcementMechanism
2.     perthis(this(Agent)) {
3.
4.     public interface Agent {}
5.
6.     abstract pointcut send(Agent sender);
7.     abstract pointcut arrived(Agent receiver);
8.
9.     interface Ruling {
10.         boolean evaluateSendRuling(Agent sndr);
11.         boolean evaluateArrivedRuling(Agent rcvr);
12.     }
13.}

```

Figure 1: Distributed Architecture

event. The ruling is done using a ruling object, thus the ruling scheme is easily chosen and can be easily altered. Figure 3 describes a ruling in which if the time interval between two consecutive send event (by the same client) is smaller than a previously set delay, the client is blocked for the amount of time needed to make the ruling hold. A different scheme can be simply dropping that message.

Figure 2 shows the aspect’s code for implementing the policy described above. The aspect represents a congestion control policy that is **independent** of a specific system, thus representing a high-level design decision. A third level, that is not presented in this paper, is needed to “hook” this abstract representation onto a specific system. This is done by making the send and arrive events concrete with respect to that system.

```

1. abstract aspect CongestionControlPolicy
2.     extends DistributedEnforcementMechanism {
3.
4.     public interface Server {}
5.     public interface Client {}
6.
7.     declare parents: Server implements Agent;
8.     declare parents: Client implements Agent;
9.
10.    abstract pointcut send(Agent a);
11.    abstract pointcut arrived(Agent a);
12.
13.    class CCAAttributes {
14.        long clock() {...}
15.        long getDelay() {...}
16.        long getLastCall() {...}
17.        ...
18.    } // end of inner class CCAAttributes
19.
20.    Object around(Agent sender) : send(sender) {
21.        if ( ruling.evaluateSendRuling(sender) ) {
22.            proceed(sender);
23.        }
24.    }
25.    ...
26.}

```

Figure 2: Congestion Control Policy

Policies of distributed systems, which are actually policies

on the functionality of such systems, seem to be a logical extension to the already known architectures that can be described using AOP (under the constraints we stated).

2.2 Example: Intensive Care Unit: A Kernelized Structure

This section follows the intensive care unit example depicted in [10]. The system is composed of two parts: the kernel which is responsible for life support missions and the rest of the system. Three principles that must be followed by such systems are stated in [10]:

Principle 1 (Exclusive Access) The kernel should have exclusive access to the machines connected to the patient.

Principle 2 (Independence) The kernel should be independent of the rest of the system.

Principle 3 (Limited Interface) The kernel must provide an interface for use by the rest of the system.

We focus on the second principle - the kernel’s independence. This high level design decision must be made formal through the use of specific programming rules. Static as well as dynamic interactions between the kernel’s components must be taken into consideration. In order to maintain the kernel’s independence the following principles must be observed:

- **use:** kernel classes cannot use (be clients of) non-kernel classes.
- **inheritance:** kernel classes cannot inherit from non-kernel classes.

To monitor kernel to non-kernel interactions the kernel’s components must be explicitly marked so they can be referenced as kernel classes making the unmarked part the non-kernel components. The marking procedure is done by having each of the kernel’s classes implement the *Kernel* interface, this is done by using the introduction mechanism of AspectJ. *Kernel* is an empty interface used only for tagging purposes. Figure 4 is the aspect representing a partition of the system. Each class within the kernel is made to implement the *Kernel* interface. An evident drawback of this method is that a complete list of kernel classes must be explicitly specified and it is the implementor’s responsibility to update it as the kernel changes.

```

public boolean evaluateSendRuling(Agent sender) {
    CongestionControlPolicy.CCAAttributes attr = context.getAttributes();

    if ( sender instanceof Client ) {
        long wait = attr.getLastCall() + attr.getDelay() - attr.clock();
        if ( wait > 0 ) {
            try {
                Thread.sleep(wait);
            } catch (Exception e) {...}
        }
    }
    ...
}

```

Figure 3: "Wait" ruling for the congestion control policy.

