Specifying Design Rules in Aspect-Oriented Systems

Marcos Dósea[1], Alberto Costa Neto[1], Paulo Borba[1], Sérgio Soares[2]

[1] Informatics Center – Federal University of Pernambuco
Caixa Postal 7851, 50740-540 – Recife – PE – Brazil

[2]Computing Systems Department – Pernambuco State University
Rua Benfica, 455, Madalena, 50720-001 – Recife – PE – Brazil

{mbd2,acn,phmb}@cin.ufpe.br, sergio@dsc.upe.br

Abstract. *Modularization of crosscutting concerns is the main benefit provided by Aspect-Oriented constructs. However, current AO languages do not address class modularity adequately. In order to achieve both class and crosscutting modularity, Design Rules for AO Systems should be defined. In this work we propose a language to specify Design Rules that establish the minimum requirements to enable the parallel development of class and aspects. Beyond the modularization improvement, the language creates a simpler and unambiguous specification, supporting the development of mechanisms for automatically checking the specified rules and making easier the using of the parametrization mechanisms.*

1. Introduction

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] has been proposed as a technique for modularizing crosscutting concerns. Previous work [Ribeiro et al. 2007] presented an analysis about Class and Crosscutting Modularity using different versions of a system both OO and AO, and confirmed that Crosscutting Concerns localized in single modules (aspects) provide better crosscutting modularity. On the other hand, this approach does not provide class modularity because in order to reason about any class it is necessary to consider all aspects implementations.

This weakness can be mitigated, achieving also class modularity, by using adequate Design Rules between classes and aspects, which are necessary to reduce syntactic and semantic dependencies [Neto et al. 2007] in AO systems. Design rules are not just guidelines and recommendations: they generalize the notion of information hiding interfaces and must be rigorously obeyed.

The use of Design Rules has been discussed in other works but none of these works propose a final solution where they can be described in an unambiguous manner and utilized not only for checking but as a guideline for developers since the initial phases of the development process. In this work we discuss about the problems found in some of these proposals (Section 2).

The main contribution of this paper is a Design Rule specification language (Section 3) that improves class and crosscutting modularity. This language defines a more modular design to decouple classes and aspects through the establishment of the minimum requirements necessary to work in parallel. These requirements are checked statically. Beyond all the advantages cited so far, the usage of a specific language for this purpose brings the following advantages:

- Allows the description of the Design Rules in a simple and unambiguous manner, making easier the development of mechanisms for an automatic checking of the rules in the written code.
- Provides a guideline to be utilized since the initial development phases of the components, specifying the essential constructions so that each developed component can have the proper functionality.

The language was initially evaluated in real systems promoting a simpler and unambiguous specification. Moreover, it seems to improve the opportunities for reuse in comparison to other approaches (Section 4).

2. The Problem

In this section we discuss some works related to the modularity problem presented in Section 1. The concept of modularity applied to software development was first introduced by Parnas [Parnas 1972]. The following quality attributes are expected in a modular design: *Comprehensibility*, *Changeability*, and *Independent Development*.

Regardless of the fact that the OO approach reach *Class Modularity*, OO applications design usually results in tangled and scattered code, due to the existence of crosscutting concerns, reducing the degree of modularity. AOP was proposed to modularize these crosscutting concerns, but constructions supported by AspectJ [Kiczales et al. 2001] like languages may lead to high coupling between classes and aspects, because aspects are usually dependent on classes implementation details.

Listing 1. OO Display Update Example.

```
1  interface IShape {
2      public void moveBy(int dx, int dy);
3  }
4
5  class Point implements IShape {
6      private int x, y;
7
8      public void moveBy(int dx, int dy) {
9          setX( x + dx )
10         setY( y + dy );
11         Display.update();
12     }
13 }
14
15 class Line implements IShape {
16     private Point p1, p2;
17
18     public void moveBy(int dx, int dy) {
19         p1.setX( p1.getX()+dx );
20         p1.setY( p1.getY()+dy );
21         p2.setX( p2.getX()+dx );
22         p2.setY( p2.getY()+dy );
23         Display.update();
24     }
25 }
```

The Listing 1 shows an implementation of the non-crosscutting concerns *Point* and *Line*. For simplicity, constructors and accessors (get and set) methods were omitted. In this implementation we observe the scattering of calls to the method *Display.update* tangled with shape positioning adjustment. That makes difficult reusing the class and suggest that it is necessary to improve the modularization somehow.

