The Journal of Systems and Software xxx (2013) xxx–xxx

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A design rule language for aspect-oriented programming

Alberto Costa Neto[a,b,*,1], Rodrigo Bonifácio[b,c,1], Márcio Ribeiro[b,d,1], Carlos Eduardo Pontual[b,1], Paulo Borba[b,1], Fernando Castor[b,1]

[a] Computing Department, Federal University of Sergipe, Av. Marechal Rondon, s/n, Jardim Rosa Elze, 49.100-000 São Cristóvão, Brazil
[b] Informatics Center, Federal University of Pernambuco, Av. Prof. Moraes Rego, 1235, Cidade Universitária, 50.732-970 Recife, Brazil
[c] Computer Science Department, University of Brasilia, Campus Universitário Darcy Ribeiro, Asa Norte, 70910-900 Brasilia, Brazil
[d] Computing Institute, Federal University of Alagoas, Av. Lourival Melo Mota, s/n, Cidade Universitária, 57072-900, Maceió, Brazil

ARTICLE INFO

Article history:
Received 2 March 2010
Received in revised form 30 March 2013
Accepted 30 March 2013
Available online xxx

Keywords:
Aspect-oriented programming
Design rules
Modularity

ABSTRACT

Aspect-oriented programming is known as a technique for modularizing crosscutting concerns. However, constructs aimed to support crosscutting modularity might actually break class modularity. As a consequence, class developers face changeability, parallel development and comprehensibility problems, because they must be aware of aspects whenever they develop or maintain a class. At the same time, aspects are vulnerable to changes in classes, since there is no contract specifying the points of interaction amongst these elements. These problems can be mitigated by using adequate design rules between classes and aspects. We present a design rule specification language and explore its benefits since the initial phases of the development process, specially with the aim of supporting modular development of classes and aspects. We discuss how our language improves crosscutting modularity without breaking class modularity. We evaluate it using a real case study and compare it with other approaches.

1. Introduction

Aspect-oriented programming (AOP) (Kiczales et al., 1997) has been proposed as a technique for modularizing crosscutting concerns. Logging, distribution, tracing, security, and transactional management have been presented as examples of crosscutting concerns well addressed by AOP. Nevertheless, AOP might lead to modular reasoning problems. For instance, in order to reason about classes, developers may have to consider the implementation of every aspect (Sullivan et al., 2005; Clifton and Leavens, 2002; Steimann, 2006). In fact, by referring to classes implementation details in aspects, one might inhibit modular reasoning and compromise changeability, requiring class modifications to be fully aware of the aspects affecting the class. For example, the developer might need to check all aspects in order to confirm that a method extraction will not lead to missing join points. Therefore, constructs aimed to support *crosscutting modularity* might actually break *class modularity* (Ribeiro et al., 2007), creating mutual dependencies between classes and aspects.

Sullivan et al. (2010) observed that some of these modularity problems can be mitigated by using adequate design rules (Section 2) between classes and aspects. The rules reduce such dependencies in AO systems, and are not just guidelines and recommendations: they generalize the notion of information hiding and interfaces, and must be rigorously obeyed. Sullivan et al. (2005) show how an AO program can benefit (in terms of modularity) from the adoption of design rules since the early stages of development, when compared to an approach in which class and aspect developers do not define any rule before implementation (*oblivious approach*). They noticed that a system developed based on pre-established design rules was less complex and presented less dependencies between classes and aspects.

One problem of the design rule approach described by Sullivan et al. is that design rules are specified informally, in natural language. This may lead to sometimes verbose, incomplete, inconsistent, and ambiguous specifications. Also, natural language cannot be checked automatically due to its expressiveness. In a subsequent work, Griswold et al. (2006) introduce the idea of expressing design rules using AspectJ through the definition of Crosscutting Programming Interfaces (XPI). Although it is possible to check part of the design rules using XPIs, the use of a language not designed for this purpose leads frequently to complex specifications because the contracts are mechanically checked using aspects. However, to the best of our knowledge, none of the previous approaches (Sullivan et al., 2005, 2010; Griswold et al., 2006; Lopes and Bajracharya, 2006) proposes a language with the specific purpose of describing design rules in AO systems.

* Corresponding author at: Computing Department, Federal University of Sergipe, Av. Marechal Rondon, s/n, Jardim Rosa Elze, 49.100-000 São Cristóvão, Brazil. Tel.: +55 79 21056678; fax: +55 79 21056678.
E-mail addresses: rbalmeida@cic.unb.br (R. Bonifácio), marcio@ic.ufal.br (M. Ribeiro), ceplc@cin.ufpe.br (C.E. Pontual), phmb@cin.ufpe.br (P. Borba), castor@cin.ufpe.br (F. Castor).
[1] Tel.: +55 81 21268430; fax: +55 81 21268438.

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.03.104

Therefore, the main contribution of this paper is a design rule specification language (Section 4) that can be used to improve the modularity of AO systems. This language leads to a more modular design, decoupling classes and aspects through the establishment of minimum requirements for parallel development. Also it supports the description of design rules in a declarative, simple and unambiguous manner, enabling automated checking of rules against code. For instance, the language allows designers to write the (un)expected join points and to define responsibilities of both class and aspect developers. These requirements are automatically checked by a tool (Section 5) that points out design rule violations. Additionally, it is possible to map specifications in AspectualAcme (Batista et al., 2006a), an Architectural Description Language, to corresponding design rules in our language as proposed by Dósea (2008) (Section 7.7), enabling the use of design rules since the initial development phases.

The language was evaluated (Section 6) with the Health Watcher system (Soares et al., 2002) (Section 3). We specified design rules related to crosscutting concerns from the aforementioned system using both our language and Crosscutting Programming Interfaces (XPIs) (Griswold et al., 2006). A comparison between these approaches is presented in Section 6.

2. Software modularity and design rules

Although presented in the early 1970s, the criterion for modular design proposed by Parnas (1972) is still used as a guide for architects and has been applied in other areas. His notion of modularity is closely related to design decisions that decompose and organize the system into a set of modules. Moreover, the following benefits are expected in a modular design:

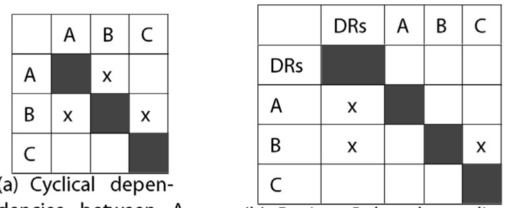
Comprehensibility: a modular design allows developers to understand a module by looking only at: (1) the implementation of the module itself; and (2) the interfaces of the other modules referenced by it.

Changeability: a modular design enables local changes. If changes are necessary in the internal implementation of a module *A*, the other modules that depend exclusively on *A's interface* will not change, since there is no modification in the module interface.

Parallel development: after the specification of the module interfaces, a modular design enables the parallel development of modules. Different teams can focus on their own modules development, reducing time-to-market and communication needs.

Besides that, Parnas proposed the *information hiding* principle as the criterion to be used when decomposing systems into modules. According to Parnas, the parts of a system that are more likely to change must be hidden into modules with stable interfaces.

Reinforcing these ideas, Baldwin and Clark (1999) proposed a theory which demonstrates that modularity is a key factor to support innovation (development of new artifacts that integrate and communicate with existent ones) and market growth, independently of industry domains. Their theory uses Design Structure Matrixes (DSMs) (Steward, 1981) to reason about dependencies among artifacts and argues that the task structure of an organization is closely related to such dependencies. As a consequence, if two artifacts without a public and stable interface are coupled, they cannot be developed in parallel, which actually requires either (a) more communication between different teams; or (b) their implementation to be assigned to a single team.



	A	B	C
A	■	x	
B	x	■	x
C			■

(a) Cyclical dependencies between A and B.

	DRs	A	B	C
DRs	■			
A	x	■		
B	x		■	x
C				■

(b) Design Rules decoupling components A and B.

Fig. 1. Example of dependencies in a DSM.

Sullivan et al. (2005, 2010) and Lopes and Bajracharya (2006) have previously used DSMs as tool for studying properties of software design. Following these works, we use Design Structure Matrixes (DSMs) as a tool for visualizing dependencies among design parameters. These parameters correspond to decisions that need to be made along product design. Design parameters might have different abstraction levels. For instance, in the software industry, some design decisions are related to development process, programming language choice, code or architectural style (Ribeiro et al., 2007). Moreover, here we also consider implementation as a design activity. Therefore, classes, interfaces, packages, and aspects are also represented as design parameters, as we need to make several decisions about them. Our language is focused on this level.

For example, suppose that the DSM depicted in Fig. 1(a) represents software components as parameters. The mark in row B, column A indicates that design decisions regarding component B depend on decisions concerned to component A. In a similar way, an "x" in row A, column B indicates that design decisions of component A depend on decisions of component B. Whenever this mutual dependency occurs, we have a cyclical dependency, which in fact implies that both components cannot be independently addressed. As a consequence, their parallel development is compromised.

In the same DSM of Fig. 1(a), component B depends on C (expressed by an "x" in row B, column C) but C does not depend on any other component (there is no mark in row C). Therefore, C can be independently developed but B cannot be completely developed until design decisions of C have been established. We can improve modularity by removing dependencies between design decisions, but several assumptions must be made before that. Such assumptions, represented as a special kind of parameter, are named, by Baldwin and Clark (1999), design rules.

Therefore, design rules are parameters that are less likely to change and are used as interfaces between modules (Lopes and Bajracharya, 2006). In this way, they are used to decouple design parameters, like typical programming interfaces remove the coupling between software components. Such design rules establish strict partitions of knowledge and effort at the outset of a design process. They are not just guidelines or recommendations: they must be rigorously obeyed in all phases of design and production (Baldwin and Clark, 1999). Fig. 1(b) illustrates components A and B being decoupled by design rules (DRs). Since A and B do not depend on each other anymore, it is possible, for example, to change A's implementation as long as it respects the design rule. In addition, when both respect the design rules, parallel development becomes possible (Neto et al., 2007). However, notice that the coupling does not disappear, but it is better managed. Its place has been changed instead: A is coupled to the design rule. The same happens to B.

3. Modularity issues in aspect-oriented programming

Aspect-oriented programming was proposed to modularize crosscutting concerns (Kiczales et al., 1997). The most well-known aspect-oriented language is a Java extension called AspectJ

(Kiczales et al., 2001). However, constructions supported by AspectJ-like languages can produce high coupling between classes and aspects, which may compromise the criteria discussed previously. In this section, we illustrate this problem with some examples, extracted from the Health Watcher system (Soares et al., 2002).

Health Watcher (HW) is a real web-based information system originally implemented in Java and later restructured with AspectJ (Kiczales et al., 2001). The system was developed to improve the quality of the services provided by health care institutions, allowing citizens to register complaints regarding health issues, so that health care institutions can investigate and take required actions. The HW system has been used as the case study of several *aspect-oriented* works (Greenwood et al., 2007a,b; Soares et al., 2002). We have selected the HW system because its design has a significant number of crosscutting concerns. Its design also involves a number of design decisions related to GUI, persistence and concurrency concerns. Fig. 2 shows the core architecture of the HW system. This architecture aims to modularize the user interface, distribution, business rules, and data management concerns. Next we describe the major architectural components of the HW system:

View layer: related to the HW web interface. The implementation of this layer is based on the Front Controller (Alur et al., 2003) and Command (Gamma et al., 1995) patterns, using *servlets* and *plain Java objects*. The communication with the business layer is implemented by means of calls to the interface `IFBusiness`, which may be distributed or not.

Business layer: responsible for implementing both the business logic and transactional concerns. The class `HWFacade`, which implements `IFBusiness`, is the unique point of interaction with this layer and is based on the Facade pattern (Gamma et al., 1995). 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 (Alur et al., 2003). 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 *domain objects*. These objects represent the core concepts of the application, transit between all architectural layers, and have some business logic. *Complaint*, *employee*, and *health unit* are examples of core concepts in the HW system.

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

By analyzing this system, we present some issues related to *aspect-oriented* modularity. First, we discuss the *fragile pointcut* problem in Section 3.1. After that, we argue that clear interfaces, for decoupling crosscutting code related to different concerns, are necessary for developing different concerns in parallel and supporting software maintenance activities (Section 3.2).

3.1. Fragile pointcut problem

The first issue mentioned above can be observed with the implementation of the transaction management concern, which affects the behavior of certain methods of the HW business layer. The implementation of this concern consists of adding transactional behavior, using `begin` and `commit` or `rollback` commands. Listing 1 illustrates a piece of the source code related to this concern in AspectJ.

Listing 1. Aspect responsible for implementing the transactional concern.

```
public aspect TransactionAspect {

    public pointcut transactionalMethods() : /* ... */

    before() : transactionalMethods() {
        transMechanism.begin();
    }
    after() returning : transactionalMethods() {
        transMechanism.commit();
    }
    after() throwing : transactionalMethods() {
        transMechanism.rollback();
    }

}
```

If a class developer that is oblivious to the `TransactionAspect` needs to implement a new functionality that requires the introduction of a new transactional method (`newMethod`), at least two problems might occur:

(1) *An existing aspect misses join points that supposedly demand aspect intervention*. This occurs if `newMethod` should be transactional but transactional management is not implemented within it and its join points are not matched by the `transactionalMethods` pointcut, the method will not work as expected. This situation is relatively common because class developers might be oblivious to the pointcut expression defined by aspect developers; in an OO scenario this would not occur because developers must be aware of the transaction mechanism interfaces and their usage pattern to write a new transactional method.
(2) *Certain join points are unintentionally affected by aspects*. As an example, if the class developer manually implements the transactional management concern in the new method (calling `transactionMechanism.begin`), and this method is coincidentally matched by the `transactionalMethods` pointcut, an unexpected behavior (*e.g.*, deadlock) occurs, since two consecutive calls to *begin transaction* are made.

