Towards an Analysis of Layering Violations in Aspect-Oriented Software Architectures

Mário Monteiro[1], Marcelo Moura[2], Sérgio Soares[3], Fernando Castor Filho[4]
Department of Computing and Systems, University of Pernambuco
Rua Benfica, 455, Madalena, 50.750-410, Recife, Pernambuco, Brazil
{mqm, mlmm, sergio, fernando.castor}@dsc.upe.br

Abstract. Empirical experiments with quantitative results are of great importance to analyze the benefits and drawbacks of aspect-oriented programming (AOP) in different contexts. Although many assessments of this new paradigm have been conducted in the last few years, only a small number of studies address specifically the impact of AOP on the architecture of a software system. This seriously hinders the adoption of AOP, as AOP techniques challenge the traditional notion of modularity on which most of the research on software architecture is based. More specifically, it is not clear what the effect of AOP on layered software architectures is. In this work, we present a study with the goal of analyzing the influence of AOP on violations of the layered structure of software architectures. We argue that the existing metrics for layering violations do not appropriately accommodate the notion of aspects. We explain the problems with these metrics and motivate the need to extend them to allow more precise quantitative evaluations of layering violations when aspects are involved. The target application of the study is a real-life web-based system that has been used in many other scientific studies.

Keywords: aspects, software architecture, layered software architectures, software evolution.

1 Introduction

Aspect-Oriented Programming (AOP) [9] is a software development approach that supports the modularization of crosscutting concerns in complex software systems. Although many assessments of AOP have been conducted in the last few years [15] [6] [12] only a small number of studies addresses specifically its impact on the architecture of a software system. This seriously hinders the adoption of AOP, as AOP techniques challenge the traditional notion of modularity on which most of the research on software architecture is based [1]. More specifically, it is not clear what the effect of AOP on layered software architectures is.

Layered software architectures were introduced by Djikstra [5] in the context of operating systems research. A layered system is structured as hierarchy of modules where each module (or *layer*) only can use services that the module located immediately below it offers. Layered architectures promote benefits such as maintainability and the ability to separately understand the parts of the system. Since

[1] Supported by FACEPE.
[2] Supported by CAPES.
[3] Partially supported by CNPq, grant #480489/2007-6.
[4] Partially supported by CNPq, grants #481147/2007-1 and #550895/2007-8.

their inception, layered software architectures have been widely adopted and there are many existing systems that are structured as sets of layers. Notable examples include most of the existing web-based information systems and network protocol stacks such as the Internet's [13].

In this work, we study the impact of AOP techniques on layered software architectures. We have conducted an empirical study targeting five evolution scenarios of a real-life web-based information system to better understand this impact. More specifically, we focus our analysis on layering violations, that is, situations where a layer is a client to another layer that is not below it or not adjacent to it. Although there are already studies on layering violations in the literature [17], to the best of our knowledge, none of them gives specific attention to aspects. Actually the study [17] is an important related work, since it provides a metric suit to quantitatively measure layering violations in source code. The contributions of this work are the following:

- A motivation for the need to adapt existing layering violation metrics to take aspects into account.
- Development of a framework, using the AspectJ language, to automatically measure layering violations in Java programs.
- Quantitative and qualitative evaluation of a software system in terms of layering violations. In this study, we consider five different evolution scenarios of the same system and employ two implementation approaches: Object-Oriented Programming (OOP) and AOP.

This paper is organized as follows. Section 2 introduces the setting of this study, whereas Section 3 presents its target application, Health Watcher [18]. Section 4 describes the notion of layer that we adopt in this work. This is necessary because architectural modules are often not explicitly materialized at the implementation level. Section 5 presents a framework that we have developed in order to collect information about dependencies between layers. Finally, Section 6 presents and analyzes the results of our study and Section 7 rounds the paper.

