Modular Contracts with Procedures, Annotations, Pointcuts and Advice

Henrique Rebêlo, Ricardo Lima, and Gary T. Leavens,

CS-TR-11-05
September, 2011

Keywords: Aspect-oriented programming, programming by contract, modularity

2011 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, AOP, AspectJ; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, AOP, AspectJ; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

To appear in SBLP 2011.

Dept. of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA

Modular Contracts with Procedures, Annotations, Pointcuts and Advice

Henrique Rebêlo[1], Ricardo Lima[1], and Gary T. Leavens[2]

[1] Federal University of Pernambuco, Brazil
{hemr,rmfl}@cin.ufpe.br
[2] University of Central Florida, USA
leavens@eecs.ucf.edu

Abstract. There are numerous mechanisms for modularizing design by contract at the source code level. Three mechanisms have been the main focus of attention, metadata annotations, pointcuts and advice. The latter two are well-known aspect-oriented programming mechanisms, and according to the literature, fare better in achieving contract modularization. However, previous efforts aimed at supporting contract modularity actually hindered it. In this paper we report an enhanced use of pointcuts and advice, and show how crosscut programming interfaces (XPIs) can significantly improve contract modularity. In addition, we also discuss how these XPIs can be used together with annotations to tackle the pointcut fragility problem and minimize the limited enforcement of XPI interface rules. We compare our approach with the literature's in terms of code locality, well-defined interfaces, reusability, changeability, fragility, and pluggability.

1 Introduction

Design by Contract (DbC) is a technique for developing and improving functional software correctness [17]. The key mechanism in DbC is the use of the so-called "contracts". A contract formally specifies an agreement between a client and its suppliers. Clients must satisfy the supplier's conditions before calling one of the supplied methods. When these conditions are satisfied, the supplier guarantees certain properties, which constitute the supplier's obligations. However, when a client or supplier breaks a condition (contract violation), a runtime error occurs. The use of such pre- and postconditions to specify software contracts dates back to Hoare's 1969 paper on formal verification [10].

In this context, numerous mechanisms have been developed to instrument, modularize, and document contracts at source code level, including procedures, aspect-oriented programming mechanisms and others. In this paper we focus on three mechanisms to deal with design by contract modularization: metadata annotations [2, 3], and pointcuts and advice [11]. The latter two are well-known aspect-oriented programming (AOP) mechanisms. These mechanisms are attracting significant research on DbC [4, 6, 21] and the special case of contracts known as design rules [18, 23].

It is often claimed in the literature [11, 4, 16, 6] that the contracts of a system are de-facto a crosscutting concern and fare better when modularized with AOP mechanisms such as pointcuts and advice. To the best of our knowledge, Balzer, Eugster, and

Meyer [1] were the first to investigate the adequacy of aspects to modularize DbC. They conclude that the use of aspects hinders design by contract implementation and fails to achieve the main DbC principles such as contract inheritance.

In this paper, we go beyond Balzer, Eugster, and Meyer's work with an improved understanding of why the common uses of AOP have failed to properly modularize contracts with pointcuts and advice mechanisms. We introduce an enhanced use of such pointcuts and advice mechanisms to modularize contracts. This enhanced use relies on well-defined interfaces, known as crosscut programming interfaces, or XPIs [23]. The key idea behind the use of XPIs is to introduce a crosscutting programming interface for design constraints of the design by contract concern. Note that throughout this paper, two related, but distinct concepts are used: *contract* (description of the operations using pre- and postconditions) and *design constraint* (a pre-/postcondition, or an invariant). Hence, we can say that a contract may have one or more design constraints.

To this end, we present a classical example with four design constraints (contracts) and a comparison between six implementations of this example regarding a set of modularization mechanisms (including procedures and XPIs). We also illustrate, through XPIs (pointcuts and advice), how we can properly preserve DbC principles such as contract inheritance [17, 13]. In addition, we also discuss how XPIs can be used together with annotations to tackle the pointcut fragility problem [22].

We show that XPIs and annotations, unlike the previous efforts [11, 4, 16, 6], fare better to modularize the design by contract concern using a modularity criteria [12]. We say the code that implements a design constraint is *modular* if:

(i) it is textually local (i.e. not scattered),
(ii) there is a well-defined interface that is an abstraction of the implementation,
(iii) the implementation of a particular design constraint can be reused if applied to other parts of the same system (e.g. contract inheritance [17]),
(iv) while performing change tasks, just the related design constraint's modules are examined or changed and no new aspect is added if the change tasks are related to existing design constraints,
(v) a particular refactoring (e.g. rename method [8]) does not invalidate the application of a particular design constraint to one or a set of join points, and
(vi) we can remove or compose design constraints into the system without being invasive (i.e. without modifications to the base code).