Listing 2. Display Update Aspect.

```
1  public aspect UpdateSignaling {
2      pointcut change():
3          execution( void IShape.moveBy(int, int) );
4
5      after() returning: change(){
6          Display.update();
7      }
8  }
```

The Listing 2 shows an aspect implementation of the crosscutting concern *Display Update*. This design provides better crosscutting modularity because calls to *Display.update()* would be removed from classes *Point* and *Line* (lines 11 and 23 from Listing 1) and localized in a single place (*UpdateSignaling* aspect).

Although creating an aspect to modularize the crosscutting concern *Display Update* improves the crosscutting modularity, it breaks the class modularity, because class developers should be concerned about the aspect so that, for example, the developers of *Line* and *Point* classes do not call again the *Display.update()* method within *moveBy* method. This situation exposes some modularity problems: (1) the comprehensibility is compromised, since two modules (class and aspect) should be studied in order to understand the concern and how they interact (required join points can not be changed, for example); and (2) the parallel development and changeability is problematic, because one developer can implement unintended behavior into a module which, though it is not under his responsibility, might break the system.

This weakness can be mitigated by using adequate design rules between classes and aspects. Design Rules are not just guidelines or recommendations: they must be rigorously obeyed in all phases of design and production [Baldwin and Clark 2000]. They are design parameters used as interfaces between modules that are less likely to be changed [Lopes and Bajracharya 2006]. In this way, they can promote decoupling of design parameters, like interfaces decrease the coupling between software components. An example of a Design Rule is: ”every class from a package must be Serializable”.

The use of Design Rules has been discussed in other works. Sullivan *et al* [Sullivan et al. 2005] used Design Rules specified in natural language and in a subsequent work (Griswold *et al* [Griswold et al. 2006]) proposed a way of specifying them using AspectJ constructions so that they could be automatically checked. Some of the shortcomings of the first approach are that they can be imprecise, are difficult to enforce, and check. On the other side, trying to express the Design Rules using AspectJ results in complex verification aspects. Chavez *et al* [Chavez et al. 2006] proposed a visual model that express some kinds of Design Rules. Despite a less ambiguous specification, the model does not allow the automatic verification of the code. Morgan *et al* [Morgan et al. 2007] presents a language, inspired by the pointcut language in AspectJ to declaratively encode design rules to OO systems. However, this language is useful only in OO systems and does not permit a clear definition of the rules for the parallel development of components.

In this paper, we propose a language to specify design rules in order to reduce syntactic and semantic dependencies between aspects and classes. This language allows to specify the set of minimal requirements that enable the parallel development, by reducing the coupling between such components and promoting modularity. By using the language

it is possible specify the Design Rules in unambiguous form and automatically verify if they are being followed by both class and aspect developers. Moreover the language provides a guideline since the early phases of the system development.

3. Design Rules Specification Language

In this section we present a Design Rule specification language. The main objective of this language is to decouple syntactically and semantically classes and aspects, maximizing independent development opportunities. Through the definition of Design Rules we argue that both class and aspect developers can work independently if a minimum set of constraints is defined and respected.

