Semantic Dependencies and Modularity of Aspect-Oriented Software

Alberto Costa Neto, Márcio de Medeiros Ribeiro, Marcos Dósea, Rodrigo Bonifácio, Paulo Borba
Informatics Center
Federal University of Pernambuco
Recife, Pernambuco, Brazil
{acn, mmr3, mbd2, rba2, phmb}@cin.ufpe.br

Sérgio Soares
Computing Systems Department
Pernambuco State University
Recife, Pernambuco, Brazil
sergio@dsc.upe.br

Abstract

Modularization of crosscutting concerns is the main benefit provided by Aspect-Oriented constructs. In order to rigorously assess the overall impact of this kind of modularization, we use Design Structure Matrixes (DSMs) to analyze different versions (OO and AO) of a system. This is supported by the concept of semantic dependencies between classes and aspects, leading to a more faithful notion of coupling for AO systems. We also show how design rules can make those dependencies explicit and, consequently, yield a more modular design.

1 Introduction

Quality in software design is essential to cope with the increasing complexity in software development. Modularity is one of the most desirable software attributes that contributes to quality. A prerequisite to improve such characteristic is the ability to identify, among design options, which can lead to a better design.

Aspect-Oriented Programming (AOP) [9] is well known as an useful technique to modularize crosscutting concerns by using a concept called aspects. However, since AOP is a relatively new approach, there is not yet consensus about how to evaluate designs or even about which dimensions of modularity are supported by AOP.

Some researchers have evaluated AO software with respect to Cohesion, Coupling, Size, Complexity and Separation of Concerns using different techniques like metrics [5] and dependence graphs [8, 18, 17]. Others use Design Structure Matrixes (DSMs) and Net Option Value (NOV) as an analysis model [16, 11] to compare alternative designs (OO against AO, for example).

In this paper, we also use DSMs to analyze the structure of different versions of the Health Watcher (HW) system [15], which is described in Section 2. In particular, we build the DSMs considering semantic dependencies between aspects and classes. This kind of dependence has not been deeply discussed by other works, but has a significant impact on dimensions of modularity such as parallel development of modules.

The main contributions of this paper are (Section 3):

- A reasoning about semantic dependencies between classes and aspects. We argue that those dependencies should be expressed as design rules, reducing the dependencies between modules and, consequently, promoting modularity.
- Applying and discussing the concepts presented in three versions (Object-Oriented, Aspect-Oriented and Aspect-Oriented with Design Rules) of a real software application.

2 Health Watcher

The Health Watcher (HW) is a real web-based information system originally implemented in Java and restructured to use AspectJ [9], a general purpose AO extension to Java. The system was developed to improve the quality of the services provided by health care institutions, allowing citizens to register complaints regarding health issues, and heath care institutions to investigate and take the required actions.

This system was selected because its design has a significant number of non-crosscutting and crosscutting concerns. Furthermore, it requires a number of common day-to-day design decisions related to GUI, persistence, concurrency.

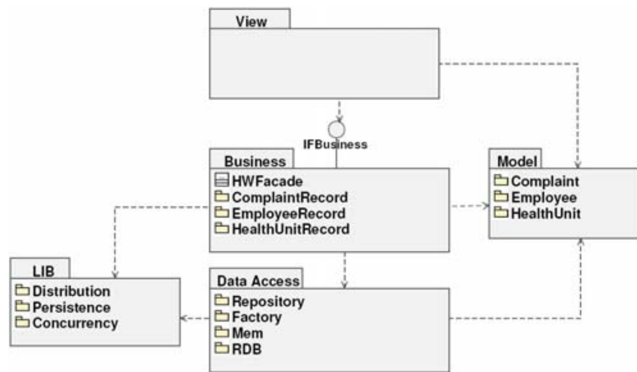

Figure 1. Base Architecture of the Health Watcher.

Figure 1 shows the base architecture of the HW system utilized in this work [6]. This architecture aims at modularizing user interface, distribution, business rules, and data management concerns. Below we describe the major architectural components of the HW system:

View Layer: related with the HW web interface. The implementation of this layer is based on Front Controller [1] and Command [4] patterns, using servlet and plain Java objects. The communication with the business layer is implemented with calls to the *IFBusiness*, which may be distributed or not.

Business Layer: responsible for business logic and transactional concern implementation. The *HWFacade*, which implements *IFBusiness*, is the unique point of interaction with this layer. This class uses *record* components to interact with the Data Access Layer.