In both cases, we need to change the pointcut in order to capture (or not) the join points. That is why this problem is called *fragile pointcut* (Störzer and Koppen, 2004). The situation above exposes some modularity problems: (1) *comprehensibility is compromised*, because two modules should be studied in order to understand and correctly implement the concern; and (2) *parallel development is problematic*, because it is impossible to implement classes and aspects independently and be sure that one does not interfere with

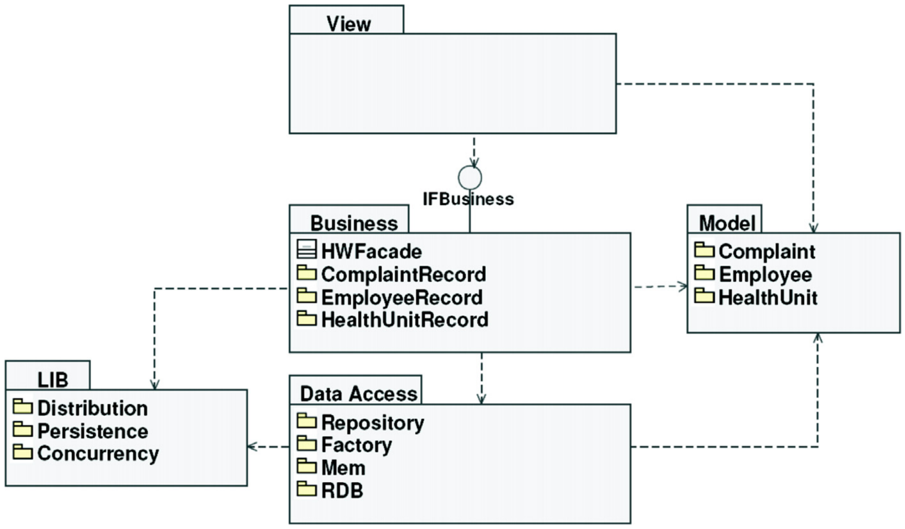


Fig. 2. Overview of the Health Watcher architecture.

the other. Also, developers can implement unintended behavior into a module which, although under their responsibility, may break other parts of the system. Moreover developers might need to know details about other modules in order to implement the modules under their responsibility correctly. This is true even for modules that are not explicitly used by the concerns implemented as aspects.

So any unanticipated change in the class might cause problems and the application may not behave as presumed. Note that we have a cyclical dependency in this situation: the aspect depends on the class syntactically (notice that pointcut expressions refer to classes, interfaces, aspects and their members); and to change the class, the developer must be aware of the aspect. Fig. 3(a) illustrates such a cyclical dependency through a DSM, whereas Fig. 3(b) shows design rules coming into play to remove the dependency between aspect and class.

In summary, this problem indicates that we need interfaces (or design rules) that delimit the scopes from which certain methods may be called (in this case `begin`, `commit`, and `rollback` within `TransactionAspect`) and expose the expected method signatures to be matched by the `transactionalMethods` pointcut. Through the establishment of these design rules at system design, acting as design contracts, both class and aspect developers can work more independently because the points of interaction between them are defined a priori (Sullivan et al., 2005). If they respect the constraints, many defects, delays and problems can be avoided.

3.2. Unsupported parallel development

Another issue may arise when a team is assigned to develop a crosscutting concern and another team is assigned to develop a non-crosscutting concern. This happens, for instance, with the transaction management concern from Listing 1 because aspect developers need to know the transactional points and the business layer class developer needs to be conscious about the aspect. Without a clear interface between those concerns, substantial communication might be required, which, in fact, compromises parallel development of both concerns and causes the fragile pointcut problem during system maintenance. This is aligned with the observations of Parnas (1972) and Baldwin and Clark (1999) that modularity is related to the assignment of development activities, which may be reflected in the program structure (Devanbu et al., 2003), and that a modular design reduces the communication paths among design decisions, in such a way that units of work can be developed in parallel.

For instance, suppose that a team is responsible for developing the use case related to the *complaint management* concern (a core concern of the Health Watcher system). And another team is responsible for an *auditing concern* that must be triggered whenever a change in a complaint occurs. Without a clear interface stating which are the relevant complaint changes (the set of join points) and how these join points should be exposed (*e.g.* through specific naming conventions) by the complaint management team, any modification in the use case must be communicated to the auditing team. Consequently, it is difficult to implement the auditing concern at the same time that the complaint management concern is being developed. Although these concerns can be encapsulated in single code units, providing better locality, their design is not modular because each one is dependent on implementation details of the other.

To better explore this, consider the DSM depicted in Fig. 4, which represents some design parameters and respective dependencies of the HW system. Based on this DSM, we can realize that

		1	2
TransactionAspect	1	■	x
Business Classes	2	x	■

(a) Cyclical Dependency.

		1	2	3
Transaction Management DR	1	■		
TransactionAspect	2	x	■	
Business Classes	3	x		■

(b) Cyclical Dependency removed.

Fig. 3. Transaction concern without and with design rules.

		1	2	3	4	5	6
Goals and constraints	1	■					
Complaint requirements	2	x	■				
Auditing requirements	3	x		■			
Architectural decisions	4	x	x		■		
Complaint implementation	5		x		x	■	x
Auditing AO implementation	6			x		x	■

Fig. 4. Cyclical dependency between complaint and auditing.

		1	2	3	4	5	6	7
Goals and constraints	1	■						
Complaint requirements	2	x	■					
Auditing requirements	3	x		■				
Architectural decisions	4	x	x		■			
Auditing Design Rule	5		x	x		■		
Complaint implementation	6		x		x	x	■	
Auditing AO implementation	7			x		x		■

Fig. 5. Auditing design rule removes cyclical dependency.

decisions about the complaint implementation (row 5) depend on decisions about complaint requirements (dependency row 5, column 2), architectural decisions[2] (dependency row 5, column 4), and the auditing concern (dependency row 5, column 6). This last dependency exists because the team responsible for developing the *complaint concern* has to know which join points must be exposed to the *auditing concern*. Moreover, as we can observe in Fig. 4, the auditing implementation also depends on the complaint implementation decisions, since changes in its implementation should be notified to the auditing implementation team. In this way, there is a cyclical dependency between complaint implementation and auditing implementation – a clear example of non-modular design.

Based on the information hiding principle, we should encapsulate the dependencies between complaint and auditing concerns in a special kind of interface (a design rule). Applying design rules to this example, we improve the design structure by removing the cyclical dependencies between the complaint and auditing concerns. The new DSM is presented in Fig. 5. Notice that a new parameter (actually a design rule) was introduced (row 5) and all dependencies are below the main diagonal (there are no more cyclical dependencies). This design rule was proposed in order to improve the parallel development between class developers and aspect developers. In fact, it is responsible for establishing contracts that define what is expected from both teams. In this case, among other things, this design rule establishes that class developers must follow specific naming conventions for methods that represent auditing points. Based on that, aspect developers can implement the auditing concern.

Our approach addresses these problems through the use of a language to express and enforce design rules to both classes, interfaces and aspects. This language is presented in the next section, followed by an evaluation in Section 6.

4. A language for specifying AO design rules

We propose the *Language for Specifying Design Rules* (*LSD*) as a tool for clearly and unambiguously expressing design rules with the aim of decoupling classes and aspects through the establishment of the minimum requirements for developing and maintaining them independently. It provides an automated method to tackle mainly two design issues:

- Prevent changes in existing types, such that the types are not unintentionally affected by a given aspect.
- Instruct types that want to become affected by a given aspect, to ensure that they do it and that they do it in the right way.

[2] Examples of architectural decisions for the Health Watcher system are the selected architectural style (layers), patterns, and technologies for each layer or concern (presentation, distribution, persistence, and so on).

In the following subsections we present LSD main constructs.

4.1. LSD overview

The concept of interface supported by LSD, which we call *design rule*, goes beyond the one typically supported by OO languages, involving more than public methods and constants. It can include, for example, required join points, private members, inter-type declarations and expected inheritance relationships. A DR contains a set of rules that must be followed by classes, interfaces and aspects that implement the DR. LSD is focused on specifying the *minimum requisites* about the structure and behavior of the modules that will be independently developed. This is similar to how the list of methods in a Java interface works. Implementing classes must define the methods that appear in the interface, but are also allowed to define other methods. A DR can be seen as a predicate over the modules of a system that must be true for the system to be considered valid. So, system modules can exhibit any structure or behavior their designers see fit, unless something is explicitly disallowed by the DR specification. LSD supports two kinds of rules: *structural* and *behavioral*.

Structural rules are *roles* that actual classes, interfaces, and aspects will perform when the system is implemented. For each role, developers must implement an actual type (or a set of types) that satisfies the role constraints. A design rule does not refer directly to actual type implementations, but defines their expected structure and behavior by means of the structural rule elements. Each structural rule in a design rule specifies elements that are visible from other structural rules in the same design rule, supporting design and implementation time analysis of constraints.

Behavioral rules provide a mechanism for specifying constraints about the behavior of classes and aspects. Examples of behavioral rules are required method calls (`call/xcall`), and attribute accesses (`get/xget`) and changes (`set/xset`). These behavioral rules are contained by structural rules with corresponding behavior requirements.

The general structure of a design rule is presented in Fig. 6. In accordance with this meta-model, a *design rule* is composed by one or more *structural rules* (Section 4.2), which are used to define structural constraints for interfaces, classes and aspects. In other words, a structural rule can define ordinary type members, like constants and method signatures (Interface Members), attributes, constructors, and methods (Class Members), pointcuts, advice, inter-type declarations (attributes, methods, and constructors), and declare declarations (Aspect Members). Additionally, *structural rules* support member expressions (Section 4.6.3), quantification expressions (Section 4.6.4) and behavioral rules (Section 4.3) within their bodies. *Behavioral rules* can be defined both within the body of a structural rule and within constructors, methods, inter-type declarations, and advice bodies.

A design rule instance (DRI) is responsible for binding classes, interfaces, and aspects to corresponding structural rules in a DR. Through this mapping, both developers and our checking tool can infer the constraints associated with each component. So, each type associated to a SR must be in conformance with the SR constraints.

We developed a tool (COLA) that checks design rules described in LSD against their implementing types during normal compilation process. Whenever a design rule violation is found, our tool reports which rule was violated and inhibits bytecode generation, forcing the developer to solve the problem. But if the code is in accordance with the design rule, the compilation process continues normally and the bytecode is generated. It is important to note that no bytecode is necessary to represent a design rules after compilation because the checking process is completely performed at compilation time. We describe this tool in more detail in Section 5.

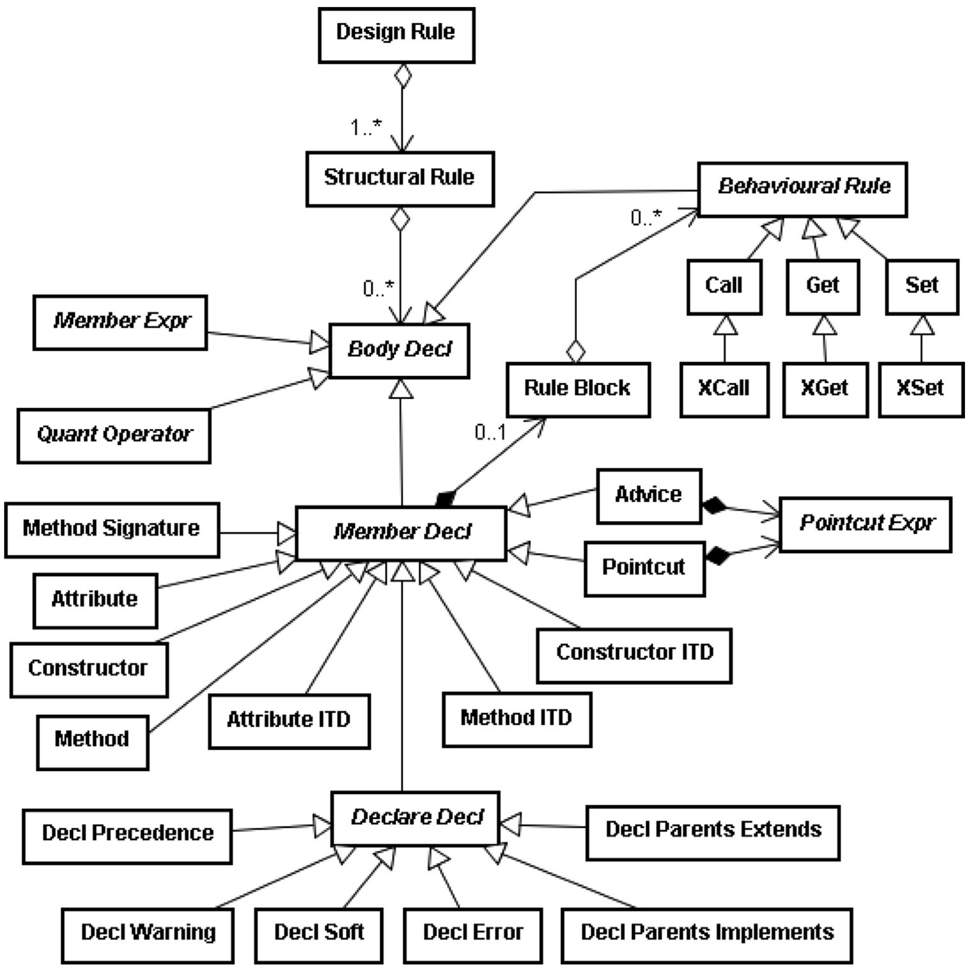


Fig. 6. LSD meta-model.

Listing 2 shows part of the `RepositoryDR` rule. The latter is useful for enforcing design constraints on types of the data access layer of the Health Watcher system (Soares et al., 2002). `RepositoryDR` declares the minimum requirements to support the parallel development of `Component`, `Record`, and `Repository` types.

Listing 2. Structural Rule for Repository.

```
1  dr RepositoryDR [Component, Record, Repository] {
2     ...
3     class Record {
4        public new(*);
5        public void insert(Component)
6          throws ObjectAlreadyInsertedException, ObjectNotValidException,
7                 TransactionException;
8        public Component search( (int || String) )
9          throws ObjectNotFoundException, TransactionException;
10       ...
11    }
12    ...
13 }
```

Line 1 of Listing 2 contains, after the design rule name, a list of three *Design Rule Parameters*, namely `Component`, `Record`, and `Repository`. The objective of declaring those parameters is to abstract the real type names, giving support to both design rule reuse and decoupling between types. In this listing we will focus only on part of the specification of the second DR parameter (`Record`) (Lines 3–11). DRs contain more details than a Java interface because DRs are designed to support the parallel development of classes and aspects. For instance, the `public` constructor declaration, the alternative parameter type of the `search` method, and the long list of exceptions, are required in this case, but make the DR look verbose. But depending on the system, they might no be required.

4.2. Structural rules

Structural rules (SRs) are design rules that describe constraints over members of interfaces, classes, and aspects. They are similar to Java interfaces but they support additional constraints (beyond required public methods and constants), like protected attributes that must be declared and required method calls. Moreover, there are specific constraints about aspect structure, like required inter-type, pointcut, and advice declarations. SRs must be declared exclusively within DRs, in other words, they cannot be declared in isolation. Some degree of reuse can be achieved through DR inheritance (Section 4.6.7). It enables the reuse of DRs and its SRs to build specialized versions (*e.g.* with more constraints), similarly to the OO inheritance mechanism.