Based on this, this paper provides: (i) an understanding of how the XPIs (pointcuts and advice) and annotations mechanisms work together to modularize contracts in a system; (ii) an understanding of how the previous works can be improved; (iii) an analysis of the different mechanisms that considers code locality, interface, reusability, changeability, fragility, and pluggability.

This paper is structured as follows: Section 2 presents the example and its four design constraint concerns. Section 3 presents six implementation of the studied mechanisms. Section 4 analyzes in qualitative ways these implementations. Section 5 discusses related work and outlines open research issues. We finish with a summary in Section 6.

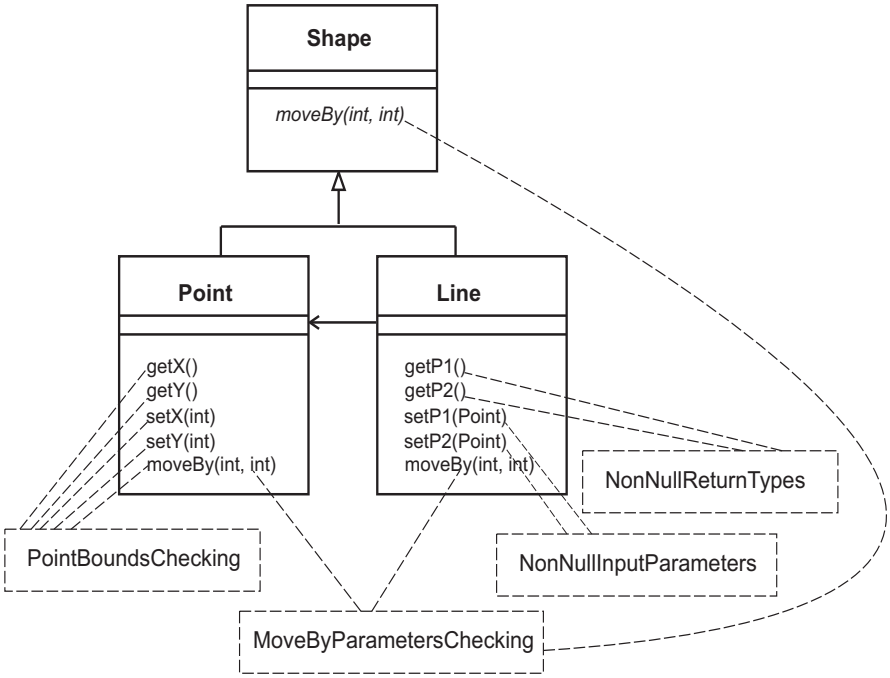

Fig. 1. The design of the traditional figure editor system [11, 12, 23], showing the main classes and four design constraint concerns that crosscuts methods in the `Shape`, `Point`, and `Line` classes.

2 A Running Example

This section introduces the example that will be used throughout the paper. It consists on a simple example from the literature: a figure editing system for editing drawings made up of shape objects (e.g. points) [11, 12, 23]. Its design is shown in Figure 1.

2.1 Four Design Constraint Concerns

In addition to concerns involving the core functionality of the shapes, the design comprises four key design constraint concerns, which are shown as dotted-line boxes in Figure 1. All these design constraint concerns are discussed and enforced in terms of pre- and postconditions, and invariants [17].

Point-Bounds-Checking – Denotes an invariant constraint on all methods of the `Point` class. This constraint states that the coordinates of a point are stored in particular bounds (between specific `MIN` and `MAX` values).

Non-Null-Input-Parameter – Denotes two preconditions that constrain the input object parameters, of methods `setP1` and `setP2` of `Line` class, to be non-null.

Non-Null-Return-Types – Denotes two postconditions that constrain the return object types, of methods `getP1` and `getP2` of `Line` class, to be non-null.

Move-By-Parameters-Checking – Denotes a precondition that constrains the input parameters of the method `moveBy` to be greater than or equal to zero. This precondition crosscuts three types (the interface `Shape` and `Point` and `Line` classes) that declares the method `moveBy`. Since this precondition is defined in the supertype `Shape`, it represents a contract inheritance [17].

Some of these constraints are interconnected in the sense that their realizations crosscut the same class. For instance, as observed in Figure 1, the `Line` class has its methods constrained by three of the four design constraint concerns.