2 Study Setting

This study is divided into five major phases: (1) selection of a target application, including the selection of relevant evolution scenarios; (2) documentation and placement of each module in a specific layer; (3) analysis and motivation for the need to adapt existing layering violation metrics, so as to consider the effects of AOP; (4) development of a framework to automatically collect the metrics; (5) evaluation, both quantitative and qualitative, of layering violations in the target application, considering five evolution scenarios and two versions, one aspect-oriented and the other one object-oriented. Sections 3-7 describe each of these phases, respectively.

In order to analyze architectural layering violations in the web information systems domain, this work considers two different implementations techniques: AOP and OOP. We chose AspectJ [10] as a representative of AOP approaches, since it is the most mature and widely adopted amongst existing AOP languages. It is important to stress that our goal is not to compare different AOP mechanisms. Therefore, we attempt to be consistent in our use of AOP mechanisms and always opt for the simplest possible solution to solve design and implementation problems. Our main goal is to assess the impact that AOP technology has in a layered software

architecture, in terms of creating, removing, and/or changing layering violations. As a representative of OOP, we employ Java because it is often used to implement web-based information systems and it is the language from which AspectJ derives.

The quantitative metrics to determine layering violations [17] are very important in our study. They are the main tool we employ to analyze the influence that AOP and OOP have in terms of layering violations. We consider three types of layering violation: Skip-Call, Back-Call and Cyclic Dependency. We explain each type of violation using an example. Let *A* and *B* be two components in a software architecture. Assume that *A* invokes services from *B* and that *A* and *B* belong to layers *LA* and *LB*, respectively. In this scenario, we define the types of layering violations as follows:

- *Skip-Call*: Occurs if *LB* is located below *LA*, but they are not adjacent, i.e., there is some other layer, *LC,* between *LA* and *LB*.
- *Back-Call*: Occurs if *LB* is located above *LA*.
- *Cyclic Dependency*: Occurs if there is a cycle in the dependence graph formed by the dependencies amongst the layers of the system.

More details about these types of layering violations can be found elsewhere [17].

3 Target Selection

The first important decision in this study is the selection of the target application. We have chosen a typical web-based information system, called Health Watcher (HW) [18]. HW is a complaint registration system. Its major objective is to improve the quality of the services provided by the health care institutions, by allowing the public to register complaints about the quality of the service provided to them. In this study, we did not consider another target application, but we believe that more studies should be taken into account in order to collect more evidences considering our approach (see Section 7).

This system was selected due to the several reasons related to our study goals and constraints. First, it is a real and non-trivial application that has a pure Java version and an AspectJ version where some concerns, most notably distribution and persistence, are modularized through aspects. This factor can help us to better identify the effect that each implementation technology has on the study results. The HW design presents common quality requirements, e.g. web-based GUI, persistence, concurrency, distribution, and implementation technologies, e.g. Servlets, JDBC, and RMI. Besides, it has a good representativeness in terms of crosscutting and non-crosscutting concerns [8], i.e., there are a variety of real-life application concerns. This justifies the aspect-oriented implementation besides the object-oriented one.

Second, HW was developed based on classical n-tier architecture, a very well-known and widely used layered architecture. Therefore, we believe it is a good representative of real layered software systems.

Finally, one of the previous empirical studies targeting HW [8] applied a number of maintenance and evolution change scenarios to a baseline version of the system. This effort aimed to simulate a software development environment with realistic change scenarios. In this study, we selected the first five among these scenarios, as depicted in Table 1. Each scenario was applied to two different branches of HW: one purely object-oriented and the other one using AOP. Hence, each release of the system produces two different versions.

Table 1. HW evolution scenarios employed in this study.

Release	Scenario description
R1	Factor out multiple Servlets to improve extensibility.
R2	Ensure the complaint state cannot be updated once closed to protect complaints from multiple updates.
R3	Encapsulate update operations to improve maintainability using common software engineering practices.
R4	Improve the encapsulation of the distribution concern for better reuse and customization.
R5	Generalize the persistence mechanism to improve reuse and extensibility.