The definition of the Structural Rule `Record` (Line 3) does not impose constraints like visibility, inheritance, and interface implementation over types that implement it because no specification related to that is defined. For example, if we replace `class Record` by `public class Record`, the required type visibility becomes `public`. On the other hand, the body of Structural Rule `Record` contains three member declarations:

- The first one (Line 4) contains a *constructor declaration*, which must be `public` and have exactly one parameter, no matter the type. It is easy to identify a constructor declaration within a DR because it is always named `new` and has no return type.
- Following the constructor (Lines 5–7), there is an *instance method declaration*. The method must be called `insert`, have `public` visibility, a `void` return type, and a parameter of type `Component`. Moreover, its `throws` clause must contain exactly three exceptions: `ObjectAlreadyInsertedException`, `ObjectNotValidException` and `TransactionException`. Section 4.6.5 provides more details about exceptions.
- Another *instance method* is declared within `Record` (Lines 8–9). The method must be called `search`, have `public` visibility, return a `Component`, and contain in its `throws` clause exactly the exceptions `ObjectNotFoundException` and `TransactionException`. Method `search` must have only one parameter, either an `int` or a `String`. This does not mean that the method must accept both types of parameters. It means that if the class provides either a `search(int)` method or a `search(String)` method, it satisfies the design rule constraints.

4.3. Behavioral rules

Behavioral rules (BR) provide a mechanism for specifying constraints about the behavior of classes and aspects. They are useful, for instance, because aspect developers usually trust on some method call (or attribute access/change) at pointcut specification time. However, if the developer changes the class, removing the join point shadow (the method call), the pointcut has to be adapted. Otherwise, the aspect will behave differently from what is expected. If some behavioral rule is defined to check the method call occurrence and it is not found, the class developer is warned, avoiding the missing join point problem. But if the class developer really decided to change the class, the design rule must be changed to accommodate the conflicting behavior. This decision is clearly documented by the revised design rule and is available to other developers.

Table 1 shows the *behavioral rules* provided by LSD. The scope of these rules includes classes, aspects, constructors, methods and advice. A behavioral rule defined within the scope of a class or aspect has to be respected in that scope. For example, if the `call` is specified within a method of a SR, the corresponding class or aspect is in conformity with the design rule if it makes a call to the method in that scope. If the `call` rule is defined within the scope of a structural rule, it means that the required method call must occur at least once in the corresponding class (or aspect) by any of its members (*e.g.*, methods, constructors and advice). Another example is to use `get` or `set` in a SR, requiring a read (or write) of the specified attribute within the required scope, Every rule initiated by “x” guarantees that the rule will be followed only in the defined scope and it will not be possible in any other location among the types included in the corresponding design rule instance. It is important to notice that types not included in the design rule instance are not restricted by “x” rules. For example, the `xcall` rule guarantees that a specific method will be called exclusively within the scope in which it was defined. Calls outside of that scope are not allowed, except from types that are not bound, by the design rule instance, to any structural rule of the design rule. Our scope notion was inspired by the XPIs scope (Griswold et al., 2006), used to delimit the parts of the program that are subject to design rule checking.

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 (!). When a *behavioral rule* is defined within the scope of a class or aspect, it is valid for the entire class. One `call` within a class indicates that a method call to a specific method must exist in some method or constructor of that class. In fact, all possible statements are considered, like the statements from static and field initializers. However, this construct does not guarantee that a call will in fact occur at runtime. For example, if a required method call depends on a condition based on user input, the call will be executed only if the condition is satisfied.

Listing 3. Behavioral Rules for Transaction Management.

```
dr TransactionManagementDR
    [ITransactionMechanism, TransactionManagement, Facade] {

     interface ITransactionMechanism {
         void begin() throws TransactionException;
         void commit() throws TransactionException;
         void rollback() throws TransactionException;
     }

     aspect TransactionManagement {

        pointcut transactionalPoints(): call(* Facade.*(..));

        before(): transactionalPoints() {
             xcall(void ITransactionMechanism.begin());
        }
        after() returning: transactionalPoints() {
             xcall(void ITransactionMechanism.commit());
        }
        after() throwing: transactionalPoints() {
             xcall(void ITransactionMechanism.rollback());
        }
    }

    class Facade {}
}
```

Table 1
Behavioral rules provided by LSD.

Rule	Description
`call(method)`	The enclosing scope must have a *call to* `method()`
`xcall(method)`	The enclosing scope is the only place in the design rule instance that can *call* `method()`
`get(attribute)`	The enclosing scope must have an *access to* `attribute`
`xget(attribute)`	The enclosing scope is the only place in the design rule instance that can *access* `attribute`
`set(attribute)`	the enclosing scope must have an *assignment to state* `attribute`
`xset(attribute)`	The enclosing scope is the only place in the design rule instance that can perform *assignments to* `attribute`

Listing 3 shows three rules illustrating the use of the `xcall` behavioral rule (Lines 15, 18 and 21). These three rules indicate that methods `begin`, `commit` and `rollback` from `ITransactionMechanism` must be called exclusively within the scopes of the respective advice, prohibiting calls from any other place among the types bound to structural rules of `TransactionManagementDR` by the design rule instance. This is useful to guarantee that no other type can call these methods, leading to transaction management errors. For example, assuming that `TransactionManagementDR` is declared and instantiated, if the method `begin` is called from any method of `TransactionManagement`, an error is reported, because calls to it can only occur within the before advice associated with the `transactionalPoints` pointcut.

Additionally, the DR declares a `Facade` structural rule which is referred to by the `transactionalPoints` pointcut and has no associated constraints. For simplicity, we are assuming that all `Facade` methods are transactional. Considering the previous example, we observe that through simple rules it is possible to improve system modularity, documenting the rules of interaction between types and checking if these rules are respected.

It is important to notice that a class that does not implement and is not bound to (by a design rule instance) any *structural rule* defined by the design rule `TransactionManagementDR` can call the method `begin`. This happens because the design rule checking scope is restricted to the set of types that are bound to the DR. This binding is defined by the design rule instance specification (Section 4.5). However, since it is possible to expand the set to include any number of classes, interfaces and aspects, we can adapt the scope in accordance with developers needs.

4.4. Explicit implementation of design rules

The structural and behavioral rules are used to create design rules, that is, contracts. We need also to specify the parts involved in those contracts. This can be done by explicitly implementing design rules. So, trying to keep similarity with the concept of Java interfaces, and following the LSD principle of establishing an interface between classes and aspects, LSD requires that both classes and aspects explicitly implement the DRs. When a class implements a DR, all constraints contained in the DR are explicit to the developer. This means that class developers are partially oblivious to aspect implementation. Only the necessary details are exposed to the developer.

Examples of components that implement `TransactionManagementDR` are shown in Listing 4. In the listing, interface `IPersistenceMechanism` implements the DR, playing the role of `ITransactionMechanism`. `HWTransactionManagement`, in its turn, implements the same design rule, but as `TransactionManagement`, whereas the `HWFacade` class plays the role of `Facade`.

For each implemented design rule, we instantiate one or more DR parameters. Based on the chosen parameters, it is possible to obtain the set of constraints imposed by the design rule. For instance, if a class developer needs to make a change, it is easier to identify which design rules must be respected, facilitating evolution tasks and preventing errors. Besides, its use enables automatic verification and separate compilation because, based on the type and its list of implemented DRs, it is possible to infer dependencies on other types based on the relationships present in the design rule (*e.g.*, a class that depends on a method inter-type declaration) and compile the type without the elements on which it depends. The use of `implements` requires a class or aspect to follow all the rules defined by the corresponding *structural rule*. It is important to note that in order to implement this feature, we changed the syntax and semantics of Java `implements` clause to support design rules, besides ordinary Java interfaces. In spite of that, during compilation our tool removes design rule names from all implements clauses, avoiding changes to normal Java compilation process. Therefore, the mapping from types to implemented design rules does not causes side effects to Java ordinary bytecode but is kept by our tool to be used during the design rule checking process.

Listing 4. Explicit implementation of the Design Rule TransactionManagementDR.

```
public interface IPersistenceMechanism
   implements TransactionManagementDR(ITransactionMechanism) {
... }

public aspect HWTransactionManagement
   implements TransactionManagementDR(TransactionManagement) {
... }

public class HWFacade
   implements TransactionManagementDR(Facade) {
... }
```

An alternative to explicit implementation of design rules in classes, aspects, and interfaces would be to provide a *separated mapping* between these types and the list of design rules implemented by them. Also, it would be necessary to inform the corresponding structural rules implemented by each type (binding to DR parameters). We did not choose this alternative because it does not make it clear to the developer which design rules are implemented by a certain type unless all existing mappings are inspected. With an explicit indication (our approach), it is easier to discover the implemented design rules in cases where dependencies could prohibit separate understanding and compilation.

Another alternative that is similar to explicitly implementing the design rules is using *annotations* to provide the list of implemented design rules. Compared to the chosen approach (explicit implementation) it is very similar because in both of them the information of which design rules are implemented is contained in classes, interfaces and aspects, but with annotations the frontend could become simpler. The downside of this approach is requiring the distribution of a default *Annotation Type* that is referred to by classes, interfaces and aspects.

4.5. Defining a design rule instance

Instead of explicitly implementing a DR, when a class does not depend on declarations or behavior specified by aspects, we omit the `implements` clause in the class declaration and indicate that the class takes part in the contract only when instantiating the DR. A design rule instance (DRI) is responsible for binding classes, interfaces, and aspects to corresponding structural rules in a DR. Through this mapping, both developers and our checking tool can infer the constraints associated with each component. So, each type associated to a SR must be in conformance with the SR constraints. For example, the design rule of Listing 3 has three parameters (Line 2): `ITransactionMechanism`, `TransactionManagement` and `Facade`. Listing 5 shows the instantiation of `TransactionManagementDR` by assigning a name to the instance `TraMngDRI`, associating interface `IPersistenceMechanism` to `ITransactionMechanism`, aspect `HWTransactionManagement` to `TransactionManagement` and `HWFacade` to `Facade`.

Listing 5. Design Rule instance.

```
dri TraMngDRI = TransactionManagementDR (
  ITransactionMechanism = IPersistenceMechanism;
  TransactionManagement = HWTransactionManagement;
  Facade = HWFacade
);
```

When a structural rule does not have dependencies that demand an `implements` in the types, it is possible to bind any type, even types that do not explicitly implement the DR. On the other hand, when their `implements` clause includes the DR, it is checked if those bindings match, which means that each structural rule associated with the type by the design rule instance must be in its `implements` clause. We choose this approach because it enables us to use existing types to instantiate DRs whenever the associated structural rules are free of inter-type declarations dependencies.

Listing 6. DR instantiation grammar.

```
DRInstance  ::= 'dri' Id '=' Id '(' ParamIdList ')' ';'
ParamIdList ::= ParamId [';' ParamId]*
ParamId     ::= Id '=' IdSet
IdSet       ::= IdSetExpr [',' IdSetExpr]*
IdSetExpr   ::= TypeNamePat | TypeGroup | '!' IdSetExpr
TypeGroup   ::= 'class' | 'interface' | 'aspect'
TypeNamePat ::= Equals to AspectJ type name pattern
```

Besides allowing the binding of more than one type to one DR parameter, one can also use wildcards "*", like in AspectJ, instead of concrete type names. Besides, DR instances support the negation of type(s) using the "!" operator. For instance, we can provide *all classes from a certain package except one* (*or more*) as a parameter at DR instantiation. In addition to that, it is possible to use "class", "interface", and "aspect" to explicitly select classes, interfaces, and aspects. We detail the DR instantiation grammar in Listing 6.

Our approach is flexible enough to support any number of instances of the same design rule with different parameters in a single system. Also, the same type can be bound to different structural rules in different design rule instances. This approach can be useful in Software Product Lines (SPL) implemented with AOP, where design rule instantiation can be used as a configuration mechanism to support any number of product instances. Each product instance typically uses different classes to implement product-specific variations, but all these classes must adhere to a common set of constraints. We can express these constraints through design rules and bind the structural rules to each product specific classes.

4.6. Additional constraints

In this section we present more language constructs that provide ways of expressing additional constraints and discuss some situations where they are useful.

4.6.1. Modifiers

One type of constraint is related to component modifiers. It might be necessary to state that a certain class has public and not package visibility. Also, it could be necessary to prohibit developers from declaring a specific method as synchronized. These constraints usually represent dependencies between aspects and classes. For instance, if an aspect implements a concurrency mechanism but the class developer (oblivious to the aspect) declares a method as synchronized in an affected class, the aspect implementation can misbehave. With LSD, developers can state these constraints and avoid hidden implementation dependencies like that.

LSD is focused on specifying the minimum requisites about type structure and behavior. This means that declarations contained in a DR are satisfied if at least the listed modifiers are included in the types implementing the DR. So, unless they are explicitly prohibited by the DR specification, other modifiers are accepted.

Listing 7 shows part of the design rule for repositories (`RepositoryDR`) presented in Listing 2. Observing the structural rule `Record`, it is possible to note that no constraints about class visibility are defined in the DR. So, `Record` can be implemented by a class with any visibility – `public`, `protected`, `private` and package (when none is specified). In contrast to Java, package visibility is not assumed when no visibility modifier is declared. In LSD, package visibility is represented by the `pack` keyword. This is also valid for other class modifiers (*e.g.*, `final` and `abstract`) Contrasting, the `insert()` method must be `public`. Moreover, the implemented `insert()` method can also be `synchronized`, `final`, and `static`, for example. If we want to prohibit the method from being declared as `synchronized`, we just use `!synchronized` in its modifier list after the `public` modifier.

Table 2
Examples of parameter expressions.

Parameter expression	Description
m()	No parameters are accepted
m(*)	One parameter of any type
m(..)	Zero or more parameters of any type
m(*,..)	One or more parameters of any type
m(T, *,..)	One parameter of type T, followed by one or more parameters of any type
m(.., T)	One or more parameters and the last one is of type T
m(!T)	One parameter of any type other than T
m(T1 || T2)	One parameter of type T1 or T2
m(T* && *2)	One parameter of type with name that starts with T and ends with 2
m(T* ^^ *2)	One parameter whose type name either starts with a 'T' or ends with a '2', but not both

Table 3
Quantification operators semantics.