3 Six Implementations

This section presents the code for six implementations of the running example (Figure 1). As discussed, the running example also comprises four design constraint concerns. Due to lack of space and for simplicity, we present the implementations of some design constraints (e.g. *Move-By-Parameters-Checking*) and informally argument about the others. A complete implementation of the four design constraints is available at [20]. We conduct an analyzes of these implementations in Section 4.

3.1 GOFP

This first implementation uses good old-fashioned procedures (GOFP) to implement the DbC concern code. In this implementation, each of the constrained methods of the figure editor system (Figure 1) includes a call to a procedure (a static method in Java). This call is placed at the beginning (for precondition code), at the end (for postcondition code) or both (for invariant code) of each constrained method. Note that this is the standard form commonly adopted in practice [5].

The following code illustrates the checking of the precondition constraint (stated by the *Move-By-Parameters-Checking* concern) of `moveBy` method declared at the `Point` class.

```
void moveBy(int dx, int dy) {
  JC.requires(dx >= 0 && dy >= 0,
              "dx is "+dx+" dy is "+dy);
  setX(getX() + dx);
  setY(getY() + dy);
}
```

The shadowed code shows the precondition constraint code. It checks whether or not the input parameters of method `moveBy` are greater than or equal to zero; if it is not, a precondition error is thrown to signal the contract violation. Note that the body of the procedure `requires` (shown below) encapsulates/modularizes the signaling error code (i.e. the **`throw`** clause).

However, the call to the procedure `requires` and its related error message are still tangled and scattered within the figure editor system. For example, the method `moveBy` of class `Line` (not shown) must have an identical call to this procedure in order to check the precondition constraint. The scattered call is a limitation of the object-oriented programming mechanisms [11, 12] (later we discuss how to avoid such scattering). Regarding the error message code, besides being tangled and scattered, it still makes the code more polluted and verbose. This error message code is really important in the context of DbC purposes [17, 14]. Once a precondition error is detected, a user should receive a detailed description of the violation in order to track and fix the error. A detailed error message could be in terms of class name, method name, context values (e.g. values of input parameters), and so forth.(The second shadowed line of the method `moveBy` illustrates the context values passing.)

```
static void requires(boolean constraint, String errorMsg) {
  if (!constraint)
    throw new PreconditionError(errorMsg);
}
```

This implementation illustrates the declaration of the procedure `requires` used to check precondition constraints. Let us assume that this procedure was declared in a Java class called `JC` (used to encapsulate all Java contract based operations). This is an alternative instead of Java assertions, since in Java we do not have other built-in support for DbC [17].

The postcondition and invariant constraint are implemented similarly. The main difference is that the procedure call is placed at the end of a constrained method (for the postcondition) or at beginning and end of a constrained method (for the invariant).

3.2 Enhanced-GOFP

This implementation enhances the previous one with the use of a *strategy* design pattern [9]. It is used to encapsulate the constraint and error message code. Hence, we need only to pass the input parameters and the name of the method used to compose a useful error reporting message. As a result, the precondition constraint is modularized in the `MoveByParametersCheckingTester` class. This implementation is not commonly used in practice [5], but it helps to reduce the tangling and scattered implementation of the precondition constraint related to the GOFP procedure.

```
void moveBy(int dx, int dy) {
  JC.requires(new MoveByParametersCheckingTester(dx, dy,
              "Point.moveBy"));
  setX(getX() + dx);
  setY(getY() + dy);
}
```

Even though we were able to modularize the precondition constraint and also the error reporting code in the strategy class `MoveByParametersCheckingTester`, we cannot remove the scattered calls to the procedure `requires` (we still have a similar call to

this procedure in the body of method `moveBy` in `Line` class). This is a limitation of the object-oriented code [11, 12]. Later in this section, we discuss how to improve these implementation through an enhanced use of pointcuts and advice.

3.3 Pointcuts-Advice

In this implementation, we show how the work described in the current literature [11, 4, 16, 6] uses the aspect-oriented programming mechanisms such as pointcuts and advice to implement and "modularize" the same discussed precondition. We evolve enhanced-GOFP implementations with these AOP mechanisms.

```
void moveBy(int dx, int dy) {
  setX(getX() + dx);
  setY(getY() + dy);
}

before(int dx, int dy):
  execution(void Point.moveBy(int, int))
  && args(dx, dy) {
     JC.requires(new MoveByParametersCheckingTester(dx, dy,
                 "Point.moveBy"));
}
```

A single **`before`** advice is used to modularize the call to procedure `requires` in `Point.moveBy`. For the `Line` class, a similar **`before`** advice is used; the only change is in the **`pointcut`** (i.e. **`execution`**`(`**`void`** `Line.moveBy(`**`int, int`**`))`). In the literature [11, 4, 16, 6], other authors employ an aspect per class. Hence, we have two aspects (one for class `Point` and other for the `Line` one) where each one has a **`before`** advice used to check the precondition constraint stated by the *Move-By-Parameters-Checking* concern.