4 Identifying and Documenting Architectural Layers

Before we start measuring architecture layering violations, it is necessary to identify what are the layers in the system and to which layer each module belongs. First, we consider that each implementation unit in the source code is a module , i.e., a module can be a class, an aspect, or an interface.

Every module should belong to a certain layer [17]. Besides, considering a Layer L, argument, return, and error types of operations provided by modules on the border of L (invoked by modules from other layers) must be defined either in L or, more commonly, in shared data definition modules [2]. However, in order to measure architectural layering violations, we need to place shared data definition modules into a certain layer. Sometimes it is difficult to determine when to assign a module to a certain System layer. It is clear that a module responsible for persisting objects should be in the data layer, a module that manages user interaction, appearance, button events and menus should be in the graphical user interface (GUI) layer. However, it is not clear in which layer reusable elements on which modules located in many different layers of the system should be placed. Examples of such elements would be the `Account` class in a banking system or, in the case of HW, the `Complaint` class. We view these classes as offering services to modules located in many other layers of the system, even though they do not offer services in strict sense. Reusable classes usually do not require services from GUI nor business modules. Instead, basic classes are used by all of them. This line of reasoning led us to place these elements at the lowest layer of the hierarchy, which we shall call “Reusable Elements layer”.

The design choice of placing reusable elements in the lowest architectural layer is based on the assumption that skip calls are a less severe layering violation than back calls. This is intuitive if we consider that one of the motivations for the use of layered architectures is to be able to replace a certain layer by another one without affecting any lower layers. When a module makes a back call, it breaks this principle and, as a consequence, partially defeats one of the main benefits of layered architectures. If, on the other hand, a module at a layer *L* makes a skip call, this means that it depends on layers that are not adjacent to *L*, but are still below it. Hence, *L* can still be replaced without affecting layers located at lower levels. By placing reusable elements at the lowest layer, we avoid making back calls to them by introducing some skip calls. The UML Components software development method [3] takes a similar approach to separate reusable from non-reusable software components.

Another problem arises in the aspect-oriented systems domain. Aspects are modules and, therefore, should belong to a certain layer [17]. However, it is not clear to which layers the aspects belong. There are examples of crosscutting that fit appropriately within a layered structure, e.g. Distribution. Nevertheless, for simplicity, since this is still an ongoing work and most of the crosscutting concerns of Health Watcher cannot be matched to a specific layer, we argue that aspects do not belong to any of the layers. In fact, aspects that are intended to crosscut can easily cross layers, so arbitrarily assigning them to a layer obviously invites problems. Therefore, it is not straightforward to claim that aspects should be localized into layers. In this case, we have to treat the metrics defined in [17] differently when dealing with aspect-oriented systems.