Parameter expression	Description
all	All selected members (N = scope size)
exists	At least one ($N > 0$)
one	Exactly one ($N = 1$)
none	Zero ($N = 0$)
opt	At most one ($N = 0$ or $N = 1$)
range[min-max]	Range from min to max (min $\ll N \ll$ max)

N is the number of members from the Scope that respect the rule.

Listing 7. Structural Rule for Repository.

```
dr RepositoryDR [Component, Record, Repository] {
   ...
   class Record {
      ...
      public void insert(Component)
        throws ObjectAlreadyInsertedException, ObjectNotValidException,
               TransactionException;
      ...
   }
   ...
}
```

Additionally, it is possible to define private attributes and methods and associate constraints with them. It might sound odd to define a constraint about a private member of a class, but sometimes aspects need to refer to private members. If this dependency is defined in a design rule, it is possible to explicitly express it.

4.6.2. Method parameters

Method parameters are important to method specifications in a DR because, in conjunction with a method name, they uniquely identify one method within a type. That is why some pointcut designators (*e.g.*, `call`, `execution` and `withincode`) in AspectJ support expressions involving parameters types and method names. Since these pointcut designators depend on method parameters, LSD supports the definition of rules over parameters, aiming to avoid join point misses caused by changes to method parameters.

LSD supports the definition of rules that expose to developers name patterns for each parameter type of methods and constructors.

Table 2 shows examples of supported method (and constructors) parameter expressions using the "*" and ".." wildcards available in AspectJ pointcut designators. Also, it contains examples of the LSD support to expressions for parameter type names using the ! (NOT), || (OR), && (AND), and ^^ (XOR) operators.

4.6.3. Member expressions

LSD also allows the use of expressions involving structural rule members. This feature enables one, for example, to include in a structural rule a constraint that requires a method `public Component search(int)` or a method `public Component search(String)`, using the disjunction operator `||`. The class implementing this structural rule will be correct if it provides at least one of them.

Requiring two or more members to exist at the same time (using the conjunction operator `&&`) is also possible and is actually the default semantics for members that appear in a structural rule.

4.6.4. Quantification

Java interfaces implicitly specify *existential quantification* of public instance methods and constants. In previous sections we presented some LSD features that support more types of constraints than Java interfaces can express. In addition, LSD provides *universal quantification*. It supports expressing rules based on several members of a structural rule.

For example, suppose that we need to express that all public methods from a certain class cannot be synchronized because the concurrency management is implemented by an aspect. This rule can be expressed by selecting the desired scope (*all public methods*) and applying some rule to it (`!synchronized`). In LSD, we can express this constraint by including the quantification expression in a structural rule, as shown in Listing 8.

Listing 8. Structural Rule for Concurrence Management with Quantification.

```
dr ConcurrencyManagementDR [SyncType, SyncAsp] {
   class SyncType {
      all (public * *(..))
      then (!synchronized * *(..));
   }
   aspect SyncAsp {
      pointcut syncPoints() : execution(public * SyncType.*(..));
   }
}
```

Observing the quantification expression in Listing 8, we can notice two method signature patterns: one immediately after the `all` keyword and another one after the `then` keyword. We refer to the first one as the *Quantification Scope* and to the second one as the *Quantification Rule*, following the general form:

Operator(*Scope*)*then*(*Rule*);

where

Operator: specifies the amount of elements (*e.g.*, one, none, or all) from the scope that must respect the Rule. Table 3 contains the list of the quantification operators supported by our language.
Scope: establishes the scope over which the rule is checked, selecting the members, according to the member expression, among methods, constructors, inter-type declarations (methods and attributes), pointcuts, and advice (with or without body).
Rule: expresses the rule that must be true for the selected members. Any valid member expression can be used here.

With quantification expressions, we can write more elaborate design rules with a clear semantics. LSD supports quantification expressions with behavioral rules and also set operators "+" (Union) and "−" (Intersection). For example, We can use these features to write the rule "No method except `getInstance()` can call the singleton class constructor" (shown in Listing 9), which is useful to write a design rule to check if a class respects the constraints of the Singleton design pattern.

Listing 9. Example of Quantification Expression (Singleton).

```
dr Singleton [S] {
   class S {
      none ( * *(..) − * getInstance() )
      then ( * *(..) { call(*.new(..)); } );
      ...
   }
}
```

Listing 10. Checking Design Quality with Quantification Expressions.

```
dr QualityDR [C] {
   class C {
      range[1..10] (* *(..)) then (public * *(..));
      none (* *) then (public * *);
   }
}
```

Although LSD was designed aiming to support the parallel development of classes and aspects, with quantification operators it also supports the specification of some design rules related to program design. We can enforce, for instance, that "*The number of public methods in a class must range between 1 and 10*" and that "*No public attributes are accepted*" through the design rule `QualityDR` from Listing 10.

It is important to notice that these quantification expressions are limited to structural rule members (*e.g.*, attributes, methods and constructors). We do not allow them to express global rules to Classes/Interfaces/Aspects. However, it is possible to express quantification through DR instance bindings (shown in Section 4.5). The binding, which may use quantification over type names, is responsibility of a design rule instance.

4.6.5. Implements/extends/throws clauses

Another type of constraint requires classes implementing a certain *structural rule* to implement or extend a specific type. We show an example of this constraint in Listing 11. Classes that implement the `Record` structural rule must explicitly implement the `BaseRecord` interface. The structural rule `Repository` extends class `AbstractRepository`. Both `AbstractRepository` and `BaseRecord` are existing interface and class, but could be structural rules if they were defined in the design rule `PersistenceDR`. This would improve flexibility by enabling the mapping of these design rule elements to different existing interfaces and classes.

These constraints are useful for writing generic pointcuts based on the implemented interfaces and superclasses. The design rule prevents class developers from changing `implements`/`extends` clauses and removing required join point shadows.

Listing 11. Implements/Extends constraints.

```
dr PersistenceDR [Record, Repository] {
   class Record implements BaseRecord {
      ...
   }
   class Repository extends AbstractRepository {
      ...
   }
   ...
}
```

Both `extends` and `implements` clauses in a SR represent the *minimum requirements* for the type that implements it. In other words, the type is free to implement or extend other types beyond the ones defined in the SR. However, the latter case only applies to interfaces because classes and aspects do not support multiple inheritance.

Although this default semantics has been sufficient in most of the cases, we found some specific constraints that demand more expressivity. A simple example is trying to enforce that a class cannot implement any interface. With the *includes* semantics this is impossible. Also, we are unable to express that a class must implement a certain number of interfaces. In summary, in some cases we need a semantics that refers to one specific element but in other situations, we need to express constraints over the complete list of elements.

Since the includes semantics covers most of the common cases, we adopted it as the default semantics, maintaining the similarity with the Java/AspectJ syntax. However, aiming to cover the other cases, we enhanced the `extends` and `implements` clauses of structural rules, to explicitly choose the appropriate semantics to express the design rule. We introduced three possibilities of explicit semantics, namely: `includes`, `excludes` and `exactly`. They are used after the `implements`/`extends` keyword and can be used in conjunction, except for `exactly`.

This feature is also present in the `throws` clause from methods and constructors, but with a different default semantics when compared to the `implements` and `extends` clauses. The latter two implicitly assume the `includes` semantics as default, contrasting to the `throws` clause, which assumes the `exactly` semantics. This difference is due to the fact that method and constructor callers must be prepared to catch or rethrow all the exceptions present in the `throws` clause of the called method. Hence, following the minimum requirement principle in this case would mean that method implementations could define exceptions in their `throws` clauses that might not be present in the corresponding SR. This could effectively break the clients of the method. As a consequence, from the developer's point of view, the `throws` clause of a method or constructor signature in a SR has an equivalent semantics to a common method or constructor signature. However, when an aspect developer writes a pointcut based on the `throws` clause, the `includes` semantics is more appropriate because the pointcut matches the method call if the method declares at least the exception referred to by the pointcut in its `throws` clause. LSD allows developers to choose the semantics they need using the `includes`, `excludes` and `exactly` keywords.

4.6.6. *Inter-type declarations*

Inter-type declarations enable aspects to introduce attributes and methods in classes. These declarations are very useful, but they may also create dependencies between classes and aspects. Among other applications, this mechanism is frequently used in software product lines implemented with aspects to introduce variable method implementations or attribute values in classes (Alves, 2007). Also, the template method design pattern uses inter-type declarations to introduce the template method. Design rules can be used to explicitly create a contract between classes and aspects introducing product line variabilities, as illustrated in the example of Listing 12. It shows a design rule, `ScreenAtributtes`, that declares two structural rules: `MainScreen` and `SizeVariability`. The last one requires two inter-type declarations of fields `WIDTH` and `HEIGHT` in the class that implements the structural rule `MainScreen`. `ScreenAtributtes` also requires from the `SizeVariability` aspect an inter-type declaration of a method called `paint()` in the `MainScreen` class. As a consequence, the class developer can be sure that these two attributes and the method will exist (introduced by some aspect) and can use them within the class.

Listing 12. Inter-type declarations in SR.

```
dr ScreenAttributes [MainScreen, SizeVariability] {
    class MainScreen {
        ...
    }
    aspect SizeVariability {
        public static final int MainScreen.WIDTH;
        public static final int MainScreen.HEIGHT;
        public void MainScreen.paint(Graphics);
    }
}
```

The checking of inter-type declarations from a DR is relatively simple when their target is a type (not a SR). In this case, LSD searches an inter-type declaration with the same target, name and parameters (for methods). If some matching declaration is found, it also checks if the other elements match, namely modifiers, type (in the attribute case), return type, `throws` clause, and body. In summary, the inter-type declarations checking is similar to the common attribute and method declarations.

In the same way that occurs with pointcut declarations in a DR, inter-type declarations have different behaviors when they use SR names as targets of inter-type declarations. Basically we transform the inter-type declaration in the aspect that implements the DR, exchanging the reference to the target by the SR name, obtained from the design rule instance. Moreover, when the SR is associated with more than one type, we *clone* the inter-type declaration, creating one instance for each type name associated with the SR, and exchange the target name by one of the type names. After that, we have one inter-type declaration for each different target. For instance, if `MainScreen` is bound to a class called `MainCanvas` and `SizeVariability` to an aspect named `DefaultScreen`, first we check if `DefaultScreen` provides the three inter-type declarations, as requires `ScreenAttributes`. Then, we replace the SR name `MainScreen` by the type name `MainCanvas` in the inter-type declarations from aspect `DefaultScreen`.

On the other hand, if `MainScreen` is bound to more than one class, copies of the inter-type declaration are created in each type to which `MainScreen` is bound. In the case of a method inter-type declaration, the body is also duplicated.

A consequence of our approach is that it avoids both the explicit use of an artificial interface and the declaration of one declare parents for each target type, as it is usually done when we need an inter-type declaration with multiple targets. With LSD, this is hidden from developers, since they only need to provide the list of types associated to a SR in the design rule instance.

4.6.7. *Design rule inheritance*

With design rule inheritance, we can compose constraints. For example, observing the DR `TransactionManagementDR` from Listing 3 we can note that the DR depends on the adoption of the Facade design pattern (Gamma et al., 1995). Although the Facade is a feasible solution, we might choose a different mechanism to establish the set of methods that should have Transaction Management, like a naming convention to Record creation and update methods. So we might, for example, declare `TransactionManagementDR`, which focuses on specifying the expected behavior – (calls to transactional methods `begin`, `commit` and `rollback`) – at transactional points without explicitly stating what are these points. These transactional points are matched by the `transactionalPoints` pointcut and lead to the activation of certain advice. Listing 13 shows a general DR for Transaction Management, while Listing 14 shows how to extend it to use an approach that depends on a Facade class.

The approach that depends on name convention is shown in Listing 15. Through DR inheritance, we can identify reusable parts of the design and delay some decisions, or even provide more than one design option.

Listing 13. General Transaction Management DR.

```
dr TransactionManagementDR [ITransactionMechanism, TransactionManagement] {
    interface ITransactionMechanism {
       void begin() throws TransactionException;
       void commit() throws TransactionException;
       void rollback() throws TransactionException;
    }

    aspect TransactionManagement {
      pointcut transactionalPoints();
      before(): transactionalPoints() {
          xcall(void ITransactionMechanism.begin());
      }
      after() returning: transactionalPoints() {
          xcall(void ITransactionMechanism.commit());
      }
      after() throwing: transactionalPoints() {
          xcall(void ITransactionMechanism.rollback());
      }
    }
}
```

Listing 14. Transaction Management based on a facade.

```
dr TranMngFacadeDR
   [ITransactionMechanism, TransactionManagement, Facade]
   extends TransactionManagementDR[ITransactionMechanism,
                                   TransactionManagement] {
   aspect TransactionManagement {
      pointcut transactionalPoints(): call(* Facade.*(..));
   }
   class Facade {}
}
```

Listing 15. Transaction Management based on naming conventions.

```
dr TranMngRecordDR
   [Component, Record, ITransactionMechanism, TransactionManagement]
   extends TransactionManagementDR[ITransactionMechanism,
                                   TransactionManagement] {
   class Component {}
   class Record {
      public void insert(Component);
      public void update(Component);
   }
   aspect TransactionManagement {
      pointcut transactionalPoints():
         execution(public void Record.insert(Component)) ||
         execution(public void Record.update(Component)));
   }
}
```

5. A Compiler for LSD and AspectJ

Another contribution of our work is the development of a tool to automatically check if the design rules specified using LSD are being followed by developers. We decided to extend an AspectJ compiler and try to consider design rules as similar as possible to Java/AspectJ interfaces.

Since extending the *ajc* (*AspectJ compiler*) (Eclipse.org, 2013b) is a time-consuming task, because it was not designed for that, we decided to use the extensible *AspectBench Compiler* (*abc*) (M. University, 2013). This compiler has been used to successfully implement some extensions to AspectJ (Hoffman and Eugster, 2007; Harbulot and Gurd, 2006; Chen and Chien, 2006; Aotani and Masuhara, 2007; Akai et al., 2009). In this section, we present our tool for checking design rules.