However, as the reader can observe, this has the same scattering problem presented as in the GOFP and enhanced-GOFP implementations. As a consequence, the following question can be raised, "*What are the benefits of the aspectization of design by contract besides the physical separation?*". We answer this question based on some variations of this implementation and some analysis and discussions carried out in Sections 4 and 5.

3.4 Annotation-Pointcuts-Advice

This implementation uses Java 1.5 metadata annotations [2, 3]. Thus, each constrained method that follows the restrictions imposed by the *Move-By-Parameters-Checking* concern has a custom annotation that states that the execution of the constrained and annotated method should check the precondition.

```
@PointMoveByParametersChecking
void moveBy(int dx, int dy) {
  setX(getX() + dx);
  setY(getY() + dy);
}
```

The following **before** advice differs from the previous one (without annotations) in the sense that it intercepts the execution based on methods marked with the annotation `@PointMoveByParametersChecking`. It is written as:

```
before(int dx, int dy):
  execution(@PointMoveByParametersChecking * *(..))
  && args(dx, dy) {
     JC.requires(new BoundsCheckingTester(dx, dy,
                 "Point.moveBy"));
}
```

Supplying annotations. Aspect-oriented programming languages such as AspectJ offers specific static crosscutting constructs to supply (introduce) annotations in a crosscutting manner [3]. Hence, we do not need to directly mark the method declaration, instead we can perform the following:

```
declare @method: void Point.moveBy(int, int):
  @PointMoveByParametersChecking;
```

This declaration introduces the annotation `@PointMoveByParametersChecking` on method `Point.moveBy` in a crosscutting way. Such a mechanism is useful when we have several methods with the same annotation, leading to annotation clutter. Moreover, this AspectJ feature [3] works as a contributing factor to the pointcut fragility problem [22] encountered in AspectJ-like languages. If we apply a *renaming* method refactoring [8] in method `Point.moveBy`, we got a compile-time error saying that the method "`void Point.moveBy(int, int)`" does not exist. In this case the developer is forced to implement the exposed member. Note that this design rule enforcement [18] is not possible in the literature-based approach without metadata annotations.

3.5 Enhanced-Pointcuts-Advice

In this implementation, we show how the use of pointcuts and advice, in contrast to the literature [11, 4, 16, 6], can be enhanced. We seek an enhanced design by contract code modularization by exploring the quantification property of AOP. Quantification is one of the main benefits when adopting aspect-orientation [7, 24].

Another key idea behind our methodology is to combine pointcuts and advice with the notion of *crosscut programming interfaces*, or XPIs [23]. We use XPIs to introduce a short design phase before the design of the base and aspect code. During this design phase, for each (design constraint) concern, we define an (XPI) interface to decouple the base design an the aspect design.

The following code is related to the XPI declaration used to expose all the constrained join points by the *Move-By-Parameters-Checking* concern.

```
aspect XMoveByParametersChecking {
  public pointcut jp(int dx, int dy):
    execution(void Shape+.moveBy(int, int))
    && args(dx, dy);
}
```

By convention, aspects that specify XPIs begin with an "X" in order to distinguish them from non-interface aspects. The syntactic part of the XPI exposes one named pointcut (`jp`). The **`execution`**(**`void`** `Shape+.moveBy(`**`int, int`**`))` pointcut means execution of any method `moveBy` defined in `Shape` or a subclass of `Shape`, that returns **`void`** and take two integer arguments. In contrast to previous works [11, 4, 16, 6], our implementation of the precondition gains in reusability achieved by the AspectJ quantification property, in this case, expressed by the use of '+' (used to intercept a hierarchy).