```

abstract aspect KernelArchitecture {

    public interface Kernel {}

    declare parents: <kernel classes> implements Kernel;
}

```

Figure 4: Kernelized system: marking of the kernel part

2.2.1 Regulating Use Interaction

A class is said to be a *client* of another if it **declares** either an attribute, a local variable, or a formal parameter of that class[10]. This is a definition on the interaction between two classes, therefore a static one. This interaction, as is, cannot be regulated with current version of AspectJ. Thus, in order to regulate the use of a class by another we must express the *declare* interaction using dynamic events that will be enforced at compile time by AspectJ's *declare error* mechanism: A class declares an element of another class in order to generate an instance of that class, call a feature, or access a variable. These are dynamic events and therefore can be monitored using AspectJ. Figure 5 shows how such events are being regulated at compile time. The aspect restricts the use of non-kernel classes from within kernel classes. The first part (lines 3-10) of the aspect monitors generation of non-kernel instances, feature calls to features defined in non-kernel classes, and field access to non-kernel fields. The second part (lines 12-19) declares these events to be an error.

Using a combination of generation, feature calls, and field access events instead of the definition in [10] covers all actual uses that a class does with another. However, it does not cover the static compilation dependency. Thus, we cannot monitor and regulate such events. This event occurs when a class only defines a variable of another class, but does not make any use of it, leaving it as a pure compilation dependency.

2.2.2 Regulating Inheritance Interaction

Inheritance poses a rather different problem. The problem originates from the fact that AspectJ does not supply an explicit construct for regulating inheritance interaction. This means that in order to be able to deal with inheritance we need to interpret it via available means.

When a class *c1* inherits from class *c2* it assumes both types. Thus, **types** provide us with the means to handle inheritance. Unfortunately, the tagging mechanism, described in the previous section, is not sufficient in the case of a kernelized system: by tagging the kernel only, and then attempting to refer to a kernel class that inherits from a class outside of the kernel, we run into logical failure since this class is represented as both kernel and non-kernel (!Kernel). In a formal manner, referring to a class as being a member of a set defined as the intersection between the set of kernel classes and the set of !kernel classes leaves us with an empty set. In order to overcome the stated difficulty we need to explicitly tag the non-kernel classes as well by introducing a second type (line 4, figure6).

As discussed in the previous section, the static event "being of type" needs to be stated in dynamic manners. We rely on the fact that every class has a constructor and therefore the initialization of a constructor of a desired type may be considered as being of that type. The aspect is depicted in figure 6.

We soon discover, however, that this solution doesn't get us closer to our desired goal. In the terms we stated, a kernel class that inherits from a non-kernel class may be considered as an initialization of a class that is of Kernel and NotKernel types. However, even though this solution discovers the desired breaches of the inheritance regularity, it has a side effect of discovering a set of classes where non-kernel classes inherit from kernel classes. Another fault of this method is that only user classes can be tagged, thus making the enforcement mechanism applicable only for such classes.

We might attempt various manipulations to limit the set of produced answers to include only the actual breaches of the regularity. This, however, goes directly against the "say what you mean" principle that we have labored to maintain throughout our work. The inheritance regularity repre-