The AspectBench Compiler (abc) (Avgustinov et al., 2005) is an extensible AspectJ compiler intended as a workbench for aspect-oriented language research, and it has been successfully adopted as the basis for the implementation of a number of extensions. The system is divided into a *frontend*, taking care of parsing and static semantic analysis, and a *backend*, performing optimization and aspect weaving. The abc frontend is itself implemented using *JastAdd* (Avgustinov et al., 2008). JastAdd is a meta-compilation system for generating language-based tools such as compilers, source code analyzers, and language-sensitive editing support. It is based on a combination of attribute grammars and object-orientation. The key feature of JastAdd is that it allows properties of abstract syntax tree nodes to be programmed declaratively. These properties, called attributes, can be simple values like integers, composite values like sets, and reference values which point to other nodes in the abstract syntax tree (AST). The reference values allow graph properties to be defined. For example, linking identifier uses to their declaration nodes, or representing call graphs and dataflow graphs. AST nodes are objects, and the resulting data structure, including attributes, becomes an object-oriented model, rather than only a simple syntax tree (Hedin, 2009).

5.1. Extending abc to support LSD

In this section we present more details about the implementation of the *Compiler for LSD and AspectJ* (*COLA*), which is an extension to abc aiming to support the new constructs introduced by LSD. LSD has an implementation using the new abc frontend (Avgustinov et al., 2008) based on JastAdd (Ekman and Hedin, 2007).

We have implemented a scanner (lexical analyzer) and parser (syntactic analyzer) to LSD source code files using same tools and implementation steps adopted by other abc extensions. We also made a few small changes to the original abc parser. After that, we created the LSD checker, which uses both LSD and abc (AspectJ) AST nodes to, respectively, obtain the set of constraints imposed by the design rules and check if they are satisfied by the AspectJ program. Then we integrated the LSD checker with abc. During the compilation process, abc activates the LSD Parser concurrently to the AspectJ parser. When both ASTs are built, abc calls the LSD checker to analyze these ASTs, recording the *warnings* and *errors* found. This process was supported by JastAdd attributes that were gradually incorporated into both LSD and AspectJ AST nodes and avoids the creation of visitors to navigate through the AST. As a result, COLA produces a list of *errors* and *warnings* pointing out the design rule violations to the developer. If no error is found, the LSD checker executes the required changes to the AspectJ AST (*e.g.*, introducing DR inherited pointcuts) before the abc backend starts the bytecode generation process. In summary, we extended the abc frontend, adding a separate AST for LSD and keeping untouched the abc backend.

5.2. Using the LSD Compiler (COLA)

As explained before, COLA was implemented as an extension to *abc*, so we decided to maintain its invocation as similar as possible to the original *abc*. As a matter of fact, if no design rule is defined, the compiler works exactly as before.

In order to differentiate between AspectJ and LSD source code, we established that design rules and design rule instances must be defined in files with the extensions ".dr" and ".dri", respectively. Whenever file names with these extensions are provided as parameters to *COLA*, the design rule checker is executed during the compilation process, warning and pointing out errors found in the source code.

For example, when we execute the command:

$$abc \quad -ext \quad abc.lsd * .java * .aj * .dr * .dri$$

All AspectJ (".java" and ".aj") and LSD (".dr" and ".dri") source files are processed by COLA, executing the design rule checker and generating common Java bytecode files (".class"). After compilation, the generated bytecode files can be executed by any standard Java Virtual Machine without any extra library. It is important to notice that the current version of COLA does not check class files, requiring the corresponding source code files from all types referred by a design rule instance.

For instance, consider a class `C` and an aspect `A` with a pointcut (`callToM2`) that depends on a method call to `m2` within `m1`. This dependency can be expressed in LSD by the design rule `DREx` and its instance `DRIEx` (Listing 16). The resulting class `C` and aspect `A` are shown in Listing 17.

Listing 16. DR and DRI used by COLA.

```
dr DREx [C,A] {
  class C {
    void m1() {
      call(* C.m2());
    }
    void m2();
  }
  public aspect A {
    pointcut callToM2() : call(* C.m2()) && withincode(* C.m1());
  }
}


dri DRIEx = DREx(C = C;
                 A = A);
```

Listing 17. Aspect dependent on a method call.

```
public class C {
   public void m1() {
      m2();
   }
   public void m2() {...}
}

public aspect A {
   after() : callToM2(){...}
}
```

If the class developer extracts the call to method `m2` from the body of method `m1` to the body of `m3`, violating the design rule `DREx`, COLA reports an error at compilation time (Listing 18), giving information to the class developer about the constraints that were violated.

Listing 18. Error Reported by COLA.

```
[Error in class C] Method declaration with required behavior not
found:

void m1() { call(* C.m2()); }

(Check structural rule C within design rule DREx)

Found 1 error(s)!
```

Table 4
Comparison between LSD and XPI (transaction management).

	Expressiveness					Conciseness				
	C1	C2	C3	C4	C5	C1	C2	C3	C4	C5
XPI	AC	N	AC	N	AC	39	–	16	–	25
Extended XPI	AC	N	AC	P	AC	39	–	16	96	25
DR	AC		AC		AC		25	4		75

Where the acronyms mean automatically checked (AC), partially checked (P) and no checking (N of natural language representation).

6. Evaluation

This section contains an evaluation of LSD. We discuss advantages and disadvantages of using our approach when compared to Crosscutting Programming Interfaces (XPIs) (Sullivan et al., 2005, 2010; Griswold et al., 2006). Such interfaces specify the exposed join points by classes and aspects. Although just an abstract XPI representation was provided in Sullivan et al. (2005), Griswold et al. explain how to implement some crosscutting interfaces as syntactic constructs of AspectJ (Griswold et al., 2006; Sullivan et al., 2010). Basically, design rules are documented using *abstract pointcut descriptions* and constraints applied to classes and aspects. These constraints might be written as `declare warning` (or `declare error`) constructs or just as comments in the source code (when a constraint cannot be expressed as a `declare warning` or `declare error`).

The advantages of using a design rule based approach over an oblivious approach are discussed by Sullivan et al. (2005). They observed that aspects in design rules approach are much simpler because specific naming conventions are followed and interesting abstract states are implicitly exposed by modules. Without design rules, the aspect has to compute complex pointcuts by going through lots of details of base code. Moreover, the design rule pointcut captures newly-coded changes, because it is quantified and base code designers are constrained to write new code following the design rule. Since these advantages were previously discussed, we focus on the direct comparison between LSD and the XPI approach.

Our evaluation compares three specifications of the transaction concern from the Health Watcher system (Soares et al., 2002): the `TransactionManagementDR` shown in Listing 3 and two XPIs introduced next. We also compare the design rule specification of the distribution concern in LSD with another one that uses the XPI approach (Section 6.2). Our goal is to discover to what extent LSD satisfies the objective of being more expressive (supporting the automatic checking of design rules) and concise when compared to XPIs. That is why the following criteria are considered (Felleisen, 1990):

- *Language expressiveness*: quantifies the degree to which a language is able to express a constraint. In fact, it is a three level factor – language supports, does not support, or partially supports a specific rule.
- *Language conciseness*: measures how simple it is to express a constraint in a language. Here, we use the minimum number of tokens required to express a constraint as a measurement of conciseness.

6.1. Transaction concern

In order to support the parallel development between classes and the aspects responsible for implementing the transaction concern, we can specify the following constraints:

(C1) There must exist an interface (`ITransactionMechanism`) that defines methods for starting (begin), committing (commit), and rolling back transactions (rollback);
(C2) These methods may throw an exception `TransactionException` in case of error;
(C3) Also, in order to enable the definition of the pointcuts, there should be a facade named HWFacade;
(C4) The transaction aspect `HWTransactionAspect` must call the transactional methods of `ITransactionMechanism`. Moreover, these calls have to occur at specific events, detailed in what follows:
- A transaction must be started before any facade method[3] (the aspect should call `ITransactionMechanism.begin`);
- After the return of any facade method, the current transaction should be committed (the aspect calls `ITransactionMechanism.commit`);
- If any exception is raised by facade methods, the current transaction should be rolled back (`ITransactionMechanism.rollback` should be called by the aspect); and

(C5) These calls must only occur within the aspect `HWTransactionAspect`.

Listing 19 shows a first XPI specification for the transaction management concern. It is supposed to guarantee constraints C1, C3, and C5. Lines 4–7 specify C1, stating that the `ITransactionMechanism` interface should exist and have `begin`, `commit` and `rollback` methods. Although we expect `ITransactionMechanism` to be an interface, we cannot enforce such a restriction using XPIs (unless in the form of comments). The *ajc compiler* (Eclipse.org, 2013b) assumes that `ITransactionMechanism` is an interface, a class or even an aspect, reporting an error only if this type does not exist. No error is reported if that interface exists but does not define any of the mentioned methods. As we can see on Lines 8–15, constraint C2 is described only in natural language, so it is not possible to check it in an automatic way.

The pointcut defined in Line 2 states that a class (or interface) named `HWFacade` must exist. Moreover, all methods of `HWFacade` should have a transactional context. Such a pointcut encompasses constraint C3.

The `staticMethodScope` pointcut defined in Line 17 states that the type `HWTransactionAspect` must exist. As aforementioned, we cannot guarantee that this type will be an aspect using XPIs. This pointcut is used on the `TransactionContractXPI` (Lines 30–35) to guarantee that any call to methods of the `ITransactionMechanism` interface occurs *only* within the `HWTransactionAspect`. Otherwise, an error is raised (constraint C5).

Finally, Lines 20–26 require the `HWTransactionAspect` aspect to call methods `begin`, `commit`, and `rollback` (`tm` represents an instance of one class that implements `ITransactionMechanism`). Moreover, these calls have to occur within specific advices (C4). Nevertheless, these constraints were specified using natural lan-

[3] Although capturing calls to all facade methods is a feasible solution, but not so frequently used in practice, we might choose a different mechanism to establish the set of methods that should have transaction management, like a naming convention to transactional methods or using an additional interface that explicitly exposes the set of transactional methods. This last alternative can be easily applied to the transaction management concern case.

guage, which means that no compilation error is reported if these expected calls do not occur within specific `HWTransactionAspect` advice.

Listing 19. Base version of the TransactionManagementXPI.

```
public abstract aspect TransactionManagementXPI {
 pointcut transactionalMethods(): execution(* HWFacade.*(..));

 pointcut callsToTransactionContext() :
   call(void ITransactionMechanism+.begin())  ||
   call(void ITransactionMechanism+.commit()) ||
   call(void ITransactionMechanism+.rollback());
 /*
  * Methods begin(), commit() and rollback() of ITransactionMechanism
  * can throw TransactionException in case of error. Their signatures are:
  *
  * void begin() throws TransactionException;
  * void commit() throws TransactionException;
  * void rollback() throws TransactionException;
  */

 public pointcut staticMethodScope(): within(HWTransactionAspect);

 /*
  * HWTransactionAspect must call the methods begin(), commit(),
  * and rollback() defined in the ITransactionMechanism interface.
  * These calls should occur within advice like the following ones:
  *
  * before() : transactionalMethods() {... tm.begin(); ...}
  * after returning() : transactionalMethods() {... tm.commit(); ...}
  * after throwing() : transactionalMethods() {... tm.rollback(); ...}
  */
}

public aspect TransactionContractXPI {
   declare error:
      TransactionManagementXPI.callsToTransactionContext() &&
      !TransactionManagementXPI.staticMethodScope()
          : "Illegal use of a transactional method";
}
```

We could improve this XPI to check part of constraint C4 at compile time. In order to do that, we should introduce Lines 3 – 19 of Listing 20 into the `TransactionManagementXPI`. As the *ajc compiler* is unable to raise errors if a given pointcut does not match *join points shadows*, we define three empty advice (Lines 15–19). In that way, a compilation warning is raised if any of the given advice are not applied, i.e., if methods `begin()`, `commit()` or `rollback()` of `ITransactionMechanism` are not called inside the type `HWTransactionAspect`. Note that `after()` or `around()` could have been used as advice, instead of `before()`. However, despite requiring calls to transactional methods within some advice of `HWTransactionAspect`, the (extended) version of `TransactionManagentXPI` does not enforce the exact advice where each call is supposed to occur. For example, although we expect a call to the `commit` method within an `after returning` advice, no error is reported if such a call happens within another advice, method or constructor. This problem arises because, since advice in AspectJ are anonymous, we cannot write pointcuts that explicitly refer to an advice because they do not have names.

Listing 20. Extended version of TransactionManagementXPI. This version specifies that HWTransactionAspect must call begin, commit and rollback methods.

```
 1 public abstract aspect XPITransaction {
 2  ...
 3  public pointcut expectedCallToBegin() :
 4    within(HWTransactionAspect) &&
 5    call(void ITransactionMechanism+.begin());
 6
 7  public pointcut expectedCallToCommit() :
 8    within(HWTransactionAspect) &&
 9    call(void ITransactionMechanism+.commit());
10
11  public pointcut expectedCallToRollback() :
12    within(HWTransactionAspect) &&
13    call(void ITransactionMechanism+.rollback());
14
15  before(): expectedCallToBegin() { }
16
17  before(): expectedCallToCommit() { }
18
19  before(): expectedCallToRollback() { }
20 }
```

The use of LSD can mitigate these problems. The `TransactionManagementDR` (Listing 3) restricts that `ITransactionMechanism` must be an interface and that `TransactionManagement` must be an aspect, otherwise a compilation error is raised. In addition, the `xcall()` construct of LSD assures a call occurs only within a specified scope among the DR components. With the use of this constructor, the same DR restricts that `begin()`, `commit()` and `rollback()` can only be called within before, after returning and after throwing advice, respectively. As a result, all presented constraints are satisfied by the LSD specification. But, as we show in the following sections, some constraints cannot be expressed by our language.

Table 4 summarizes our evaluation. Regarding expressiveness, both XPIs automatically check C1, C3, and C5. Both C2 and C4 cannot be checked using XPIs, since they are described in natural language expressed by commented code. The pointcuts introduced in Listing 20 allow the partial checking of constraint C4. However, by using these pointcuts, the *ajc* compiler only reports constraint violations if the `TransactionAspect` does not call any of the transactional methods (`begin()`, `commit()`, or `rollback()`). On the other hand, by using LSD we could specify and automatically check all the mentioned constraints.

Considering the conciseness criterion, we could specify C4 and C5 by means of the `xcall` behavioral rule in LSD, stating: (a) calls to transactional methods must occur within `TransactionAspect` and (b) these calls must occur at a specific advice. In fact, this leads to increased conciseness, since the LSD design rule requires 75 tokens (instead of 96 tokens in the extended XPI) to specify these constraints. In addition, our DR can check C4, which is not possible with XPIs. Moreover, constraints C1 and C2 can be easily described and automatically checked with the description of the `ITransactionMechanism` interface present in Listing 3.