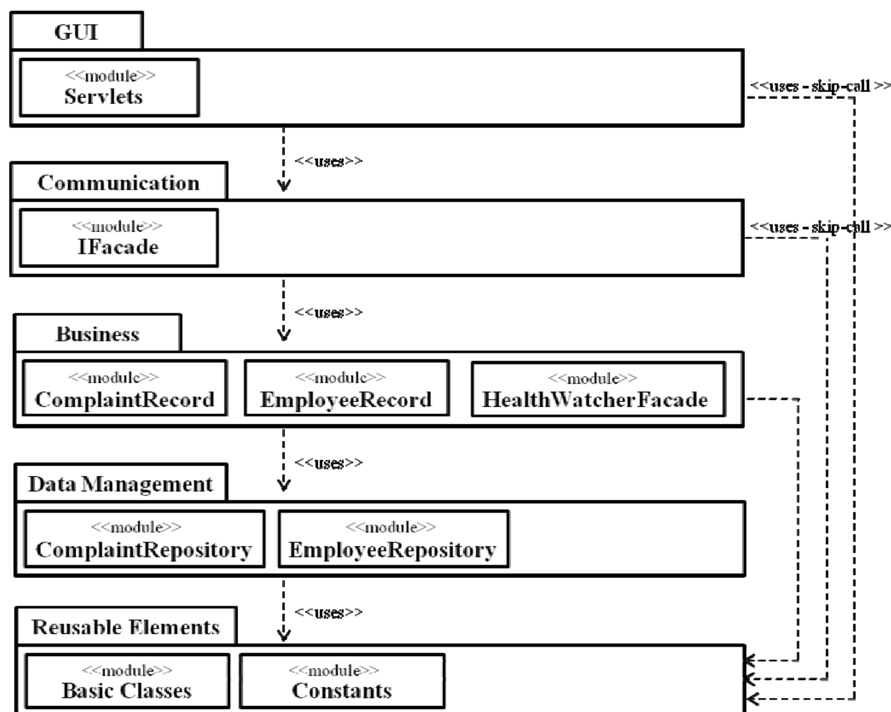

Figure 1. HW layer architecture.

The existing metrics for architectural layering violation [17] do not consider aspects. However, even though we believe, in this study, that aspects should not belong to any layers, there can still exist many dependencies from aspects to modules within the layers. This raises the question of whether aspects should be considered, for the purpose of counting skip-call, back-calls, and cycles. Our opinion is that aspects should not be considered for this purpose. These types of violation are inherently dependent on the location of a layer in the architectural layers hierarchy and aspects, considering Health Watcher System, are not part of any specific layer. It is undeniable that AOP still affects the results of our study, since their introduction changes some dependencies in the system, as discussed in Section 6. Nevertheless, ideally, the influence of aspects in layering violations should also be measured directly, in terms of dependencies on/from aspects, and not only indirectly, in terms of the modules that aspectization affects. This kind of measurement is important for

maintenance because changing a module on which an aspect depends might break the aspect. Currently, though, there is no metric suite that bridges the gap between a layered architecture and the crosscutting dependencies that AOP makes explicit and we consider this fertile ground for future research. The last section of this paper briefly elaborates on this topic.

In order to ease the identification and documentation process, we have developed a simple tool to assist us in associating modules to layers. According to the HW System documentation [8], the modules can be placed in four layers: (1) GUI, (2) Communication, (3) Business, (4) Data Management. As previously mentioned, we have also defined a fifth layer: the Reusable Elements layer. Every aspect implementation unit is registered so that it does not belong to any layer. After the registration of all modules, a configuration file is created with all registered information and then it is used as input by the framework (explained in detail in Section 5), in order to automatically collect the layering violation metrics.

All module registration was carried out carefully, with professional supervision (HW developers) and researchers with long-term experience on the development of the HW. As much as possible, we have employed automated tools to reduce the impact of human errors in the registration process. The correctness of this task is primordial, since it will directly affect all metrics.

Figure 1 depicts the HW architecture. Due to space constraints the figure shows only a few modules and examples of skip-calls violations.

5 Architectural Layer Violation Measurement Framework

After registering all implementation units mentioned in Section 4, we need to identify all the dependencies between modules. In this work, we adhere to the definition of dependency adopted by the UML [16]. The latter considers that a dependency between two elements implies that if one of them changes, the other one might have to change as well. For simplicity, we consider only the following five kinds of dependencies:

1. **Method calls**: Module A depends on Module B if A calls a method from B, or if A instantiates B.
2. **Field Access**. Module A depends on Module B if A reads some field from B.
3. **Field Assignment**: Module A depends on Module B if A performs an assignment to a field from B.
4. **Exception Handling**. Module A depends on Module B if A handles exception.

All these dependencies should be considered in the entire system. The identification of such dependencies should not be done manually, since this is an error prone and time-consuming task. Therefore, we developed a simple framework using the features available in AspectJ to identify all the dependencies automatically.