Listing 3. Display Update Design Rule.

```
1  dr DRUpdateShape {
2      interface IShape {
3          void moveBy(int dx, int dy);
4      }
5
6      class Shape implements IShape {
7          void moveBy(int dx, int dy) {
8            xset(IShape+.*);
9          }
10     }
11
12     class Display {
13         void update();
14     }
15
16     aspect UpdateSignaling {
17         pointcut change():
18             execution( void IShape.moveBy(int, int) );
19
20         after() returning: change(){
21             xcall( void Display.update() );
22         }
23     }
24 }
```

Listing 3 contains a Design Rule specification for the Display Update concern shown in the section 2. The Design Rule *DRUpdateShape* enforces that it is necessary to define an interface *IShape* containing at least the method *moveBy* (Lines 2-4). Also, all classes acting as a *Shape* must implement *IShape* and state changes on attributes are allowed only within the *moveBy* method (Lines 6-10). It also requires that a class *Display* with a method *update* must exist (Lines 12-14). Additionally, it requires the existence of the aspect *UpdateSignaling* with a pointcut *change*. This pointcut must be referred by an *after returning* advice that is the exclusive point, among the components specified by the Design Rule, allowed to call the method *update* from class *Display* (Lines 16-23).

The Design Rule *DRUpdateShape* specifies the minimum requisites that each component developer must know about the others, hence allowing their independent development. This specification uses a language similar to that used in the development of the components, making the specification much simpler. Comparing with Griswold *et al* [Griswold et al. 2006] approach, it would be necessary approximately twice lines of code to specify the XPI, yet, many rules would still being expressed in natural language.

The Listing 4 shows the Design Rules using the Griswold *et al* [Griswold et al. 2006] approach. This approach allows only to check if the Design Rules are being followed, but requires from the class developer deep knowledge

about AspectJ constructions, besides it does not guide developers. Our language supports the declarative specification of Design Rules adopting a syntax similar to the used by both developers.

Listing 4. Display Update XPI.

```
1
2  public aspect XDisplayUpdate {
3
4    /* The purpose of the joinpoint() PCD is to expose all and only Shape abstract
5       state transitions. We require that all such transitions be implemented by calls to
6       Shape mutators with names that match the PCDs of this XPI, and we assume that
7       any such call causes such a state transition. Advisors of this XPI may not change
8       the state of any Shape directly or indirectly. The topLevelJoinpoint() PCD exposes
9       all and only "top level" transitions in the abstract states of compound Shape
10      objects. */
11   public pointcut joinpoint(Shape s):
12     target(s) && call(void Shape+.moveBy(..));
13
14   public pointcut topLevelJoinpoint(Shape s):
15     joinpoint(s) && !cflowbelow(joinpoint(Shape));
16
17   protected pointcut staticscope():
18     within(Shape+);
19
20   protected pointcut staticmethodscope():
21     withincode(void Shape+.moveBy(..));
22 }
23
24 /* Checks the contracts for the XDisplayUpdate XPI. */ public
25 aspect FigureElementChangeContract {
26
27   /* PROVIDES: XPI matches all calls and only calls to Shape mutators */
28   declare error:
29     (!XDisplayUpdate.staticmethodscope() &&
30      set(int Shape+.*)): Contract violation: must set Shape field inside
31                          setter method!;
32
33   /* REQUIRES: advisers of this XPI must not change the state of any Shape object */
34   private pointcut advisingXPI():
35     adviceexecution();
36
37   before(): cflow(advisingXPI())
38     && XDisplayUpdate.joinpoint(Shape) {
39     ErrorHandling.signalFatal(Contract violation: advisor of
40       DisplayUpdate cannot change Shape instances);
41   }
42 }
```

The language supports the specification of the *structural rules* of classes, interfaces and aspects, enabling also the establishment of *behavioral rules*. These rules are valid within classes and aspects in both methods and advices.

The Table 1 shows the *behavioral rules* provided by the language. The scope of these rules includes classes and aspects methods and advices. Every rule initiated by 'x' guarantees that the rule will be followed only in the defined scope and this will not be possible in any other location. For example, the *xcall* rule guarantees that the method will be called exclusively within the scope which it was defined and no other scope among the components specified by the Design Rule will be call this method. In case that the rule *xcall(method)* exists in more than one scope, then this call can only be made within these defined scopes.