Data Access Layer: responsible for abstracting the persistence mechanism following the Data Access Object pattern [1]. Some interfaces to manage data persistence are defined in this layer. Two implementations are available: the first one uses volatile memory whereas the second one is based on relational databases.

Model: responsible for implementing the *transfer objects*. These objects represent the core concepts of the application; transit between all architectural layers; and have few implementation logic.

Lib Components: represent reusable components that are useful to the implementation of concerns like persistence, distribution, and concurrency.

3 Assessing Health Watcher's Modularity

The concept of modularity applied to software development was first introduced by Parnas [12]. In his paper, the modular design is an attribute that enables better comprehensibility, changeability, and independent development.

More recently, Baldwin and Clark [2] have defined a theory which considers modularity as a key factor to innovation and market growth. This theory can be applied in different industries. It uses DSMs to reason about dependencies between artifacts and defend that tasks structure organization is closely related to them. Therefore, if two modules are coupled, their parallel and independent development is impossible.

Sullivan and Cristina Lopes have already applied Baldwin and Clark theory to assess software design, confirming the usefulness of the theory in this context [11, 16].

An essential step for DSM construction consists of selecting and clustering design parameters. A *design parameter* is any decision that needs to be made along the product design. The notion of dependency arises whenever a design decision depends on another. Using DSMs, each design parameter is disposed in both rows and columns of the matrix. A dependency between two parameters is marked with a X.

Figure 2 represents software components as parameters in a DSM. A mark in row B, column A represents that component B depends on component A . In the same way a X in row A, column B represents that component A depends on component B. Whenever this mutual dependency occurs, we have an example of cyclical dependency, which implies that both components can not be independently addressed.

	A	B	C
A		X	
B	X		X
C			

Figure 2. Example of dependencies in a DSM

Additionally, component B depends on C (expressed by a X in row B, column C) but C does not depend on any other component. Therefore, C can be independently developed and B can not be completely developed until C has been concluded.

Design parameters may have different abstraction levels. In software industry, some design decisions are related to process development, language, code/architectural style, and so forth. Moreover, if we consider implementation as design activities, software components like classes, interfaces, packages, and aspects could also be represented as design parameters.

Design Rules are parameters used as interfaces between modules and that are less likely to be changed [11]. In this way, they can promote decoupling of design parameters,

like component interfaces decrease the coupling between software components.

In our work, since one of the main focus is to compare the design structures of different HW implementations, only software components will be represented in DSMs. In addition, we do not use X to represent the dependencies. Rather than, we count the number of dependencies between design parameters and write this number in the matrix. A similar approach was presented elsewhere [13].

Initially our notion of dependency was associated with explicit references between components, like the instantiation of a class inside a method, inheritance or composition relationships[1]. Nevertheless, when we analyzed AO dependencies, we realized that this notion is not enough, as discussed in Section 3.3.

3.1 Health Watcher OO version

Although HW's architecture prevents some code tangling, it is not completely avoided. For instance, the *HWFacade* class (Figure 1) implements several concerns, including transaction management (persistence) and distribution. This architecture also fails to prevent code scattering. Almost all components must deal with the Exception Handling concern as well.

Figure 3 illustrates a DSM of the HW OO version. The result is a modular design with some dependencies within the *health watcher* components. The *Conceptual Model* design rule defines the core concepts (complaint, employee, health unit) that are referenced by the other models, as we can see in Column 1. Without establishing this design rule, the dependencies towards core concepts would be widespread throughout the matrix. Also, the *health watcher* components have dependencies with *lib* components.

	Design Rules						Lib			Health Watcher				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1 Conceptual Model	-													
2 IFDistribution		-												
3 IFPersistence			-											
4 IFView	4			-		1								
5 IFData	7				-									
6 IFPatterns						-								
7 Distribution							-							
8 Persistence								-						
9 Concurrency									-					
10 Model						2	4			-				
11 View	35			21						3	-			
12 Business	10		1	1	5	1		1	1			-		2
13 Data	19		7					12					-	
14 Patterns	12		1	1	16		8					2	12	-

Figure 3. Health Watcher OO DSM

This design structure follows the hierarchical pattern [2], which requires a sequential development: only after the *design rules* and *lib* components have been designed, it is possible to develop the other ones. The parallel development of such components is possible because there are only internal dependencies on them.

The design parameters *model* and *data* could also be developed in parallel because they do not depend on the others *health watcher* parameters. Afterwards, the development of the parameters *view*, *business* and *patterns* could be started.