The aspect code that uses the XPI for the *Move-By-Parameters-Checking* concern is written as:

```
aspect MoveByParametersPreconditionChecking {
 before(int dx, int dy):
   XMoveByParametersChecking.jp(dx, dy) {
     JC.requires(new BoundsCheckingTester(dx, dy,
                 "Shape+.moveBy"));
 }
}
```

The aspect now depends only on the abstract public pointcut signature denoted by `jp`. Unlike in the literature approach [11, 4, 16, 6], the pointcuts used within the advice code do not depend anymore on implementation details of the `Shape`, `Point`, or `Line` types. As the pointcut `jp` intercepts execution of any Shape objects (including subclasses), the precondition constraint code is reused and automatically applied to method `moveBy` of `Point` and `Line` classes. As a result, we augmented the reuse by using the quantification mechanism present in aspect-oriented languages such as AspectJ. Hence, instead of two **`before`** advice with a duplicated call to the procedure `requires` (precondition checking code), we only have a localized one. This is one example of how to properly implement the contract inheritance principle of DbC methodology [17].

One of the main objectives of the XPI interfaces is to guarantee that the exposed join points are really implemented in a system, thus avoiding the pointcut fragility problem [22]. However, the XPI approach has limited enforcement of interface rules [23]. The application of object-oriented refactorings [8] such as *renaming* can break the exposed join points in the XPI interfaces. In the following, we discuss how XPIs can be significantly improved by using metadata annotations in combination with supplying annotations [3] (previously discussed).

3.6 Enhanced-Annotation-Pointcuts-Advice

This implementation differs from the previous one only in the way that XPIs expose the join points. We combine the Java 1.5 metadata annotations [2] with the AspectJ supplying annotations mechanisms [3].

Consider the discussed XPI implementation (without annotations) of the *Move-By-Parameters-Checking* concern. With the application of metadata annotations, the exposed pointcuts become as follows:

```
aspect XMoveByParametersChecking {
  declare @method: void Shape+.moveBy(int, int):
    @MoveByParametersChecking;

  public pointcut jp(int dx, int dy):
    execution(@MoveByParametersChecking * *(..))
    && args(dx, dy);
}
```

The XPI now exposes all the `@MoveByParametersChecking` join points. The rest of the code to check the precondition (the advising code) remains the same as discussed. In addition, we mentioned that XPIs, without metadata annotations, have a limited enforcement of interface rules. For example, in the non-annotation approach, we can not ensure that the subclasses of `Shape` really implements the exposed join point by the XPI. Let us assume that we performed an OO *renaming* refactoring [8] to change the name of the method `moveBy`. If the programmer does not change the XPI as well, the crosscutting behavior will be discarded. This problem is caused by the pointcut fragility. However, by employing annotations, all the marked methods must exist during compile-time [1]. Hence, if the hierarchy of `Shape` replaces the name of the `moveBy` method with a new one, we got a compile-time error saying that the method `moveBy` does not exist in the Shape's hierarchy.

Therefore, the use of metadata annotations enhance the XPI specification when exposing all the join points of a particular concern. This approach helps to improve the enforcement (in a compile-time enforcement fashion) of interface (design) rules. The idea to ensure design rules through a well-defined interface is discussed by Neto *et al*.'s work [18]. They extend the AspectJ syntax with design rules that are enforced for both base and advised code. Hence, they provide more powerful design rules than those achieved by XPIs [23]. We rely on XPIs because they are AspectJ-based, require no new constructs, and they are currently available for AspectJ users (including our enhancements using metada annotations [2], Bonner).

4 Analysis of the Implementations

In this section, we present an analysis of the six implementations (non-AOP and AOP based) of the figure editor (Figure 1) regarding the four design constraints concerns identified in Section 2 and based on the different mechanisms investigated in this paper. The assessment uses six modularity criteria (summarized in Table 1 and discussed in Introduction Section): code locality, interface, reusability, changeability, fragility, and pluggability. The analysis is broken into three parts: (i) non-AOP implementations, (ii) AOP implementations, and (iii) reasoning about change. The last part uses the changeability modularity criterion.

[1] This can only be enforced for those join points that were explicitly defined without quantification.

Table 1. Analysis of the six implementations.

		Locality	Interface	Reusability	Changeability	Fragility	Pluggability
non AOP	GOFP	no	low	no	no	n/a	no
	EGOFP	medium	medium	medium	medium	n/a	no
AOP	PA(1)	medium	medium	medium	medium	yes	yes
	APA(2)	medium	medium	medium	medium	no	yes
	EPA(3)	high	high	high	high	yes	yes
	EAPA(4)	high	high	high	high	no	yes

(1) Pointcuts-Advice.
(2) Annotation-Pointcuts-Advice.
(3) Enhanced-Pointcuts-Advice.
(4) Enhanced-Annotation-Pointcuts-Advice.