These rules are also useful to guarantee, for example, that a method must not be called in a given scope, by using the negation operator (!).

Table 1. Behavioral Rules provided for the Design Rule Language.

Rule	Description
call(method)	Must have a method call within the defined scope.
xcall(method)	Must have a method call exclusively in the defined scope.
set(attribute)	Must have an attribute state change within the defined scope.
xset(attribute)	Must have an attribute state change exclusively in the defined scope.
get(attribute)	Must have an attribute read within the scope.
xget(attribute)	Must have an attribute read exclusively in the defined scope.

Listing 5. Display Update Design Rule.

```
1  class Shape implements IShape {
2      void moveBy(int dx, int dy) {
3        !call(IShape.moveBy(int dx, int dy));
4      }
5  }
```

Listing 5 shows an improvement on the Design Rules previously established, forcing the inexistence of calls to method *moveBy* within it.

In order to express that classes and aspects must follow a Design Rule, we have chosen to include explicit references from each component to the Design Rules that it implements. It is important to note that although this breaks the obliviousness principle, we argue that even without the explicit references, the class developer would have to be aware of all aspects presents in the system. Using our approach, he can be oblivious about aspects, but must be aware of the constraints contained in the Design Rules.

Listing 6. Components.

```
1  interface IShape implements DRUpdateShape {
2      // original code
3  }
4
5  class Display implements DRUpdateShape {
6      // original code
7  }
8
9  aspect UpdateSignaling implements DRUpdateShape {
10     // original code within the pointcut description
11 }
12
13 class Line implements DRUpdateShape(Shape) {
14     // original code
15 }
16
17 class Point implements DRUpdateShape(Shape) {
18     // original code
19 }
```

The Listing 6 depicts the behavior of a concrete implementation of the components complying with a Design Rule *DRUpdateShape* shown in the listing 3. When the component name matches the name of a component specified by the Design Rule, for example like class *Display* (Line 5-7)), it must follow the rules established for the component Display. Moreover, when the component name differs, we must explicitly inform (between parentheses) the name of the corresponding component specified in the Design

Rule. The class *Line* (Line 13-15), for example, must follow the rules specified for the component *Shape*.

Furthermore, the language allows that several components implement the same Design Rule assuming the same function. In the Listing 6, it could exist, for example, different implementations for the aspect *UpdateSignaling*. Generally, this is necessary when there is a possibility of different configurations for the same system.

To indicate which components are going to be used in a given instance of the system, the specification of the Design Rule configuration module is needed. Through the module it is also possible to create different system configurations where each module would register one configuration option.

Listing 7. Configuration Module Example.

```
1  module ModuleUpdate implements DRUpdateShape {
2      IShape display.IShape;
3      Display display.Display;
4      UpdateSignaling display.UpdateSignaling;
5  }
```

The Listing 7 shows an example of module configuration for the Design Rule specified in the Listing 3. This module indicates which classes or concrete aspects will be used for the Design Rule instantiation *DRUpdateShape*. For example, the aspect *UpdateSignaling* specified in the Design Rule will be bounded to the aspect with the same name that is inside the package *display*.

3.1. Parametrization

The language also allows the specification of parameterized Design Rules. The parametrization is important for the specification flexibility and reuse.

The Listing 8 describes a partial implementation of the Design Rule *DRUpdateShape* where the update method is parameterized. This description makes the nomenclature that can be used by the method flexible and also allows that numerous methods can be considered as update methods. The concrete value of each utilized parameters will always be specified within the configuration module, thus, it is possible to create different system configurations by modifying only the parameters values within the module.

Listing 8. Parametrization Example.

```
1  dr DRUpdateShape{
2      aspect UpdateSignaling {
3          pointcut change():
4              execution( void IShape.moveBy(int, int) );
5
6          after() returning: change(){
7              xcall( <update_method> );
8          }
9      }
10     ...
11
12     class Display {
13         <update_method>;
14     }
15 }
```

The Listing 9 show the configuration module of the parameterized Design Rule. The value of the parameter *update_method* is associated using the reserved word *where*

inside the *Display* component specification. The parameter value could also be associated inside the *UpdateSignaling* component, but this value must be specified by only one of these components.