3.2 Health Watcher AO version

Using AOP, some dependencies between *health watcher* components and *lib* components were transferred to *aspects* components. An example of such case is shown in Listings 1 and 2. Listing 1 presents the implementation of a method that saves an employee into a specific repository. Notice that Lines 4 and 11 make explicit references to an instance of *ConcurrencyManager* class.

Listing 1. Inserting an Employee Method

```
public void insert(Employee employee) throws
 ObjectNotValidException, ObjectAlreadyInsertedException,
 ObjectNotValidException, RepositoryException {
 manager.beginExecution(employee.getLogin());
 if (employeeRepository.exists(employee.getLogin())) {
   throw new ObjectAlreadyInsertedException(
     ExceptionMessages.EXC_JA_EXISTE);
 } else {
    employeeRepository.insert(employee);
 }
 manager.endExecution(employee.getLogin());
}
```

Listing 2 shows how the concurrency concern is implemented as an aspect. The aspect calls concurrency management methods from *lib* components at appropriate join points. This design solution provides a better separation of concerns and possibly enhances the support to parallel development. Most of direct calls to *lib* components methods were transferred from scattered points of *health watcher* components to the *aspect* components.

Listing 2. Synchronization Aspect

```
public aspect HWManagedSynchronization {
 private ConcurrencyManager
  manager = new ConcurrencyManager();

 public pointcut synchronization(Employee emp) :
  execution(* EmployeeRecord.insert(Employee))
  && args(emp);

 before(Employee emp) : synchronization(emp) {
  manager.beginExecution(emp.getLogin());
 }

 after(Employee emp) : synchronizationPoints(emp) {
  manager.endExecution(emp.getLogin());
 }
}
```

Figure 4 presents the DSM of the HW AO version. A substantial reduction of coupling between *health watcher* components with respect to *lib* components can be observed (Figure 3: Rows 10-14, Columns 7-9; Figure 4: Rows 9-13,

[1] Dependencies between application components and native API components are not considered

Columns 6-8). However, these dependencies were transferred to *aspect* components. Although the number of dependencies in the AO version is greater than OO version, the decoupling between *health watcher* components with respect to *lib* components turns the former more reusable and suitable to changes.

		Design Rules					Lib			Health Watcher					Aspects			
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	Conceptual Model	-																
2	IFPersistence		-															
3	IFView	4		-														
4	IFData	7			-													
5	IFPatterns					-												
6	Distribution						-											
7	Persistence							-										
8	Concurrency								-									
9	Model									-								
10	View	35		17						3	-	1						
11	Business	10		1	5							-						
12	Data	19	6					12					-					
13	Patterns	22	1		15								10	-				
14	Aspects - Concurrency	2						3	1		1	1	3	1	-			1
15	Aspects - Distribution	4		1			7				1	1				-		
16	Aspects - Persistence		4									7		1			-	
17	Aspects - Patterns	32		2							29	2		9				-

Figure 4. Health Watcher AO DSM

Looking to Figure 4, we can assume that the *health watcher* components could be developed before the *aspect* components. On the other hand, there are dependencies between these components that are not present in the source code, and consequently in the DSM, leading to sequential development of those components. These dependencies are discussed in the next section.

3.3 Syntactic x Semantic Coupling

Syntactic coupling in OO software components (classes and interfaces) occurs when there is a direct reference between them, such as inheritance, composition, methods signatures (parameters, return types, exceptions throwing), class instantiations, and so forth. This coupling causes compile errors whenever a component is modified or removed from the system, being thus easily detected.

In the same way of classes and interfaces, direct references can appear between aspects and other components, which means that aspects can also have such syntactic coupling. However, we report in this paper that syntactic coupling is not enough for a complete study of coupling in AO systems.

Thus, we argue that there is another kind of coupling that is not so easy to realize because it is not visible as the syntactic coupling. For this reason, we call this *semantic* coupling. Semantic coupling is a dependency which is not syntactically defined in the code, so that there is no explicit reference between system components (classes, interfaces, and aspects - in AO systems). Besides, this kind of coupling does not cause compile errors when removing or modifying components. Although there are semantic dependencies in OO systems, we did not find such dependencies without a corresponding syntactic representation. As we describe in the following examples, such coupling is not suitable when considering parallel development.