The framework provides two major functionalities: (1) identification of dependencies; and (2) metrics collection.

Identification of Dependencies. This functionality is responsible for determining all dependencies between modules and for creating a text file with this information. First of all, it is necessary to create a pointcut selecting all joint points that represent a

dependency on a given module (in this example, module `Address`), along with a `declare error` statement as shown in the code snippet below.

```
pointcut methCallAddress():call(* Address.*(..)) && !within(Address);
pointcut constructorAddress():call(Address.new(..))&& !within(Address);
pointcut getValueAddress():get( * Address.*) && !within(Address);
pointcut setValueAddress():set( * Address.*) && !within(Address);
pointcut handlerAddress():handler(Address) && !within(Address);
pointcut dependencies():methodCallAddress() || constructorCallAddress() ||
                        readFieldValueAddress() || assignFieldAddress()  ||
                        handleAddress();
declare error : dependencies() : "#Address#";
```

The declare error statement will cause a compilation error whenever a module (that is not `Address` itself) depends on it. Therefore, the compilation error log contains all dependencies from all modules to the module `Address`. If similar code is produced for every module of the system, the error log will contain all the dependencies for a specific project. The construction of the code snippet above can be automated for all modules from a project, since the only difference is the name of the module in each `pointcut` and `declare error` statement. In our study, we developed a small program which accesses each module from a given project in the source code folder and then generates dependency-tracking aspects for all implementation units. The result is set of aspects including a `declare error` statement for each module of the project. We treat the error log as the dependency graph and use it as input to metrics collection. In this study, the AspectJ error log is generated in the Problem View (using Eclipse/AJDT) and the contents (which is very simple to read and understand) is saved as a simple text file, where each line represents a dependency. In this case, a simple text parser is sufficient to obtain the graph dependency automatically.

Metrics Collection. This phases takes as input the list of layers in the system, including their order, set of modules associated to each layer (Section 4), and the dependency graph, and uses them to detect layering violations. The framework works by first picking a dependency of an arbitrary module A on a module B and then determining to which layer each of these modules depend. It then checks, for each dependency from a module A on a module B, whether: (i) A's layer is higher than B's; and (ii) A and B are adjacent. The first condition detects back calls, whereas the second one detects skip calls. The framework detects Cycles in the same way that cycles are usually detected in graphs [4]. This procedure is repeated for every dependency in the dependency graph and the result is a text file comprising layering violation metrics for the architecture of a software system.

6 Results Analyzes

This section reports and discusses the measurement results for the architecture layer violation principles. Table 2 shows, for each layer in both OO and AO version along the selected releases, the number of skip-call violations (SC), back-call violations (BC), number of modules with skip-call violations (NMSC), number of modules with back-call violations (NMBC), total number of modules. Table 3 also contains the back-call violation index (BCVI) for the entire system: BCVI(S), as well as the skip-call violation index (SCVI) for the entire system: SCVI(S). It is desirable that these

two indexes have the value 1 (maximum value), meaning that there are neither back-calls nor skip-calls in the entire system. A value near to 0 indicates violation to large extent. More information about these metrics can be found elsewhere [17]. In this study, we did not measure cyclical dependence violations, because the cycles identified in the selected releases we analyzed were between modules of the same layer, which is not a violation.

Table 2. Evaluation of architectural violations present in HW scenarios.