Listing 9. Configuration Module with Parametrization Example.

```
1  module ModuleUpdate implements DRUpdateShape {
2      IShape display.IShape;
3      Display display.Display where
4          update_method: void update();
5      UpdateSignaling display.UpdateSignaling;
6  }
```

3.2. Design Rule Specification Language Meta-Models

This section presents the language and the configuration module meta-models that show other possible Design Rules the language can describe. They are not explored in detail due to lack of space.

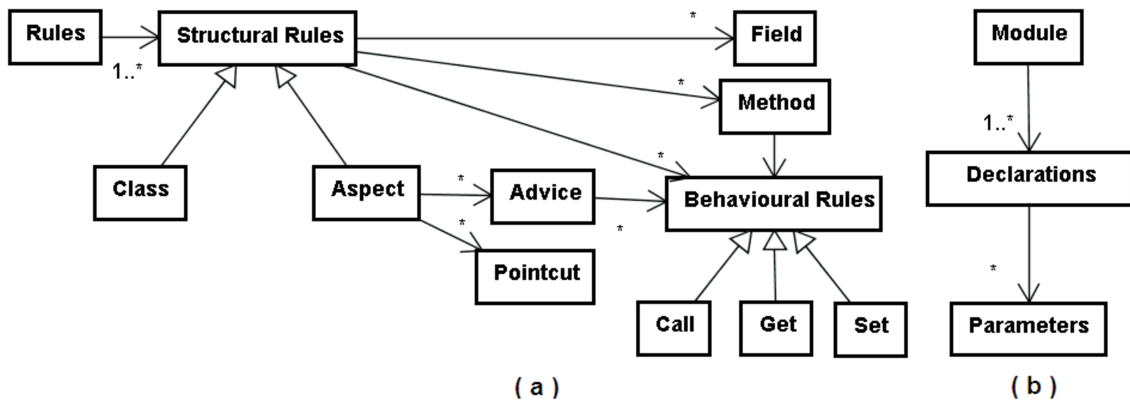

Figure 1. Design Rule Language and Configuration Module Meta-Models

Figure 1(a) shows the Design Rules specification language meta-model. A rule description is composed by one or more structural rules. They are used to define structural constraints for classes and aspects. Thus, a class can define fields, method signatures and behavioral rules. This behavioral rule can be defined within classes and aspects scope or within methods and advices. When a behavioral rule is defined within the scope of a class or aspect it means that it has to be executed at any place, for example, if a rule *call* is used within the scope of a class means that the required method must be called at least once in any place this class. The Figure 1(b) depicts the configuration module meta-model used to describe a configuration option of the specified components by Design Rule. The configuration module define which components will follow a particular structural rule (*Declaration*) and which parameters (*Parameter*) are informed.

For simplicity we do not detail the parametrization mechanism in this model. This mechanism can be used to parameterize the method signature and fields. For example, we can use the rule *set(field)* to allow that a particular field is updated. This field could be parameterized and the concrete value would be specified by configuration module.

4. Evaluation

This section describes our initial experience in using the proposed language to specify the Design Rules in two real AO systems. We focused on defining Design Rules that enable

parallel development, but there are others not covered by the language. These Design Rules were checked manually because we did not develop yet a verifier.

4.1. Health Watcher system

The first system where we use the Design Rule language was the Health Watcher (HW) system, a real web-based information system implemented in AspectJ [Kiczales et al. 2001]. This system was selected because it was used in many previous works [Soares et al. 2002, Greenwood et al. 2007, Neto et al. 2007] and its design has a significant number of non-crosscutting and crosscutting concerns.