Three examples of semantic coupling are described in this paper, but certainly there are others. Although we do not provide a formal definition of semantic dependency, we argue that these (and others) examples should be previously categorized in order to help formulating a precise definition.

The first one is achieved by the use of wildcards in the aspects, which provide a concise way [10] to capture join points throughout the system. Listing 3 illustrates part of an aspect responsible for injecting the transaction management into the methods of the *HealthWatcherFacade* class. The use of wildcards in this example matches every facade method, except the static ones.

Suppose that a class developer is unconscious about the aspect and creates a method in the *HealthWatcherFacade* class that does not need transaction management. According to the Listing 3, the aspect will add the transaction concern into this method. In this case, the class will have an unintended behavior: executions of such method will create a new transaction which is no longer needed.

Listing 3. Transactional Methods pointcut

```
1  pointcut transactionalMethods():
2
3  execution(* HealthWatcherFacade.*(..)) &&
4
5  !execution(static * *.*(..));
```

The second example of the semantic coupling occurs when the class depends on the aspect to work correctly: if the pointcut of Listing 3 is changed, some methods could not be matched. Because they need transactional management, these methods will not work as desired. From the class developer point of view, this dependency is semantic because he can not see any dependency with the aspect. Moreover, we can observe that removing such aspect does not cause compile errors in the class which depends on it.

Although the use of wildcards does not impose a syntactic dependency, it is difficult to implement the transaction concern in parallel if there is no set of rules previously defined by the class and aspect developers (naming conventions, for example) so that class and aspect can work correctly.

The third example is about dynamic join points. When using the *cflow* or *cflowbelow* pointcuts, for example, we do not know in advance which points of the program will be matched at compile time. Therefore, we believe that this is the worst case to identify the semantic coupling because it will only be realized at execution time. In this case, the parallel development is not possible unless the details of the program control flow intercepted by the aspects are known

by the aspect developer.

Semantic coupling is highlighted in Figure 5. According to this concept, dependencies so far hidden between *aspects* and the *health watcher* components have arose. Moreover, due to the cyclical dependencies between the parameters inside these modules, it is difficult to independently evolve the design.

	Design Rules					Lib			Health Watcher					Aspects			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1 Conceptual Model	-																
2 IFPersistence		-															
3 IFView	4		-														
4 IFData	7			-													
5 IFPatterns					-												
6 Distribution						-											
7 Persistence							-										
8 Concurrency								-									
9 Model									-					1	10		7
10 View	35		17						3	-	1				7		4
11 Business	10		1	5							-			1	1	8	
12 Data	19	6					12					-		3			
13 Patterns	22	1		15								10	-				2
14 Aspects - Concurrency	2						3	1		1	1	3	1	-			1
15 Aspects - Distribution	16		1			7				11	1				-		
16 Aspects - Persistence		4									7		1			-	
17 Aspects - Patterns	32		2							3	31		9				-

Figure 5. Health Watcher AO DSM with semantic coupling

When considering only syntactic dependencies we can derive inconsistent analysis. Consequently, we concluded that it is necessary to improve the existing AOP coupling metrics [14, 18]. The next section proposes using design rules in order to reduce the syntactic and semantic dependencies between *aspects* and *health watcher* components.

3.4 Health Watcher AO with new Design Rules

In this section we will discuss the use of design rules as a strategy to establish and document the dependencies between classes and aspects. These design rules must be specified during the design activities.

Some examples of constraints present in the new design rules are:

- All classes in *model* components must be serializable, in order to support the distribution concern;
- The method used to insert an employee must deal with synchronization; and
- All business facade methods must have a transactional management support.

Figure 6 presents the DSM of the HW AO version with new design rules that decouple classes and aspects. The semantic dependencies between classes and aspects were

	Design Rules									Lib			Health Watcher					Aspects			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1 Conceptual Model	-																				
2 IFPersistence		-																			
3 IFView	4		-																		
4 IFData	7			-																	
5 IFPatterns					-																
6 DRConcurrency						-															
7 DRDistribution							-														
8 DRPersistence								-													
9 DRPatterns									-												
10 Distribution										-											
11 Persistence											-										
12 Concurrency												-									
13 Model						1	10		4				-								
14 View	35		17				7		4				3	-	1						
15 Business	10		1	5		1	1	7							-						
16 Data	19	6				3					12					-					
17 Patterns	22	1		15					2							10	-				
18 Aspects - Concurrency	2					3					3	1						-			1
19 Aspects - Distribution	16		1				2												-		
20 Aspects - Persistence		4						3												-	
21 Aspects - Patterns	32		2						3												-

Figure 6. Health Watcher AO DSM with new Design Rules

converted to dependencies between classes and the design rules, as shown in the DSM (Rows 13-17, Columns 6-9). Moreover, the dependencies between aspects and classes were removed, originating dependencies between aspects and design rules.