4.1 The Non-AOP Implementation

In the non-AOP code, the GOFP implementation of the figure editor system (Figure 1) fails to satisfy our modularity criteria related to the four design constraints. Firstly, it is not localized. The GOFP implementation is good in the sense that it modularizes the signaling of contract violation. However, its realization is still scattered due to all necessary calls to it. The legibility and tangling become even worse in GOFP due to the error message code (including context information).

The GOFP implementation has a clearly defined interface, but this interface fail to say anything about the design constraints. Even if the interface in GOFP is an abstraction of the implementation, any change to a design constraint may propagate several other changes due to the tangled and scattered nature of such constraint. This also hinders our reusability and pluggability criteria.

On the other hand, the enhanced procedure implementation (EGOFP) fares better than the GOFP one. Since we use a strategy design pattern [9] to encapsulate the constraint and error reporting code, we have augmented the reuse of DbC constraints such as preconditions. The encapsulation of the constraint code also improved the code locality criterion. Finally, the use of the strategy pattern isolates code related to design constraints; this leads to a notion of interface that we do not have with plain GOFP. However, EGOFP implementation still have scattered and tangled calls to procedures that check constraints.

A common property of both GOFP and EGOFP is that they compromise our pluggability criterion, because the DbC concern removal is very invasive in a non-AOP way. Table 1 summarizes our analysis results.

4.2 The AOP Implementation

In the AOP code, each implementation exhibits better code locality (against non-AOP) resulting in a non-tangled DbC code. However, only the enhanced AOP implementations really exhibit improved code locality (see Table 1). This happens due to the reusability achieved for procedure calls (which check pre- and postconditions, and invariants). The non-enhanced AOP implementations only exhibit reusability (quantification) for invariant code. This compromises the overall reusability (in terms of pre- and

postconditions) and code locality (pre- and postconditions are still scattered in AOP code as procedure calls).

The enhanced AOP implementations offer better interfaces for the DbC concern than non-enhanced ones. The interfaces are now a more accurate reflection of the design constraints. For example, one can look at a particular XPI and reason about the effect of a design constraint in the entire system. This separation simplifies the reasoning during a change (our changeability criterion). Since the DbC code is well-localized, we just need to change the pointcut declaration of XPIs in order to reuse and apply a common constraint code, for example, to new added methods of an existing `Shape` class.

All AOP implementations satisfy the pluggability criterion, because the DbC code is completely localized as aspects and can be easily removed and composed when necessary. Only the implementations that consider metadata annotations can improve the fragility pointcut problem. In summary, our last implementation fares better in all modularity criteria.

4.3 Reasoning about Change

This section analyzes the implementations in terms of how well they fare when performing some change tasks (Table 1 summarizes how fare the changeability criteria for all the implementations). The selected change tasks combined affect all the design constraint concerns discussed in Section 2. This is useful to analyze the impact of the changes regarding the existing implementation of the design constraints.

Adding Color to Figure editor. The first change task (used in other work [23]) adds `Color` (new class) as an attribute, with getter and setter methods, in both `Point` and `Line` classes. The requirement is that added (`Color`) setter methods must fulfill the constraints imposed by the *NonNullInputParameters* concern. The added (`Color`) getter methods should in turn fulfill the constraints imposed by the *NonNullReturnTypes* concern. Finally, any method added in `Point` class must satisfy the invariant condition imposed by the *PointBoundsChecking* concern.

Adding a new Shape class. Our second change task adds a new Shape (`Square`) class in the figure editor system. This new added class has a set of new getter and setter methods in addition to the `moveBy` method (implemented through the `Shape` interface). The getter methods must satisfy the constraints imposed by the *NonNullReturnTypes* concern, the setter ones must satisfy the *NonNullInputParameters* concern, and the `moveBy` method must satisfy the *MoveByParametersChecking* concern.

In GOFP, the programmer must edit all the added operations to fulfill the design constraint concerns of these change tasks. These edits are related to the addition of procedure calls responsible for checking the design constraint and passing the context information useful for generating good error messages. Besides the changes of every added operation, this implementation also fails if we change or refine the existing design constraints. This again leads to changes on every operation related to the refined design constraints. This is a direct consequence of the lack of reuse previously discussed. In EGOFP, it also involves editing the same added operations (related to the

change tasks). However, these edits are related only to procedure calls since the design constraint code are, in fact, encapsulated in the strategy design pattern. As a result, EGOFP implementations fare better while maintaining existing design constraints.

Another important issue to consider is when the number of shape classes increases. This indicates that the number of edited places also goes up while using the GOFP and EGOFP implementation mechanisms. In sum, the GOFP and EGOFP implementations have limited benefits to satisfy our changeability criteria.