We specified some Design Rules of this system using the language. In the generated specifications, the rule *xcall* was utilized fairly enough, for example, allowing calls to the synchronization mechanism only within its responsible aspect.

Another language mechanism fairly utilized in the Design Rules specification is the parametrization. In the HW the parametrization was used, for example, to define facade methods where transactional management would be necessary. These methods were specified within the configuration module belonging to the defined Design Rule. Considering the utilization of the same transactional management mechanism, this Design Rule could be easily reused in other systems, changing only the configuration module.

The utilization of the language to define Design Rules seemed fairly simple. The parametrization mechanism created Design Rules that can be reused in other applications. However, we believe that some improvements have to be done in the parametrization mechanism to enhance the language expressivity.

4.2. BestLap - Arena

We have also applied the proposed language to specify Design Rules related to an optional feature called Arena in a J2ME mobile game called BestLap. This feature consists of posting the player score to a server after a game execution. This feature imposes a series of restrictions on the base code, like calling methods from certain points and providing a specific set of members (attributes and methods).

The Arena feature is characterized by a strong heterogeneous crosscutting, involving components like menu exhibition and event handling, network communication, additional screens and data. Our language seems to be useful to establish which methods must be present in certain core classes and to enforce some method calls within specific points. By making this clear in a Design Rule, it is possible to evolve core classes and aspects because the dependencies between them are visible to developers.

Although the language has been useful to express the constraints imposed by the optional feature Arena, we believe that more constructs are necessary to deal with alternative features, that may impose different constraints. Another point we did not explore was the feature interaction.

5. Related Work

AspectJ *declare error* and *declare warning* constructions [Kiczales et al. 1997] are useful to prohibit some join points, but can not be used to force the existence of a specific join point. Besides, there are design rules with structural requirements that again can not be

checked with these constructions. As an example, We can check if an specific method is called or executed using *declare error/warning* and consider this an error, but we can not check if the same method is defined in a class (structural constraint) because defining a method is not a join point in AspectJ. In addition, it is not possible using *declare error/warning* to force a call to a method from an specific point because its semantics is to prohibit a join point and not require it. In contrast, the proposed language aims to guide developers during initial development phases.

Sullivan *et al.* [Sullivan et al. 2005] 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. In this work the design rules are specified in natural language, leading to a long and ambiguous interpretation in an automatic checking.

Griswold *et al.* [Griswold et al. 2006] showed how to express part of the design rules into a set of Crosscutting Programming Interfaces (XPIs) that are useful to document and check part of the design rules (contracts). These XPIs were specified using AspectJ. Although it was possible to check part of the design rules, the use of a language not designed to this propose leads to very complex specifications (contracts imposed by aspects). The language we propose in this paper eliminates the ambiguity that may be introduced by natural language and easies the task of specifying design rules. Additionally, it enables the parallel development of components, supporting the clear specification of responsibilities that each component developer must attend.

Chavez *et al* [Chavez et al. 2006] presented crosscutting interfaces as a conceptual tool for dealing with complexity of heterogeneous aspects at the design level. This work also presents a modeling notation for description of architectural-level aspects that also supports the explicit representation of crosscutting interfaces. However, despite of using a visual notation is important for documentation purpose, it does not enable to check if the code was built according to established interfaces. Our proposal, besides enabling the specification a major number of design rules uncovered by the visual notation, allows to verify automatically if code is in conformity with the specified design rules.

Morgan *et al* [Morgan et al. 2007] presents a language, inspired by the pointcut language in AspectJ to declaratively encode design rules to OO systems. However, this language is useful only in OO systems and does not permit a clear definition of rules for parallel development of components.

Open Modules [Aldrich 2005] permits defining an interface composed by a set of pointcuts that can be advised by clients, introducing a form of encapsulating join points occurring inside a module and protecting them from external advising. Although join point hiding is an important concern, it does not provide information to the aspect developer beyond exported join points. Our approach aims to support class and aspect developers since initial development phases through the establishment of some design rules that serve as interface between them. It also provides a mechanism that enforces a structure to both classes and aspects. This structure can be useful to write pointcuts, advices and inter-type declarations that refer only to classes methods present on the design rule.