		SC		**BC**		**NMSC**		**NMBC**		**TNM**	
		OO	AO	OO	AO	OO	AO	OO	AO	OO	AO
Release 1	GUI	227	171	0	0	18	19	0	0	25	25
	Communication	10	10	0	0	2	2	0	0	5	7
	Business	26	15	12	0	5	3	1	0	9	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	0	0	0	0	0	0	21	21
Release 2	GUI	227	171	0	0	20	21	0	0	29	29
	Communication	10	10	0	0	2	2	0	0	5	7
	Business	26	15	12	0	5	3	1	0	9	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	0	0	0	0	0	0	21	25
Release 3	GUI	227	171	0	0	20	21	0	0	29	29
	Communication	10	10	0	0	2	2	0	0	5	7
	Business	26	15	12	0	5	3	1	0	9	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	0	0	0	0	0	0	33	38
Release 4	GUI	233	171	0	0	20	21	0	0	29	29
	Communication	10	10	0	0	2	2	0	0	5	7
	Business	26	15	15	0	5	3	1	0	9	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	24	0	0	0	3	0	35	37
Release 5	GUI	233	171	0	0	20	21	0	0	29	29
	Communication	10	10	0	0	2	2	0	0	8	7
	Business	26	15	0	0	4	3	0	0	8	8
	Data	0	0	0	0	0	0	0	0	28	28
	Reusable Elem.	0	0	24	0	0	0	3	0	35	37

* SC: Skip-Call violation; BC: Back-Call violation; NMSC: Number of Modules with Skip-Call violation; NMBC: Number of Modules with Back-Call violation; TNM: Total Number of Modules.

An interesting point to notice is that, in all releases, the number of skip-call in AO version is smaller. In fact, many skip-calls in the GUI layer are due to the handling of exceptions signaled by data layer, such as `ObjectAlreadyInsertedException`. Besides, there are also some skip-calls to the reusable elements layer, such as basic classes. In fact, the GUI layer provides text field components so that the user can register a complaint. The Servlets obtain the values from the text fields and instantiates a `Complain` object, resulting in a skip-call from GUI to reusable elements layer. In AO version, some exceptions thrown in data layer are treated with aspects, which uses AspectJ `declare soft` construct so that is not necessary to treat some exceptions in GUI layer. However, the exception is treated within the appropriate aspects, which does not result in skip-call violation because aspects do not belong to

any layer (as explained in Section 4). This contributes to the smaller occurrence of skip-calls in AO than in OO version.

Table 3. Skip-Call and Back-Call Violation Indexes in HW releases.

	Release 1		**Release 2**		**Release 3**		**Release 4**		**Release 5**	
	OO	AO	OO	AO	OO	AO	OO	AO	OO	AO
BCVI(S)	0.972	1	0.972	1	0.972	1	0.950	1	0.978	1
SCVI(S)	0.575	0.528	0.626	0.581	0.626	0.581	0.624	0.580	0.766	0.58

* BCVI(S): Back-Call Violation Index; SCVI(S): Skip-Call Violation Index.

The analysis made by this work found an interesting skip-call, that revealed to be business code mixed with GUI code. This is a very trivial issue, which occurred in a single location, and could be easily fixed by any developer. In this case, a business validation is done in a Servlet (GUI layer), whereas it should be done in a business layer module. This example shows that using the framework also helps to identify possible mistakes that can violate layer principles and might be passed unnoticed otherwise.

In business layer, there are skip-calls to reusable elements layer, because some record classes, such as `ComplaintRecord`, need to call methods from basic classes to verify if the object is already inserted in the `ComplaintRepository` class (data layer). In AO version, the number of skip-calls in business layer is smaller. In this case, there are some skip-calls to `ConcurrencyManager` module from business layer in OO, but in AO an aspect responsible for synchronization (`HWManaged-Synchronization`) takes care of such calls.

Even though the number of skip-calls in AO is smaller than in OO version, the SCVI(S) metric indicates the skip-call violations in AO version is more problematic than in OO. The reason is because SCVI(S) is based on the proportion of the number of skip-call compared to the total number of calls for each layer.