Using design rules requires both *aspect* and *base code* developers agreement. In this way, it promotes the decoupling between *health watcher* and *aspect* components, allowing their parallel development.

4 Related Work

Metric Suites - Zhao uses the concept of dependence graphs that represents various dependency relations in a program to create a dependence model for AO software. Based on this model, he proposed methods to assess the complexity [17], coupling [18], and cohesion [8]. Santanna et al [14] presented an assessment framework for AO software that consisted of an extension to the metrics suite (known as CK metrics) proposed by Chidamber and Kemerer in [3]. They proposed some metrics for measuring separation of concerns (useful in AO context) and reviewed the size, coupling, and cohesion to consider aspects. Subsequent work evaluated their framework in practical case studies, like the GoF Design Patterns [5]. We refined the notion of coupling presented by the aforementioned papers to consider semantic coupling, since some kinds of dependencies were not addressed by these works.

DSM - Lopes and Bajracharya [11] used DSM and NOV to compare the modularity achieved by different design options. They concluded that aspects can increase the value of an already modularized design. We did not measure the HW NOV but we discussed the dependencies observed in the DSMs. On the other hand, we explored the concept of dependencies between aspects and classes in more detail and confirmed the importance of establishing design rules.

XPI - Sullivan et al [16] presented a comparative analysis between an AO system developed following an oblivious approach, with the same system developed with clear design rules that document interfaces between classes and aspects. Griswold et al [7] showed how to transform part of the design rules into a set of aspects Crosscutting Programming Interfaces (XPIs) that are useful to document and check part of the design rules (contracts). In our work, the design rules responsible for dealing with semantic coupling could be mapped into XPIs.

5 Conclusion

We have presented an analysis of the different versions of the Health Watcher system using DSMs. We started from the OO version and compared it with the AO version. Also, we presented the notion of syntactic and semantic coupling and showed a different DSM including both of them. Usually, only the first one is considered for coupling measurement, but in our opinion it is not enough for a complete study of the coupling when considering AO systems.

Our analysis using DSMs showed that without considering the Semantic Coupling it is impossible to achieve the independent development, an essential feature, according to Parnas [12], to categorize a software as modular.

We observed that DSMs are useful to reason about dependencies between a significant number of components, assisting the tasks structuring activity.

Finally, a version with design rules was analyzed and we confirmed that it provides a better modularization than previous versions. Such version showed how design rules can make those dependencies explicit and, consequently, yield a more modular design.

6 Acknowledgments

We would like to thank CNPq and CAPES, Brazilian research funding agencies, for partially supporting this work.

References

[1] D. Alur, D. Malks, and J. Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[2] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, March 2000.

[3] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, January 1995.

[5] A. Garcia, C. SantÁnna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 36–74. Springer, 2006.

[6] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dósea, A. Garcia, N. Cacho, C. SantÁnna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proceedings of the 21st European Conference on Object–Oriented Programming, ECOOP'07*, July 2007. to appear.

[7] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[8] B. X. Jianjun Zhao. Measuring Aspect Cohesion. In *Proc. Fundamental Approaches to Software Engineering (FASE'2004)*, 2004.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, 1997.

[10] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[11] C. V. Lopes and S. K. Bajracharya. An Analysis of Modularity in Aspect-Oriented Design. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 1–35. Springer, 2006.

[12] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[13] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 167–176, New York, NY, USA, 2005. ACM Press.

[14] C. Santánna, A. Garcia, C. Chavez, C. Lucena, and A. von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: A Assessment Framework. In *Proc. of Brazilian Symposium on Software Engineering (SBES'03)*, pages 19–34, October 2003.

[15] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'2002*, pages 174–190, Seattle, USA, 4th–8th November 2002.

[16] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 166–175, New York, NY, USA, 2005. ACM Press.

[17] J. Zhao. Towards A Metrics Suite for Aspect-Oriented Software. Technical Report SE-136-25, Information Processing Society of Japan (IPSJ), March 2002.

[18] J. Zhao. Measuring Coupling in Aspect-Oriented Systems. In *10th International Software Metrics Symposium (Metrics'04)*, 2004.