Kellens *et al* [Kellens et al. 2006] propose the concept of model-based pointcuts that capture join points based on conceptual properties instead of structural properties of

the base program entities, achieving a low coupling of the pointcut definition with the source code. However, changes to base code may impose modifications in the conceptual model, although generally protecting aspects from changes. Our approach, though based on structural properties, supports parametrization that is useful to accommodate syntactic changes. Additionally, both classes and aspects agree on a common interface that exposes requirements to both, specially during early development phases.

6. Concluding Remarks and Future Work

We presented in this work a language for Design Rules specification in AOP systems that defines the minimum requirements to be followed by developers of classes and aspects. Beyond the modularity enhancement, the utilization of a specific language for this purpose generates a unambiguous and simpler specification.

Another benefit presented is the possibility of defining and using parametrization mechanisms. This characteristic increases the possibility of specified Design Rules reuse decreasing consequently the involved costs. When using this language in real systems we verified the importance of the parametrization in several examples.

The language was evaluated specifying the Design Rules of a Web based system called Health Watcher and an optional feature called Arena in a game called BestLap. This work presented that some improvements still have to be done to improve the language parametrization mechanism and new constructions able to deal with alternative features can be also necessary.

The next steps are clearly define the language semantics, utilize the language to describe Design Rules in other case studies and develop a compiler that automatically checks if the rules are followed. In addition, we are evaluating new constructions that tackle more specific problems in AOP product lines and new parametrization mechanisms that may improve the language expressivity.

7. Acknowledgments

We would like to thank CNPq and CAPES, Brazilian research funding agencies, for partially supporting this work. In addition, we thank SPG[1] members for feedback and fruitful discussions about this paper.

References

Aldrich, J. (2005). Open Modules: Modular Reasoning about Advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming*.

Baldwin, C. Y. and Clark, K. B. (2000). *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press.

Chavez, C., Garcia, A., Kulesza, U., SantÁnna, C., and de Lucena, C. J. P. (2006). Crosscutting interfaces for aspect-oriented modeling. *Journal of the Brazilian Computer Society*, 12(1):43–58.

Greenwood, P., Bartolomei, T., Figueiredo, E., Dósea, M., Garcia, A., Cacho, N., SantÁnna, C., Soares, S., Borba, P., Kulesza, U., and Rashid, A. (2007). On the Impact

[1]http://www.cin.ufpe.br/spg

of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proceedings of the 21st European Conference on Object–Oriented Programming, ECOOP'07*. to appear.

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

Kellens, A., Mens, K., Brichau, J., and Gybels, K. (2006). Managing the evolution of aspect-oriented software with model-based pointcuts. In *European Conference on Object-Oriented Programming (ECOOP)*, number 4067 in LNCS, pages 501–525.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65.

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

Lopes, C. V. and Bajracharya, S. K. (2006). Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 1–35. Springer.

Morgan, C., Volder, K. D., and Wohlstadter, E. (2007). A static aspect language for checking design rules. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 63–72, New York, NY, USA. ACM Press.

Neto, A. C., de Medeiros Ribeiro, M., Dósea, M., Bonifácio, R., Borba, P., and Soares, S. (2007). Semantic Dependencies and Modularity of Aspect-Oriented Software. In *1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07), in conjunction with the 29th International Conference on Software Engineering (ICSE'07)*. to appear.

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

Ribeiro, M., Dósea, M., Bonifácio, R., Neto, A. C., Borba, P., and Soares, S. (2007). Analyzing Class and Crosscutting Modularity Structure Matrixes. In *Proceedings of the 21th Brazilian Symposium on Software Engineering (SBES 2007)*.

Soares, S., Laureano, E., and Borba, P. (2002). 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.

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. (2005). 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. ACM Press.