6.2. Distribution concern

Another important Health Watcher concern implemented with aspects is *distribution*. In summary, class developers from the view layer use a facade to access the *business* layer as a local class. But distribution aspects intercept calls to this facade and redirect these calls to the corresponding methods from a remote facade, which, in its turn, invokes the respective remote methods. This process is transparent to local facade clients. We identify some design rules that must be established between the local facade and the distribution classes and aspects. These rules are expressed below through a set of constraints:

(C1) There must exist an interface that contains the set of local methods that must be intercepted by the distribution aspect. In the HW case, this interface is called `IFacade` and calls to `IFacade` methods constitute the join points.

(C2) Also, there must be an interface (`IRemoteFacade`) with the same set of methods defined by `IFacade`, but with the difference that each of them contains an additional Exception (`RemoteException`) in its `throws` clause.

(C3) There must exist a class that directly implements the local facade (`IFacade`). In the case of the HW, this class is `HealthWatcherFacade`.

(C4) The remote facade class (`RemoteFacade`) must provide a static method called `getInstance`, which returns an instance of the class.

(C5) The remote facade class (`RemoteFacade`) cannot have a `main(String[])` method;

(C6) An aspect executing in the client side captures all calls to the local facade (`IFacade`), through a pointcut (`facadeCalls`), and substitutes the original call by a remote call, delegating this task to the method `MethodExecutor.invoke`.

(C7) Some aspect must declare that business classes implement the `Serializable` interface, allowing their instances to be used as parameters of remote calls;

(C8) Also, an aspect must declare that the remote facade class implements *IRemoteFacade*, aiming to enforce that all remote methods are provided by that class.

(C9) A distribution aspect must introduce a `main` method in the class that implements the remote facade `IFacade`. In the case of the HW, this class is called `HealthWatcherFacade`.

(C10) The execution of the `main` method, introduced by the distribution aspect, must activate an `around` advice that: (1) initializes the remote facade instance; (2) binds the *remote object* to a name in the naming service.

We show in Listing 21 how these constraints can be expressed in LSD through the `DistributionDR` declaration. The first constraint (C1), expressed by the SR `ILocalFacade` requires that the local facade interface to declare at least one method. C2 is only partially met because, although we can enforce that all methods from the remote facade class (`IRemoteFacade`) must declare that they might throw `RemoteException`, we cannot compare this set with the set of methods defined in `ILocalFacade`. Presently, LSD does not support constraints that involve members from different SRs, but we plan to provide some mechanism to support that as future work.

Following that, we observe that C3 is completely satisfied by the SR `LocalFacade` which implements `ILocalFacade`. In the same way, `RemoteFacade` satisfies the constraint C4, requiring the declaration of a `public synchronized static getInstance` method which returns a `RemoteFacade` (Line 13). Following that, the constraint C5 is enforced by the SR `RemoteFacade` (Line 14), which forbids the declaration of any `main` method with a parameter of type array of `String` within the remote class. As required by the constraint C6, `ClientDistribution` represents an aspect in the client side that declares a `facadeCalls` pointcut which captures calls to all methods from the local facade, and also declares an around advice using that pointcut. The advice contains a `call` behavioral rule to the `MethodExecutor.invoke` method, which requires some call to this method within the advice scope.

The remaining constraints (C7–C10) are satisfied by `ServerDistribution`, a SR that represents the server-side aspect. It contains two `declare parents` (Lines 28 and 29) which satisfy, respectively, C7 and C8. It is important to observe that `Component` can be bound only to classes. `Component` can be associated to any set of classes through a design rule instance. Following, `ServerDistribution` declares an introduction of a `public static void main` method to RemoteFacade, satisfying the constraint C9. Finally, respecting the constraint C10, `ServerDistribution` declares a pointcut (`facadeMain`) that captures the execution of the introduced method (`main`), and activates an around advice that creates an instance of the remote facade and registers it in the naming service.

Listing 21. Distribution design rules with LSD.

```
dr DistributionDR [Component, ILocalFacade, LocalFacade,
                   IRemoteFacade, RemoteFacade,
                   ClientDistribution, ServerDistribution] {
  interface ILocalFacade {
    exists (* *(..)) then (* *(..));
  }
  interface IRemoteFacade {
    all (* *(..)) then (* *(..) throws includes(RemoteException));
  }

  class LocalFacade implements ILocalFacade {}
  class RemoteFacade {
    public synchronized static RemoteFacade getInstance();
    ![* main(String[]);]
  }

  class Component {}

  aspect ClientDistribution {
    pointcut facadeCalls() : call(* ILocalFacade.*(..));

    Object around() : facadeCalls()  {
      call(Object MethodExecutor.invoke(..));
    }
  }

  aspect ServerDistribution {
    declare parents: Component implements Serializable;
    declare parents: RemoteFacade implements IRemoteFacade;

    public static void RemoteFacade.main(String[]);

    protected Remote initFacadeInstance() {
      call(* RemoteFacade.getInstance());
    }

    protected pointcut facadeMain(String[] arr):
      execution(static void RemoteFacade.main(String[])) && args(arr);

    void around(String[] arr): facadeMain(arr) {
      xcall( * ServerDistribution.initFacadeInstance() );
      call( * UnicastRemoteObject.exportObject(*) );
      call( * Naming.rebind(..) );
    }
  }
}
```

Aiming to compare the design rule for the distribution concern specified in LSD with a corresponding XPI, we created two aspects that represent the distribution XPI, which we show in Listing 22. The first one is responsible for the client-side (`ClientDistributionXPI`) and the other for the server-side (`ServerDistributionXPI`) constraints. We summarize the comparison results in Table 5.

Listing 22. Distribution design rules with XPI.

```
abstract aspect ClientDistributionXPI<ILocalFacade> {
  pointcut facadeCalls() : call(* ILocalFacade.*(..));

  Object around() : facadeCalls()  {
    return doRemoteCall();
  }

  protected abstract Object doRemoteCall();
}

abstract aspect ServerDistributionXPI<Component, RemoteFacade, IRemoteFacade> {
  declare parents: Component implements Serializable;
  declare parents: RemoteFacade implements IRemoteFacade;

  protected pointcut facadeMain(String[] arr):
    execution(static void RemoteFacade.main(String[])) && args(arr);

  void around(String[] arr): facadeMain(arr) {
    createFacadeAndRegister();
  }

  protected abstract void createFacadeAndRegister();
}
```

Table 5
Comparison between LSD and XPI (distribution).

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
	Expressiveness									
XPI	AC	N	N	N	N	P	AC	AC	N	P
DR	AC	P	AC	AC	AC	AC	AC	AC	AC	AC
	Conciseness									
XPI	8	–	–	–	–	37	7	7	–	64
DR	21	26	6	23		41	11	7	12	92

Where the acronyms mean automatically checked (AC), partially checked (P) and no checking (N of natural language representation).

`ClientDistributionXPI` declares the pointcut `facadeCalls` that intercepts calls to all methods from the type `ILocalFacade`. However, we cannot express that the calls are supposed to be redirected to a remote object, using the MethodExecutor.invoke method. So, C6 is partially satisfied by the XPI. It is important to observe that we used *generics* to decouple the XPI from the real local facade type. Since the XPI requires a type parameter, we consider that the XPI also satisfies C1. This was necessary because we assume that the developer puts all local methods in the local facade, indicating to the aspect developer the set of methods that can have the local call exchanged by a remote call. Using this approach, the class developer can establish the subset of the facade methods that can be intercepted by the distribution aspect. This approach can also be applied to the transaction management concern implementation.

After that, `ServerDistributionXPI` declares two `declare parents`, the first one declares that business classes (Component parameter type) must implement `Serializable`, satisfying C7 and requiring the same number of tokens (7) because the declarations are equal. The second declares that the `RemoteFacade` parameter type implements `IRemoteFacade` parameter type, as required by C8. However, the XPI cannot express that the aspect developer must introduce the main method, as required by C9. Finally, the advice that intercepts the main method is defined. It calls an abstract method, that must be implemented by the aspect developer, and is responsible by the remote facade initialization and registration.

Listing 23. Defining concrete aspects for the Distribution XPI.

```
interface IFacade {}

interface IRemoteFacade {}

class HealthWatcherFacade {}

interface Component {}

aspect ClientDistribution extends ClientDistributionXPI<IFacade> {
   protected Object doRemoteCall() {
      /* Calls MethodExecutor.invoke to redirect the local call to
         a remote call */
   }
}

aspect ServerDistribution extends ServerDistributionXPI
          <Component, HealthWatcherFacade, IRemoteFacade> {
   protected void createFacadeAndRegister() {
      /* Creates the facade instance and registers it */
   }
}
```

But, there is no way to automatically enforce this requirement with XPIs. So, the constraint C10 is partially satisfied, and that is why it requires less tokens (64 against 92 of LSD) to be expressed.

Furthermore, design rules C2, C3, C4 and C5 cannot be checked through XPIs. Firstly, we cannot express that all methods from the remote facade must declare that they might throw `RemoteException` and check the one-to-one mapping from each local facade method to a remote facade method (both required by C2). Secondly, we cannot assert that the local facade is implemented by some class, as requires C3. Thirdly, XPIs cannot enforce that a certain class declares a *static* method, which is why it does not satisfy C4. If `getInstance` was an *instance* method, we could have used an interface to enforce its declaration. Finally, C5 is not satisfied because XPIs cannot forbid method declarations. However, we could forbid calls and executions of this method, but that would not work either. This occurs because the design rule C9 specifies that the method will be introduced by some aspect. This seems contradictory, but the idea is to stop class developers from implementing a method that will be introduced by some aspect. Listing 23 shows how the aspects that represent the distribution XPI are extended and integrated to the classes and interfaces from the Health Watcher system.

Although we used just two concerns in our evaluation, we consider them representative enough in the Health Watcher context because other crosscutting concerns were evaluated and similar results were found (Neto, 2010).

For example, the *Repository* design rule (Listings 2 and 7) cannot be automatically checked using XPIs, since the majority of the constraints would be represented as natural language. In its turn, the *Persistence* design rule, not discussed in this work, requires the same XPIs constructs as the transaction management one. The *Exception Handling* concern can also be specified through pointcut declarations, `declare soft` declarations and methods `throws` clauses, all supported by LSD.

6.3. Relevance and applicability

The benefits that stem from our approach enable the use of AOP in large and complex applications and specially in Software Product Lines (SPL). A SPL is a set of software-intensive systems sharing a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way (Clements and Northrop, 2001). SPL can be implemented with AOP languages like AspectJ (Alves, 2007; Alves and, 2004; Alves et al., 2005; Dantas and Borba, 2003; Dantas et al., 2003) but we can control dependencies between classes (*e.g.*, expected method calls, required method inter-type declarations) and aspects (implementing the variabilities) more clearly with design rules.

On the other hand, there are some basic aspect implementations that are so general that they make no explicit reference to classes or interfaces (no coupling). This is the case in aspect-oriented frameworks (Soares, 2004), which include reusable implementations of several concerns that can be specialized as required. Moreover, some aspects are inherently simple and do not require their interaction with classes and aspects to be specified. This is the case with development aspects, such as the ones implementing tracing. LSD does not impose restrictions that constrain the development of aspects in these situations, since aspects are not required to implement design rules.

An important question that arises when we assume that developers should establish design rules between classes and aspects is whether we are compromising or not the *obliviousness* characteristic expected from AOP, and consequently limiting its usefulness. But, as there are several definitions and different degrees of obliviousness (as discussed by Sullivan et al. (2005)), we summarize some of them below:

Language-level obliviousness: class developers do not need to use any type of signal or explicit notation (callback hooks, annotations or macros) within classes in order to activate aspects.

Feature obliviousness: class developers design the code exposing event hooks (join points) in accordance with aspect developers, but still unaware of the features that aspects implement.

Designer obliviousness: class developers are completely oblivious to the existence of aspects, designing in the same way as they normally would in the absence of aspects.

Our approach does not provide *designer obliviousness* because developers usually design classes and aspects differently to facilitate the interaction between classes and aspects. However, both *language-level* and *feature obliviousness* are met by our solution. We do not see this as a limitation because we and other authors (Sullivan et al., 2005) argue that AO software developed under oblivious designer conditions frequently results in complex and highly coupled software. Designer obliviousness discourages the use of AOP to implement software with numerous aspects or in naturally complex software.

7. Related work

Modularity issues in aspect-oriented programming have been reported by different authors (Sullivan et al., 2005, 2010; Clifton and Leavens, 2002; Steimann, 2006). For instance, Clifton and Leavens argue that existing *aspect-oriented* programming languages do not contribute to comprehensibility, since they require systems to be studied in their entirety. In order to improve comprehensibility of AOP systems, they propose a simple set of restrictions that minimizes this problem (Clifton and Leavens, 2002). In their proposed approach, aspects are categorized as *observers* – aspects that do not change the effective specifications of the modules, or as *assistants* – aspects that might change the modules specifications. Observers preserve modular reasoning even in the cases that advised modules do not make explicit references to them. On the other hand, Leavens and Clifton argue that, in order to preserve modular reasoning, modules that are advised by assistants should make explicit references to them. A new construct (`accept type`) was proposed to indicate that a module (class or interface) accepts to be advised by an assistant aspect. This construct aim at indicating to the class developer that an *assistant* aspect exists, and should be studied when some maintenance task needs to be executed. In spite of that, it maintains the feature obliviousness with respect to the *observer* aspects. LSD also indicates to class developer the aspects that should be studied, through the design rules specification. These design rules contain information about aspects like inter-type declarations, pointcuts, advices and required/prohibited behavior. In doing so, LSD supports the *information hiding* principle, keeping the class developer away from the complete implementation of the aspect.

Other authors focus on discussing how to expose more stable aspect-oriented interfaces and how to compute module interfaces in AOP systems. Since these later works are more related to our proposal, we discuss them in more detail in the remaining of this section.

7.1. Open Modules

Open Modules is an approach for dealing with modularity issues in AOP (Aldrich, 2005). Besides exposing data structures and functions, Open Modules interfaces can also expose pointcuts denoting internal semantic of events. Clients of these modules are able to advise only exported pointcuts, introducing a form of encapsulating join points occurring inside a module and protecting them from external advising. As a result, the module developer is responsible to maintain the semantics of the exported pointcuts, which, in fact, mitigates the *fragile pointcut* problem. Although join point hiding is an important concern, it does not provide information to the aspect developer (beyond exported join points and public methods), like required/prohibited inter-type declarations, method calls, attribute accesses and assignments, that are useful for decoupling classes and aspects, and consequently their developers. Differently from LSD, Open Modules does not offer any mechanisms for describing aspect developers responsibilities and the relationship between the types that participate of a collaboration. As a consequence, module developers are still able to unintentionally implement part of a concern assigned to a different team because they cannot assume the existence of any behavior expected to be modularized as an aspect.