```

1. aspect CannotUse {
2.
3.     pointcut generation():
4.         call((!Kernel+).new(..)) && within(Kernel+);
5.
6.     pointcut method_call():
7.         call(* (!Kernel+).*(..)) && within(Kernel+);
8.
9.     pointcut field_access():
10.        (set(* (!Kernel+).*) || get(* (!Kernel+).*)) && within(Kernel+);
11.
12.    declare error: method_call():
13.        "method call from kernel class to non-kernel class";
14.
15.    declare error: field_access():
16.        "accessing non-kernel field from a kernel class";
17.
18.    declare error: generation():
19.        "creation of a non-kernel class within a kernel class";
20.}

```

Figure 5: Regulating use interaction

sented by this aspect becomes incoherent, which defies our overall goal of specifying regularities as aspects. As a side note, we could of course issue warnings on the whole set which results from the aspect in figure 6, and leave the decision up to the implementor, but this, obviously, defies the whole purpose of regularity enforcement.

2.3 Stylistic Constraints

Following [16] there exists another class of constraints, stylistic constraints. *All class names must begin with an upper case letter*, is an example of such a constraint. We believe that such limitations can be regarded as a crosscut in the application (i.e. over all class names), and therefore should be considered as an aspect of the system. However, AspectJ does not provide tools to declare aspects of such sort. The underlying model of AspectJ, as noted in preceding sections, refers to program events that occur at run-time (join points) and only partially supports static events. Stylistic constraints are an extreme example of static events - they are **events on text**.

3. DISCUSSION

In this paper we address the possibility of using AOP in general, and AspectJ in particular, in order to solve the problem of design enforcement. We surveyed a spectrum of problems one encounters when trying to apply AspectJ in various cases where design enforcement is necessary. The most important question that needs to be asked is whether the difficulties we encountered are due to a specific implementation of the AOP idea, or are they to show that AOP in general is inadequate for design enforcement?

This is an interesting question because there seems to be an underlying tension between programming languages that are used for **creation**, and law enforcing that is meant to prevent or limit creation. This tension can be observed in the AspectJ examples above: AspectJ provide tools for making a class inherit from another, but the opposite - preventing inheritance - is not supported. In general, we saw

that AspectJ supports the monitoring of dynamic events (i.e. events that "happen"), yet it lacks the concept of monitoring static events. When thinking of of system architecture, it is the **static** events that play a major role.

In [9] N.H. Minsky mentions several reasons why regularities cannot be enforced by means of a programming language.

1. Only very few types of regularities can be thus built into any given language;...
2. Regularities that do not have universal applicability should not be built into a general purpose language.
3. Programming languages usually adopt a module-centered view of software. ...
4. Language imposed regularities are obviously not effective for multi lingual systems, ...

AOP languages make us take another look at these reasons: The number of regularities which can be built into a language still remains small, but unlike non AOP languages, the whole purpose of some of these regularities, is to define new ones. These are low level regularities, that provide us with a mechanism for the definition and enforcement (by means of a compiler) of high-level design principles. As for Minsky's third comment, AOP makes it obsolete. The basis of AOP is to complement the module-centered approach with a mechanism for the definition of cross-module regularities.

Is AspectJ a sufficient solution? In this paper we showed that its current form is only partially adequate for design enforcement. Simultaneously, the examples we use shed light on the necessary additions to the language: for example, the same way we can monitor method calls using the *call* pointcut designator, we would want to have an *inherit* pointcut designator for monitoring inheritance and *declare* to monitor variable declaration. In general, the *pointcut* language