In the AOP-based, only the enhanced ones have exhibited better changeability. In relation to the non-enhanced ones, we have limited improvements because they have failed to properly deal with calls to procedures that check pre- and postconditions. Thus, for a new added operation that has pre- and postcondition constraints associated, we need to add two new advice with a call to a corresponding procedure. Without quantification, we tend to have the same problems occurred with GOFP and EGOFP implementations.

The use of quantification [7, 24] combined with a well-defined criteria to decompose the design constraints, led us to mitigate the limitations of the non-enhanced AOP implementations. This is the approach discussed in this work, which employs XPIs for modularizing DbC code. The main benefit of our approach is that, unlike the previous analyzed implementations, we always modify the same places while adding new shapes classes. This is an evidence that our approach is more scalable and has maintenance advantages than previous ones. Table 1 summarizes these findings. All the six implementations along with their implemented change tasks are available on the web [20].

5 Discussion

When a concern's implementation is not modularized, that is, the implementation is scattered across the program and tangled with the source code related to other concerns, the concern is said to be *crosscutting* [11]. As advocated, DbC [17] is an example of a concern in which its realization become crosscutting [11, 4, 16, 6] and that its implementation is better modularized by AOP.

However, as mentioned (in the Introduction Section), the work by Balzer, Eugster, and Meyer [1] contradicts this common belief. The authors conducted a study similar to the one we did here by using the same figure editor system [11, 12]. They argue that the use of AOP hinders the proper usage of DbC, since the former fails to emulate the latter. Their main complaints are: (i) aspects cannot deal with contract inheritance [17, 13], (ii) AOP breaks the documentation [17] property inherent of DbC, and (iii) the aspects appear separately from the base program.

In fact, our analysis confirms Balzer, Eugster, and Meyer's [1] findings. On the other hand, our work goes beyond theirs in the sense that we have also identified the main reasons why the literature efforts [11, 4, 16, 6] have failed to address the modularization problem and the main DbC principles. In this context, we discussed how best to use AOP mechanisms such as quantification [7, 24], pointcuts, and advice, in addition to their combination with metadata annotations [2, 3] which in turn makes the design more stable and less fragile [22]. These new findings confirm that common DbC objectives such as contract inheritance (complaint i) can be successfully implemented using AOP

(we demonstrated this regarding the implementation of the *MoveByParametersChecking* constraint which is common to all shape figures).

In relation to the claim that AOP hinders the documentation property (complaint ii) that is inherent in DbC, we argue that AOP languages such as AspectJ offer new mechanisms and possibilities that solve this problem. In particular, metadata annotations [2] can be used directly on the advised code, or in a crosscutting manner [3], to provide way to document the design constraints. Tools like AJDT already offer similar functionality that indicate which advice apply in a certain join point [12].

Regarding the last complaint, Where we document and instrument the contracts of libraries which source code is not available? This complaint goes against the runtime verification of APIs which we do not control source code. Hence, the previous works [11, 4, 16, 6] already showed how to properly separate/abstract the behavior (contracts) from its implementation details.

Another point to highlight is that by using XPIs [23], we explicitly make a well-defined interface that is responsible for introducing a short design phase that decouples the base and aspect designs. The application of well-defined interfaces help to decompose the DbC concern into small common design constraints and change the proper XPI interface whenever needed. Finally, we also demonstrated that by using annotations we can increase the design rules enforcement of XPIs, for example, that all exposed join points should exist in the advised (base) code.

5.1 Other forms of Aspectized DbC

As discussed throughout the paper, there are several works in the literature that arguments if favor or implementing DbC with AOP [11, 4, 16, 6]. Kiczales opened this avenue by showing a simple precondition constraint implementation in one of his first papers on AOP [11]. After that, several other authors explored how to implement and separate the DbC concern with AOP [4, 16, 6, 21]. All these works offer common templates and guidelines for DbC aspectization. However, as we have shown, only invariants have benefited from these original guidelines. We complement previous work in the sense of how to better separate and reason about contracts with aspects. One important issue to point out about these researched templates is that they have been shown to be very useful in the context of generative programming [21, 19].

5.2 Open Issues

A first open issue is to expand our concept of applying XPIs for DbC to investigate how it fares when used to modularize advanced concepts of design by contract [17], such as frame properties, information hiding with datagroups, history constraints, and abstract contracts (such as model fields and model methods used in JML [14, 15]). All these features are available in JML [14, 15] (an interface specification language for Java).