7.2. Crosscutting Programming Interfaces (XPIs)

Sullivan et al. (2005) presented a comparative analysis between an AO system developed following the widely cited oblivious approach and the same system developed with clear design rules that document interfaces between classes and aspects. This last approach promises benefits when relevant crosscutting behavior are anticipated and when new code, anticipated or not, can be written against existing interfaces (design rules). Its main problem is expressing the design rules in natural language, leading to sometimes verbose and ambiguous interpretation. With LSD, we can express part of these design rules in an unambiguous language and also check if they are being respected, as discussed in Section 6.

In a subsequent work (Griswold et al., 2006; Sullivan et al., 2010), they propose to express and check the design rules using AspectJ. They called this approach Crosscutting Programming Interfaces (XPI) (more details and examples in Section 6). XPIs authors argue that the main advantages of their approach is that: (a) it does not require any new construct in the AspectJ language; and (b) there is no restriction to the pointcut visibility. In fact, this characteristic encourage the use of XPIs in sites that are already using the AspectJ language. However, some constraints required for defining the responsibilities of both class developers and aspect developers cannot be checked using the proposed XPI language (a deeper comparison of our approach against XPIs is presented in Section 6). Consequently, it is not possible to guarantee, at least automatically, if certain design rules are being obeyed, like LSD does. Although it is possible to check part of the design rules using XPIs, the use of a language not designed to this purpose leads frequently to complex specifications (contracts imposed by aspects).

7.3. Aspect-Aware Interfaces (AAI)

Kiczales and Mezini (2005) defend that the complete interface of a module can only be determined once the complete configuration of the systems is known. They introduce the notion of aspect-aware interfaces, which describe the existing dependencies between classes and aspects, giving support to reasoning about the effect of aspects over classes by pointing out where aspects are affecting a certain class. This interface must be automatically recomputed whenever classes or aspects change and, in fact, tools like AJDT already offer similar functionality through IDE resources that indicate which advices apply in a certain point. Also, these AAIs can be saved and compared with newer ones to detect changes to the set of join points intercepted by aspects. This approach helps to identify missing and accidental join points. Although very useful for evolution, this kind of interface does not support earlier software development phases, when it is necessary to partition the system into classes and aspects, and distribute tasks among development teams. At that moment, it is necessary to establish design rules that will govern both class and aspects development. LSD serves exactly to this purpose and brings automatic checking of the developed code.

7.4. Program Description Logic (PDL)

PDL (Morgan et al., 2007), inspired by the pointcut language in AspectJ, allows succinct declarative definitions of programmatic structures which correspond to design rule violations. They express undesirable declarations in code, like public attributes declarations and calls from certain places and inheritance declarations. The general idea is explicitly declaring what is prohibited and analyze the program to check if something is not obeyed. As a result it can be used to express some design rules. However, PDL has two important limitations: it is useful only in OO systems, because the PDL does not support AOP, and it does not support a clear definition of rules for parallel development of classes and aspects. The cause for this second limitation, which is similar to that found in the `declare warning` and `declare error` constructs from AspectJ, is that it can only express what is prohibited instead of expressing what is required. For instance, with both AspectJ and PDL it is not possible to enforce a method call within other method, but it is possible to prohibit a method call within other method. Our approach seems to better address that.

7.5. Explicit and Implicit Join Point Announcement

Both *Explicit Join Points* (*EJP*) (Hoffman and Eugster, 2007) and Ptolemy (Rajan and Leavens, 2007, 2008) make the shift towards explicit announcement, sharing the common idea that classes must explicitly announce the join points that are exposed to aspects and its constraints (context). They explore new possibilities in the language design space that opens up when we assume that classes (and aspects) are aware of crosscutting aspects.

Ptolemy is a programming language that provides quantified, typed events. It joins ideas from implicit invocation (II) and AO programming languages. Implicit invocation languages have explicitly announced events, which runs registered observer methods (implicitly). On the other hand, AO languages have implicitly announced events, called join points which implicitly dispatch advice executions. Ptolemy supports the declaration of event types, which have a name that can be used in quantification. An event type also declares the types of information communicated between events and observer methods for events of its type, partially performing the role of an interface. Events are declaratively identified using event expressions that name the event's type. As event announcements are scattered and tangled throughout the code and it is possible to tell when an "advice" will execute, Ptolemy does not provide *language-level obliviousness* because developers use explicit signals to activate aspects. However, it provides *feature obliviousness* because developers are still unaware of the features that aspect implement. The language design is closer to II languages than AO languages. Recently they extended Ptolemy with the concept of *translucid contracts* (Bagherzadeh et al., 2010, 2011a,b), which establishes a contract on the code that announces events and obligations to the code that handles events. This approach is conceptually similar to XPI's *provides* and *requires* clauses and to LSD design rules, which impose constraints over classes and aspects.

Explicit Join Points (*EJP*) serves as an explicit representation of an abstract contract between classes and aspects, modeling the information required by the crosscutting concern and also constraints to be enforced upon classes wishing to be advised. Syntactically, an EJP resembles a static method call qualified by an aspect, but it can be associated with a block of statements the aspect is to advise. In contrast to method invocation, however, the target class is not predefined, and the classes are not coupled to any specific implementation. EJP introduced pointcuts whose scope is restricted to where they are defined in the base code. However, their relationship of pointcuts and explicit join points is the inverse of Ptolemy: while Ptolemy introduces polymorphic pointcuts as (class) local join point type predicates and add explicit join point creation as a natural extension (analogous to an anonymous subclass, specifying a pointcut with a single match), EJP's explicit join points govern over local pointcuts so that the latter are unavailable without the former. Also, EJP's explicit join points are typeless (no opportunity of subtyping) whereas in Ptolemy they are only bound to a join point type and its supertypes, which can be advised by any aspect declaring to do so.

Based on EJP and Ptolemy, Steimann et al. (2010) proposes IIIA, a type-based modularization of implicit invocation (in contrast to the explicit announcement approach of EJP and Ptolemy) with implicit announcement. They were inspired by the throws clause and exceptions in Java, but in place of an exception, they join point types. This approach requires that classes exhibit join points explicitly, as declared by an exhibits *join point type* clause. It also turns the dependencies of aspects explicit, by requiring them to declare – through an advises *join point type* clause – instances of which join point types they intend to advise. IIIA seems to be a better solution than EJP and Ptolemy because it supports implicit invocation but also supporting explicit announcement. It also supports *polymorphic pointcuts* that enables classes that implement the same join point type to use different pointcut expressions.

These approaches have some conceptual similarity with LSD because they assume that class developers must be aware of aspects and respect some design rules. But, in contrast to Ptolemy, EJP and IIIA, LSD tries to concentrate both class and aspect constraints in a unique interface (design rule) that is known by these developers. Other difference is that the design rules serve as guidelines during development and can (and should) be established before classes and aspects development.

7.6. *Discoverying stable abstractions for pointcut interfaces*

One interesting challenge regarding the pragmatics of LSD is how to discover stable design rules for decoupling aspect and class development. In this context, Van Landuyt et al. (2009) present a process for identifying abstractions that lead to reusable pointcut signatures. Their idea follows a top-down approach, where relevant crosscutting concerns are identified and reasoned about throughout the architecture design. Using their approach, developers should be able to "build pointcut interfaces that are resilient to evolution" (Van Landuyt et al., 2009).

Although this concept of resilient interfaces is very close to our notion of design rules, the interfaces discussed in Van Landuyt et al. (2009) mainly expose stable join points. In fact, this kind of interface could mitigate the *fragile pointcut problem*, but they are not expressive enough to clearly present which are the obligations of class and aspect developers. For instance, using the mentioned approach, architects could not be able to find that calls to a specific method have to occur within a specific advice. It is a matter of future work adapt their approach to find the other kinds of interfaces discussed in this paper.

7.7. *Architectural Description Languages*

Architecture Description Languages (ADLs) support creating and documenting software architecture design, thereby enabling early verification of software properties (before its implementation). These properties can be considered as design rules. The architecture model – which consists of abstractions such as *components* and *connectors* – used by the majority of ADLs serves as a specification during the codification process. Therefore, it is necessary to map the architectural properties to some representation at source code level in order to guarantee that the design rules established by the ADL are obeyed by the implemented types. These ADL may be extended to also support *aspects* (Navasa et al., 2002). Dósea (2008) proposed a set of mappings to translate architectural properties defined by an ADL which uses AO concepts (AspectualAcme ADL (Batista et al., 2006b,a)) into design rules in LSD. Such verification may guarantee that both modularity and architectural properties obtained by the architecture design are valid in the implemented source code. They have also implemented a tool to automatically translate a valid AspectualAcme specifications into design rules in LSD. Besides reducing both effort and time required to perform translations, it avoids errors. In summary, design rules defined at architectural level using AspectualAcme can be translated to design rules in LSD at programming language level. The main benefit of this approach is that developers are warned whenever these design rules are violated, helping to avoid design erosion during software construction and maintenance.

ArchJava (Aldrich et al., 2002) seamlessly unifies software architecture with implementation, ensuring that the implementation conforms to architectural constraints. It extends Java to incorporate architectural features and enforce communication integrity. ArchJava adds new language constructs to support components, connections, and ports, aiming to allow programmers to describe software architecture. As a result, source code in ArchJava serves as a canonical representation of the architecture. They argue that the approach offers better program understanding, reliable architectural reasoning about code, keeping architecture and code consistent as they evolve, and encouraging more developers to take advantage of software architecture. As mentioned before, LSD is not a ADL but it is possible to translate design rules at architectural level to it. Another difference is that LSD is focused on improving the modular reasoning through the establishment of design rules in the source code level to AO systems, while ArchJava focus on OO systems.

8. Conclusion

In this work we present LSD, a language for specifying design rules in systems implemented with AspectJ. LSD assists developers to declare these design rules using an unambiguous language. We argue that through the establishment of design rules it is possible to obtain crosscutting modularity and preserve class modularity. However, besides defining design rules, it is important to have a language (with a defined semantics) that supports expressing and automatically checking them against the code. The language semantics was specified in Alloy (Neto et al., 2009; Neto, 2010) and was fine-tuned with the assistance of the Alloy Analyzer (M. I. of Technology, 2013) and its support to automatic checking. We also implemented an extension to the AspectBench Compiler (abc) (M. University, 2013) to support LSD constructs. LSD expressivity was compared to the XPI approach (Griswold et al., 2006) using as case study Health Watcher aspects, such as the transaction management aspect.

We discuss how LSD, with its broader interface notion, can improve modularity and serve as a solution to eliminate ambiguity and reduce the complexity found in other approaches, like XPIs implemented with AspectJ (Griswold et al., 2006). With LSD, it is possible to independently develop classes, interfaces and aspects, as long as the design rules are preestablished. We cataloged some existing design rules from the Health Watcher system and tried to express them both in LSD and XPIs, and concluded that LSD can express more constraints than XPIs, in a more concise way.

LSD requires the creation of new artifacts (design rules) that demand enough experience from software designers. Additionally, developers must get used to new language constructs. Although it is true that LSD provides many new language constructs, we tried to keep them as similar as possible to Java syntax, but some complex constraints may demand complex design rules specifications. With this in mind, developers must be trained to get used to LSD constructs, naturally increasing the cost of adoption. In spite of that, explicitly expressing design rules, and specially being capable of checking them, eases the task of developing new classes and aspects and also adapting existing ones.

Design rules are important for many types of aspects, but, for general aspects that make no reference to classes or interfaces (no coupling), like the basic Tracing and Logging aspects, we have little benefit from the use of design rules. LSD does not impose restrictions that constrain the development of these basic aspects, since it is not obligatory that aspects implement design rules. We assume that developers are capable of determining when it is worth to establish the design rules, in other words, if the design is mature enough to pay the price of writing a design rule.

Another important point is that although we consider defining design rules as a necessary step to modular AO development, it is possible to write classes and aspects, and later establish the design rules that were used. In this case, they will not help during development but will be useful for preventing errors and to give assistance to developers during software maintenance and evolution, like Aspect-Aware Interfaces (Kiczales and Mezini, 2005) (Section 7.3). This approach can be used whenever the best design is unknown (lack of experience) or in agile development processes, like Extreme Programming (Beck and Andres, 2004).

9. Future work

We intend to extend LSD through the addition of constructs, such as invariants, pre- and postconditions that can only be checked dynamically, probably following a similar approach to the used in JML (Leavens et al., 2006; Rebêlo., 2008; Rebêlo et al., 2008a; Rebêlo et al., 2008b; Rebêlo et al., 2009). Although, an important difference is that the constraints would be written within DRs and not in the classes. This exposes the constraints that must be respected by the developer. Also, developers of other related components can count on these constraints before the real components exist.

As another future work, we aim at building a tool for checking DR consistency, *i.e.*, if it is possible to write a program that satisfies the DR. This is useful because the developer responsible for the DR can define contradictory constraints that cannot be matched by any program. The Alloy Analyzer can be useful for implementing this tool, since it performs a complete analysis and searches by some instance. If no instance can be found, the DR is inconsistent. In addition, the Alloy Analyzer contains a functionality (the unsat core) that allows us to extract the minimum set of constraints that is making the model inconsistent, helping to identify which constraints are contradictory.

Also, we plan to perform a revision of the existent translations form LSD to Alloy to cover all LSD features. Then, based on these translations, we can implement a new translation tool from LSD to Alloy, but using abc (M. University, 2013). This alternative allows the reuse of the existent parser for LSD (from COLA).

Another future work is better evaluating LSD. First, we plan to build two versions of a completely new system using LSD and XPIs and compare them considering several perspectives, including reuse. Secondly, we plan to compare LSD with other approaches, including Semmle Code (Verbaere et al., 2007), PDL (Morgan et al., 2007) and Design Wizard (Brunet et al., 2009), aiming to identify other kinds of design rules and verify if LSD can check them. As a result we can extend LSD to support more design rules.

Besides helping to specify DRs in a declarative manner and having a tool support to verify if the DRs are being respected, we intend to implement an IDE (probably Eclipse (Eclipse.org, 2013a)) extension to support design rule creation and checking. This could be very useful for pointing to the developer the exact constraint that is not being respected. Also, this integration could include a visualization tool to, based on a design rule instance, give a view of the system configuration.