```

1. public aspect KernelArchitecture {
2.
3.     public interface Kernel {}
4.     public interface NotKernel {}
5.
6.     declare parents: <kernel classes> implements Kernel;
7.     declare parents: <non-kernel classes> implements NotKernel;
8.
9.     pointcut cannot_inherit():
10.         initialization((Kernel+).new(..)) && initialization((NotKernel+).new(..));
11.     declare error: cannot_inherit(): "inheritance law breached";
12.}

```

Figure 6: regulating inheritance

construct that currently provides us with a way to declare sets of join-points, should be extended to include the interaction between the static components (e.g classes, files) of the system.

There are problems with the demand for extending the language since the general tendency is to keep programming languages concise and to prevent cluttering. Where do we draw the line? How do we deal with enforcing style, or constraining native code? This is unclear, but intuitively some problems, like inheritance, are more pressing in terms of design enforcement.

When talking about where to draw the line it is important to consider not only the fact that we are dealing with a programming language, but also the purpose of this language. In a language that is meant for rule enforcement the cost of extension is relatively less important than the added benefit of incorporating the programming language and the enforcement tool under a single mechanism. This is because enforcing design regularities, however important, is not a wide spread practice. In order to do so a programmer needs to use a special tool, and usually learn a specification language on top of the programming language. Furthermore, the task of building software systems becomes more complicated, by using these extra tools which are not an integral part of the system at development. By using an AspectJ like language as the enforcement mechanism no new tool is introduced to the development environment, no special purpose language is needed for specifying regularities - they are specified using the language of the system, a language the programmer already knows and the enforcement is done by means of a compiler. This 'single tool' approach makes the process of enforcement an integral part of the system development.

The bottom line claim that we would like to make is that the problems we encountered were not a result of an incompetence in the general AOP idea but a consequence of it's particular implementation. AspectJ is currently the only of it's kind and is still undergoing major changes. The language was not designed for the purpose of regulating architectural decisions and thus lacks sufficient tools to accommodate this task. The range of difficulties we encountered in our attempts to implement regularities did not result from problems with the AOP concept. In fact, this concept proved to be exceptionally robust: We were able to express the regularities in AOP terminology, it is their realization that caused the difficulties. "The spirit indeed is willing, but the flesh is weak"(Matthew 26:40-42).

4. CONCLUSION

We have examined system design from the point of design enforcement and argued that the AOP idea can be used as a suitable means for design enforcement - The crosscutting nature of a design principle is what links these two ideas together. Yet, we have shown that some designs, especially those related to the static nature of the architecture, are not well addressed by contemporary tools.

5. FUTURE WORK

It is obvious that creating a tool which will be able to enforce both static and dynamic regularities by means of aspects is the direct continuation to this work. Yet, before embarking on such a task, more research as to the boundaries of such a tool (i.e. the interactions it will monitor) needs to be carried out.

6. REFERENCES

- [1] Elisa L. A. Baniassad, Robert J. Walker, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *Proceedings of the 21st International Conference on Software Engineering*, pages 120–131. ACM Press, 1999.
- [2] G. Booch, I. Jacobson, and J. Rumbaugh. The unified modeling language reference manual, 1999.
- [3] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Technical Conference Proceedings*, pages 99–115, Portland, OR, 10–13 1992. USENIX Assoc. Berkeley, CA, USA.
- [4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
- [5] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *Software Engineering*, 21(4):269–274, 1995.
- [6] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.
- [7] M. Lippert and C. Lopes. A study on exception

- detection and handling using aspect-oriented programming. In Proc. of ICSE, pages 418–427, 2000.
- [8] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.
- [9] Naftaly H. Minsky. Law-governed regularities in object systems. part 1: An abstract model. *Theory and Practice of Object Systems*, 2(4):283–301, 1996.
- [10] Naftaly H. Minsky. Law-governed regularities in object systems. part 2: the eiffel case. *Theory and Practice of Object Systems*, 1997.
- [11] Naftaly H. Minsky. Why should architectural principles be enforced?. in *Computer Security, Dependability, and Assurance*. Paul Amman et al., editors; IEEE Computer Society, 1999.
- [12] Naftaly H. Minsky and Victoria Ungureanu. Regulated coordination in open distributed systems. In *Coordination Models and Languages*, pages 81–97, 1997.
- [13] M. Sefika A. Sane and R.H. Campbell. Monitoring compliance of a software system with its high-level design model. In Proceedings of the 18th International Conference on Software Engineering (ICSE), March 1996.
- [14] Boris Bokowski, *Coffeestrainer: Statically-checked constraints on the definition and use of types in java*, ESEC / SIGSOFT FSE, 1999, pp. 355–374.
- [15] P. Devanbu, genoa a customizable, language and front-end independent code analyzer. In Proc. of 14th Int'l Conf. on Software Engineering (ICSE), pages 307–317. IEEE Press, 1992.
- [16] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss, *CCEL: A metalanguage for C++*, USENIX C++ Technical Conference Proceedings (Portland, OR), USENIX Assoc. Berkeley, CA, USA, 10–13 1992, pp. 99–115.
- [17] N.H. Minsky., *Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems*, Software Engineering and Methodology **9** (2000), no. 3, 273–305.