Some interesting back-calls are presented in OO version 4. In this case, the observer pattern [17] requires that basic classes (reusable elements layer) add a notify method, which treats repository exceptions (data layer). In AO version, on the other hand, this pattern is implemented with aspects, and therefore there are no back-calls. Many of these violations could be solved by a simple refactoring. For instance, the skip-calls from GUI modules that treat repository exceptions could be avoided by creating business exceptions that can be thrown whenever a repository exception appears. This new exception can contain business information and the repository exception can be passed as its cause. In fact, all violations can be easily checked by using our framework. This is a useful tool during software development for quickly identifying layer violations principles that could be passed unnoticed by the developer otherwise.

7 Conclusions and Future Works

In this work, we analyzed the differences between AOP and OOP in the Health Watcher System. In both versions, we could find some architecture layer violations that could suggest refactoring in order to conform to the layer architecture. We analyzed how AOP mechanisms influence the metrics and we motivate suggestions to

extend these metrics to analyze the dependency of aspects in layer architecture, quantitatively.

We discussed the problems involving the documentation related to the placement of certain modules into layers. In fact, this task is very important, since it directly influences the metrics on violation principles. Sometimes this task is not so simple, especially when dealing with reusable elements and aspects.

This study has some important limitations. However, this is an ongoing work and we expect to use the feedback from the workshop to improve the following restrictions. First, we are not taking into account the aspects dependency, since aspects do not belong to any layer. Second, besides the five selected evolution scenarios analyzed in this study, we can also evaluate the others evolution scenarios provided in [8], in order to perform conclusions more consistent and representative. Finally, we also pretend to apply these metrics in different systems in order to assess with more evidences on the impacts of AOP in layer architecture violation principles.

References

1. Bass, L., P. et al. Software Architecture in Practice, 2nd edition, Addison-Wesley, 2003.
2. Buschmann, F., et al. Pattern-Oriented Software Architecture: A System of Patterns. Wiley, 1996
3. Cheesman, J., et al: A Simple Process for Specifying Component-Based Software. Addison-Wesley, October 2000.
4. Cormen, T. H. et al. Introduction to Algorithms. 2nd Edition, MIT Press, 2001.
5. Dijkstra, E. W. The structure of THE-multiprogramming system. Communications of ACM, 11(5):341-346, 1968.
6. Figueiredo, E. et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. Proceedings of ICSE'08, Leipzig, Germany, 2008.
7. Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.
8. Greenwood, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. Proc. of Ecoop.07, Berlin, Germany, 2007.
9. Kiczales, G. et al. Aspect-Oriented Programming. Proceedings of ECOOP'07, LNCS 1241, Springer, pp. 220-242, 1997.
10. Kiczales, G., et al. Getting Started with AspectJ. Communications of the ACM, 44(10):59–65, October 2001.
11. Kulesza, U. et al, Quantifying the Effects of AOP: a Maintenance Study. Proceedings of ICSM'06, Philadelphia, Sep 2006.
12. Lobato, C., et al. Evolving and Composing Frameworks with Aspects: The MobiGrid Case. In Proceedings of ICCBSS 2008 - Volume 00 IEEE Computer Society, 53-62, 2008.
13. McClain, G. R., ed., Open Systems Interconnection Handbook. New York, NY: Intertext Publications McGraw-Hill Book Company, 1991.
14. Mezini, M. and Ostermann, K. Conquering Aspects with Caesar. Proc. of AOSD, pp. 90-99, Boston, USA, 2003.
15. Molesini, A., et al. On the Quantitative Analysis of Architecture Stability in Aspectual Decompositions. In Proceedings of WICSA 2008 - IEEE Computer Society, 29-38.
16. Rumbaugh, J., Jacobson, I., and Booch, G. 2004 Unified Modeling Language Reference Manual, the (2nd Edition). Pearson Higher Education.
17. Sarkar, S., et al. A Method for Detecting and Measuring Architectural Layering Violations in Source Code. In Proceedings of APSEC. IEEE Computer Society, DC, 165-172.
18. Soares, S., et al: Implementing distribution and persistence aspects with AspectJ. In Proceedings of. OOPSLA '02. ACM, New York, NY, 174-190.