Finally, DRs written in LSD could be used to generate classes, interfaces and aspects based on their design rule specifications. One future work is implementing a tool integrated to Eclipse (Eclipse.org, 2013a). This feature is commonly found in UML (Booch et al., 1999) modeling tools (Harrison et al., 2000) supporting the generation of components based on models.

Acknowledgments

This work was partially supported by CNPq/Brazil, Grant 308383/2008-7, and the National Institute of Science and Technology for Software Engineering (INES[4]), funded by CNPq and FACEPE, Grants 573964/2008-4 and APQ-1037-1.03/08. Fernando is also supported by CNPq/Brazil (306619/2011-3), FACEPE/Brazil (APQ-0395-1.03/10 and APQ-1367-1.03/12).

References

Akai, S., Chiba, S., Nishizawa, M., 2009. Region pointcut for AspectJ. In: Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'09), Charlottesville, USA, pp. 43–48.
Aldrich, J.,2005. Open modules: modular reasoning about advice. In: Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP'05). Springer, Glasgow, UK, pp. 144–168.
Aldrich, J., Chambers, C., Notkin, D.,2002. Archjava: connecting software architecture to implementation. In: Proceedings of the 24th International Conference on Software Engineering (ICSE'02). ACM, New York, NY, USA, pp. 187–197.
Alur, D., Crupi, J., Malks, D., 2003. Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall Ptr, New Jersey, USA.
Alves, V., 2007. Implementing software product line adoption strategies. PhD Thesis. Informatics Center, Federal University of Pernambuco, Recife, Brazil.
Alves, V., Borba, P., 2004. An incremental aspect-oriented product line method for J2ME game development. In: Workshop on Managing Variability Consistently in Design and Code, in conjunction with the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04).
Alves, V., Matos Jr., P., Cole, L., Borba, P., Ramalho, G.,2005. Extracting and evolving mobile games product lines. In: Proceedings of the 9th International Software Product Line Conference (SPLC'05). Lecture Notes in Computer Science, vol. 3714. Springer-Verlag, Rennes, France, pp. 70–81.
Aotani, T., Masuhara, H.,2007. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In: Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07). ACM, New York, NY, USA, pp. 161–172.
Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Moor, O.D., Sereni, D., Sittampalam, G., Tibble, J.,2005. abc: an extensible AspectJ compiler. In: Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05). ACM Press, Chicago, USA, pp. 87–98.
Avgustinov, P., Ekman, T., Tibble, J.,2008. Modularity first: a case for mixing AOP and attribute grammars. In: Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08). ACM, New York, NY, USA, pp. 25–35.
Bagherzadeh, M., Leavens, G.T., Dyer, R.,2011a. Applying translucid contracts for modular reasoning about aspect and object oriented Events3. In: Proceedings of the 10th International Workshop on Foundations of Aspect-oriented Languages (FOAL'11). ACM, New York, NY, USA, pp. 31–35.
Bagherzadeh, M., Rajan, H., Leavens, G.T., Mooney, S.,2010. Translucid contracts for modular reasoning about aspect-oriented programs. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (SPLASH'10). ACM, New York, NY, USA, pp. 245–246.
Bagherzadeh, M., Rajan, H., Leavens, G.T., Mooney, S.,2011b. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In: Proceedings of the Tenth International Conference on Aspect-oriented Software Development (AOSD'11). ACM, New York, NY, USA, pp. 141–152.
Baldwin, C.Y., Clark, K.B., 1999. Design Rules: The Power of Modularity, vol. 1. MIT Press, Cambridge, MA, USA.
Batista, T., Chavez, C., Garcia, A., Kulesza, U., Sant'Anna, C., Lucena, C., 2006a. Aspectual connectors: supporting the seamless integration of aspects and ADLs. In: Proceedings of the 20th Brazilian Symposium on Software Engineering (SBES 2006).
Batista, T., Chavez, C., Garcia, A., Rashid, A., Sant'Anna, C., Kulesza, U., Filho, F.C.,2006b. Reflections on architectural connection: seven issues on aspects and ADLs. In: Proceedings of the 2006 International Workshop on Early Aspects at ICSE (EA'06). ACM, New York, NY, USA, pp. 3–10.
Beck, K., Andres, C., 2004. Extreme Programming Explained: Embrace Change, 2nd ed. Addison-Wesley Professional, Boston, USA.
Booch, G., Rumbaugh, J., Jacobson, I., 1999. The Unified Modeling Language User Guide. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
Brunet, J., Guerrero, D., Figueiredo, J., 2009. Design tests: an approach to programmatically check your code against design rules. In: 31st International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results.
Chen, K., Chien, C.-H.,2006. Extending the field access pointcuts of AspectJ to arrays. In: 20th International Conference on Supercomputing (ICS'06). ACM Press, New York, NY, USA, pp. 470–475.
Clements, P., Northrop, L.M., 2001. Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Clifton, C., Leavens, G.T., 2002. Observers and assistants: a proposal for modular aspect-oriented reasoning. In: Foundations of Aspect Languages (FOAL'2002), in Conjunction with the 1st International Conference on Aspect-Oriented Software Development (AOSD'02), pp. 33–44.
Dantas, A., Alves, V., Borba, P., 2003. Using aspects to structure small devices adaptive applications. In: First Workshop on Reuse in Constrained Environments, in Conjunction with the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03).
Dantas, A., Borba, P., 2003. Developing adaptive j2me applications using AspectJ. In: VII Brazilian Symposium on Programming Languages (SBLP'03), pp. 226–242.
Devanbu, P., Balzer, B., Batory, D., Kiczales, G., Launchbury, J., Parnas, D., Tarr, P.,2003. Modularity in the new millenium: a panel summary. In: Proceedings of the 25th International Conference on Software Engineering (ICSE'03). IEEE Computer Society, Washington, DC, USA, pp. 723–724.
Eclipse.org, 2013. Eclipse. At http://www.eclipse.org
Eclipse.org, 2013. Eclipse Developer Guide: ajc – the AspectJ Compiler/Weaver. At http://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html
Ekman, T., Hedin, G.,2007. The JastAdd extensible Java compiler. In: Proceedings of the 22nd Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'07). ACM, New York, NY, USA, pp. 1–18.
Felleisen, M., 1990. On the expressive power of programming languages. In: Science of Computer Programming. Springer-Verlag, Copenhagen, Denmark, pp. 134–151.

[4] http://www.ines.org.br

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading, USA.

Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Santanna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A., 2007a. 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).

Greenwood, P., Garcia, A., Rashid, A., Figueiredo, E., Sant'Anna, C., Cacho, N., Sampaio, A., Soares, S., Borba, P., Dosea, M., Ramos, R., Kulesza, U., Bartolomei, T., Pinto, M., Fuentes, L., Gamez, N., Moreira, A., Araujo, J., Batista, T., Medeiros, A., Dantas, F., Fernandes, L., Wloka, J., Chavez, C., France, R., Brito, I.,2007b. On the contributions of an end-to-end AOSD testbed. In: Proceedings of the Early Aspects (EARLYASPECTS'07), in Conjunction with the 29th International Conference on Software Engineering (ICSE'07). IEEE Computer Society, Washington, DC, USA, p. 8.

Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H., 2006. Modular software design with crosscutting interfaces. IEEE Software 23 (1), 51–60.

Harbulot, B., Gurd, J.R.,2006. A join point for loops in AspectJ. In: Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06). ACM, New York, NY, USA, pp. 63–74.

Harrison, W., Barton, C., Raghavachari, M.,2000. Mapping UML designs to Java. In: Proceedings of the 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00). ACM, New York, NY, USA, pp. 178–187.

Hedin, G., 2009. Generating language tools with JastAdd. In: Proceedings of the 3rd Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09).

Hoffman, K., Eugster, P.,2007. Bridging Java and AspectJ through explicit join points. In: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ'07). ACM, New York, NY, USA, pp. 63–72.

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

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97).

Kiczales, G., Mezini, M.,2005. Aspect-oriented programming and modular reasoning. In: Proceedings of the 27th International Conference on Software Engineering (ICSE'05). ACM, New York, NY, USA, pp. 49–58.

Leavens, G.T., Baker, A.L., Ruby, C., 2006. Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Software Engineering Notes 31 (3), 1–38.

Lopes, C.V., Bajracharya, S.K., 2006. Assessing aspect modularizations using design structure matrix and net option value. In: LNCS Transactions on Aspect-Oriented Software Development I, pp. 1–35.

Dósea, Marcos Barbosa, 2008. Uma Abordagem Modular para Projeto de Software Orientado a Aspectos. Master's Thesis. Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil.

Morgan, C., Volder, K.D., Wohlstadter, E.,2007. A static aspect language for checking design rules. In: Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07). ACM Press, New York, NY, USA, pp. 63–72.

Navasa, A., Pérez, M.A., Murillo, J.M., Hernández, J., 2002. Aspect oriented software architecture: a structural perspective. In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02).

Neto, A.C., 2010. Specifying design rules in aspect-oriented systems. PhD Thesis. Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil.

Neto, A.C., de Medeiros Ribeiro, M., Dósea, M., Bonifácio, R., Borba, P., 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).

Neto, A.C., Marques, A., Gheyi, R., Borba, P., Filho, F.C., 2009. A design rule language for aspect-oriented programming. In: XIII Brazilian Symposium on Programming Languages (SBLP'09), Gramado, RS, Brazil, pp. 131–144.

M. I. of Technology, 2013. Alloy Analyzer. At http://alloy.mit.edu/alloy4/

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

Rajan, H., Leavens, G.T., 2007. Quantified, typed events for improved separation of concerns. Technical Report 07-14. Department of Computer Science/Iowa State University, Iowa, USA.

Rajan, H., Leavens, G.T.,2008. Ptolemy: a language with quantified, typed events. In: Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08). Springer-Verlag, Berlin, Heidelberg, pp. 155–179.

Rebêlo, H., 2008. Implementing JML contracts with AspectJ. Master's Thesis. Department of Computing and Systems, University of Pernambuco, Recife, Brazil.

Rebêlo, H., Lima, R., Cornélio, M., Leavens, G.T., Mota, A., Oliveira, C., 2009. Optimizing JML feature compilation in Ajmlc using aspect-oriented refactorings. In: XIII Brazilian Symposium on Programming Languages (SBLP'09), pp. 117–130.

Rebêlo, H., Lima, R.M.F., Cornélio, M., Soares, S., 2008a. A JML compiler based on AspectJ. In: First International Conference on Software Testing Verification and Validation (ICST'08), pp. 541–544.

Rebêlo, H., Soares, S., Lima, R.M.F., Ferreira, L., Cornélio, M., 2008b. Implementing java modeling language contracts with AspectJ. In: Proceedings of the 23rd Symposium on Applied Computing (SAC'08), pp. 228–233.

Ribeiro, M., Dósea, M., Bonifácio, R., Neto, A.C., Borba, P., Soares, S., 2007. Analyzing class and crosscutting modularity with design structure matrixes. In: Proceedings of 21th Brazilian Symposium on Software Engineering (SBES'07), pp. 167–181.

Soares, S., 2004. An aspect-oriented implementation method. PhD Thesis. Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil.

Soares, S., Laureano, E., Borba, P.,2002. Implementing distribution and persistence aspects with AspectJ. In: Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02). ACM, New York, NY, USA, pp. 174–190.

Steimann, F., 2006. The paradoxical success of aspect-oriented programming. Proceedings of the 21st International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06). ACM SIGPLAN Notices 41 (10), 481–497.

Steimann, F., Pawlitzki, T., Apel, S., Kästner, C., 2010. Types and modularity for implicit invocation with implicit announcement. ACM Transactions on Software Engineering and Methodology 20, 1:1–1:43.

Steward, D.V., 1981. The design structure system: a method for managing the design of complex systems. IEEE Transactions on Engineering Management 28 (3), 71–74.

Störzer, M., Koppen, C., 2004. PCDiff: attacking the fragile pointcut problem, abstract. In: European Interactive Workshop on Aspects in Software, Berlin, Germany.

Sullivan, K., Griswold, W.G., Rajan, H., Song, Y., Cai, Y., Shonle, M., Tewari, N., 2010. Modular aspect-oriented design with XPIs. ACM Transactions on Software Engineering and Methodology 20 (September), 5:1–5:42.

Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.,2005. Information hiding interfaces for aspect-oriented design. In: Proceedings of the 10th European Software Engineering Conference and the 13th International Symposium on Foundations of Software Engineering (ESEC/FSE'05). ACM, New York, NY, USA, pp. 166–175.

M. University, 2013. abc: the Aspectbench Compiler for AspectJ. At http://www.sable.mcgill.ca/abc/

Van Landuyt, D., Op de beeck, S., Truyen, E., Joosen, W.,2009. Domain-driven discovery of stable abstractions for pointcut interfaces. In: Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09). ACM, New York, NY, USA, pp. 75–86.

Verbaere, M., Hajiyev, E., De Moor, O.,2007. Improve software quality with SemmleCode: an eclipse plugin for semantic code search. In: 22nd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07) – Demo. ACM, New York, NY, USA, pp. 880–881.

Alberto Costa Neto is an Assistant Professor at the Computing Department at the Federal University of Sergipe, Brazil. His main research interests are: software modularity, software product lines and programming languages.

Rodrigo Bonifacio is an Assistant Professor at the Computer Science Department, University of Brasilia. His main research interests are: software reuse and modularity, empirical software engineering, and programming languages.

Márcio Ribeiro is an assistant professor in the Computing Institute at Federal University of Alagoas. He holds the ACM SIGPLAN John Vlissides Award (2010) and a Doctoral degree in Computer Science from the Federal University of Pernambuco. He has experience in Software Engineering and his main research interests are on Software Product Lines and Software Modularity. More on Ribeiro can be found at http://www.ic.ufal.br/marcio.

Carlos Eduardo Pontual is a MSc in Computer Science and currently works with software systems for mobile devices. His main research interests are mobile device systems, software modularization and software product lines.

Paulo Borba is Professor of Software Development at the Informatics Center of the Federal University of Pernambuco, Brazil, where he leads the Software Productivity Group. His main research interests are in the following topics and their integration: software modularity, software product lines, and refactoring.

Fernando Castor is an Assistant Professor at the Federal University of Pernambuco (UFPE) – Brazil. He holds a PhD in Computer Science from UNICAMP – Brazil. He is interested in developing techniques, tools, and methods to ease the construction and maintenance of large-scale dependable systems. His main research areas are exception handling, fault tolerance, concurrent programming, and software modularity.