Another next important step is the large-scale validation. There are several works [11, 4, 16, 6, 21] in the literature that advocate the use of AOP to modularize DbC. Other work criticizes this use [1]. So a larger-scale validation seems necessary to more definitively settle this question. Furthermore, a predictive model for using aspects to implement design by contract will be useful to guide developers to recognize the situations

where it is advantageous to aspectize DbC code. Studies focusing on modular reasoning and comprehensibility denote another issue. Comparison with other advanced techniques (like JML [14, 15]) will be also useful to extend or refine this work.

6 Summary

Metadata annotations, pointcuts and advice are useful mechanisms for separating the design by contract concern in source code. To better understand and be able to work with these mechanisms, we proposed the use of well-defined interfaces, known as XPIs. Such interfaces are useful to decompose and reason about the design by contract code as recurrent design constraint concerns. We also combined this use of XPIs with metadata annotations, to improve the limited enforcement of XPI interfaces on advised code. Finally, we evaluated these mechanisms, in a small classical example, in terms of some modularity criteria and how they fared when performing change tasks.

The model proposed here provides a good basis for further research on design by contract implementation and modularization. We expect improvements to the model and guidelines to the combined use of annotations, pointcuts and advice.

Acknowledgements

We thank Eric Eide, Mario Südholt, Arndt Von Staa, David Lorenz and Mehmet Aksit for fruitful discussions (we had during the AOSD 2011) about the ideas developed in this paper and about design by contract modularization in general.

This work has been partially supported by CNPq under grant No. 314539/2009-3 for Ricardo Lima. Henrique Rebêlo is also supported by FACEPE under grant No. IBPG-1664-1.03/08. The work of Leavens was partially supported by a US National Science Foundation grant, CCF-10-17262.

References

1. Stephanie Balzer, Patrick Th. Eugster, and Bertrand Meyer. Can aspects implement contracts. In *In: Proceedings of RISE 2005 (Rapid Implementation of Engineering Techniques*, pages 13–15, September 2005.
2. Joshua Block. A metadata facility for the java programming language, 2004.
3. Jonas Boner. Aspectwerks. http://aspectwerkz.codehaus.org/.
4. Lionel C. Briand, W. J. Dzidek, and Yvan Labiche. Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 687–690, Washington, DC, USA, 2005. IEEE Computer Society.
5. Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, 2010. ACM.
6. Yishai A. Feldman, Ohad Barzilay, and Shmuel Tyszberowicz. Jose: Aspects for Design by Contract80-89. *sefm*, 0:80–89, 2006.
7. Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

8. Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
9. Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
10. Charles Antony R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
11. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.
12. Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, New York, NY, USA, 2005. ACM.
13. Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Zhiming Liu and He Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, November 2006. Springer-Verlag.
14. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.
15. Gary T. Leavens and Peter Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE, May 2007.
16. Marius Marin, Leon Moonen, and Arie van Deursen. A classification of crosscutting concerns. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 673–676, Washington, DC, USA, 2005. IEEE Computer Society.
17. Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
18. Alberto Costa Neto, Arthur Marques, Rohit Gheyi, Paulo Borba, and Fernando Castor. A design rule language for aspect-oriented programming. In *SBLP '09: Proceedings of the 2009 Brazilian Symposium on Programming Languages*, pages 131–144. Brazilian Computer Society, 2009.
19. Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Gary T. Leavens, Alexandre Mota, and César Oliveira. Optimizing generated aspect-oriented assertion checking code for jml using program transformations: An empirical study. *Sci. Comput. Program.*, 2010. Submitted for publication. Also available as a TR at: `http://www.eecs.ucf.edu/~leavens/tech-reports/UCF/CS-TR-10-01/TR.pdf`.
20. Henrique Rebêlo, Ricardo Lima, and Gary T. Leavens. Modular contracts with procedures, annotations, pointcuts and advice. Available from: `http://cin.ufpe.br/~hemr/sblp11`.
21. Henrique Rebêlo, Sérgio Soares, Ricardo Lima, Leopoldo Ferreira, and Márcio Cornélio. Implementing java modeling language contracts with aspectj. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 228–233, New York, NY, USA, 2008. ACM.
22. Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.
23. Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular aspect-oriented design with xpis. *ACM Trans. Softw. Eng. Methodol.*, 20:5:1–5:42, September 2010.
24. Marco Tulio Valente, Cesar Couto, Jaqueline Faria, and Sérgio Soares. On the benefits of quantification in aspectj systems. *J. Braz. Comp. Soc.*, 16(2):133–146, 2010.