The Contract Enforcement Aspect Pattern

Henrique Rebêlo[1] **Ricardo Lima**[1] **Uirá Kulesza**[2] **Roberta Coelho**[2]
Alexandre Mota[1] **Márcio Ribeiro**[1] **José Elias Araújo**[1]

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

[2]Department of Informatics and Applied Mathematics (DIMAp)
Federal University of Rio Grande do Norte
59072-970, Natal – RN – Brazil

{hemr,rmfl,acm,mmr3,jeqca}@cin.ufpe.br, {uira,roberta}@dimap.ufrn.br

Abstract. *The most fundamental motivation for employing contracts in the development of OO applications is to improve the reliability. Contract enforcement is a well-known established technique in object-oriented (OO) programming. However, the need to intercept well defined points in the execution of a program to check design constraints makes the enforcement of contracts a crosscutting concern. Thus, contract enforcement code is interwined with the business code, hindering maintenance. Moreover, because of the difficulty in separating contract enforcement code and business code, the former is often duplicated across several different places within a system. In this paper we present the Contract Enforcement Aspect pattern, which documents an aspect-oriented solution for the modularization of the contract concern. The use of this pattern minimizes code duplication as well as increases the reusability and maintainability of the contract concern and core (business) concern.*

Intent

The Contract Enforcement Aspect pattern leverages aspect-oriented programming (AOP) [Kiczales et al. 1997] techniques to modularize the contract concern, entirely decoupling the contract enforcement code from the business code. The pattern also aims to reduce duplication of contract code by making reuse easier within the same application, which in turn improves the system maintainability.

Problem

Design by Contract (DbC), also known as Contract Enforcement, was conceived by Meyer [Meyer 1992] as a means to ensure software quality, reliability, and reusability in object-oriented (OO) software development [Diotalevi 2004]. Contracts establish mutual obligations between program modules (e.g., classes) and their clients. The client must fulfill certain conditions before calling a method defined by a class (preconditions), as well as the class must respect certain properties that must hold after a method call (postconditions). In addition to method pre- and postconditions, class invariants also become part of contracts. Class invariants or simply invariants denote properties that must be fulfilled by every instance of a class; before and after any call to an accessible method. Contracts use assertions written in a Hoare-style, defining such pre- and postconditions [Hoare 1969]. Figure 1 illustrates the use of Java assertions to enforce class invariants.

```
public class StackAsArray {
   public Object [] array;
   private int index = 0;

   public void push(Object e) {
       assert (0 <= index)&& (index < array.length);
       this.array[index++] = e;
       assert (0 <= index)&& (index < array.length);
   }
   public void pop() {
       assert (0 <= index)&& (index < array.length);
       this.array[index-1] = null;
       this.index--;
       assert (0 <= index)&& (index < array.length);
   }
   public Object top() {
       assert (0 <= index)&& (index < array.length);
       Object result = this.array[index-1];
       assert (0 <= index)&& (index < array.length);
       return result;
   }
...
}
```

Contract concern

Figure 1. The class `StackAsArray` with scattered and tangled contract code.

Even though Java has assertions (the `assert` statement was added in version 1.4), there is no other built-in support for DbC [Meyer 1992]. Hence, using the `assert` statement to represent all DbC capabilities is not a straightforward solution. Moreover, putting pre- and post-conditions directly into source code has serious drawbacks in terms of code modularity, reusability, and maintainability [Kiczales et al. 2001, Diotalevi 2004].

In a conventional way, the implementation of contracts are scattered throughout the code and tangled with the business code. As a consequence, most systems have a considerable amount of duplicated contract enforcement code (as the invariant duplicated code illustrated in Figure 1). Thus, the contract code cannot be altered without changing the application code as well. This severely increases the maintenance effort. Because the invasiveness nature of DbC that crosscuts various modules in a system, it can be categorized as a crosscutting concern [Constantinides and Skotiniotis 2002, Marin et al. 2005, Marin 2006].

Figure 1 illustrates an example of scattered and tangled contract enforcement code. The class `StackAsArray` implements the conventional data structure Stack using Array. Such a class enforces an invariant condition denoted by `assert (0 <= index) && (index < array.length)`. It states that every method call made by an instance of the class `StackAsArray` must guarantee, both before and after the call, that the value of the field `index` is greater than or equal to zero and less than the length of the field `array`. If such an assertion does not hold, an `AssertionError` is raised signaling the assertion violation. Thus, the contract implementation (assertions), corresponding to this invariant condition, is placed at the beginning and end of each method declared in the class `StackAsArray`. (Preconditions and postconditions are omitted for simplicity as well as just the invariant condition is enough

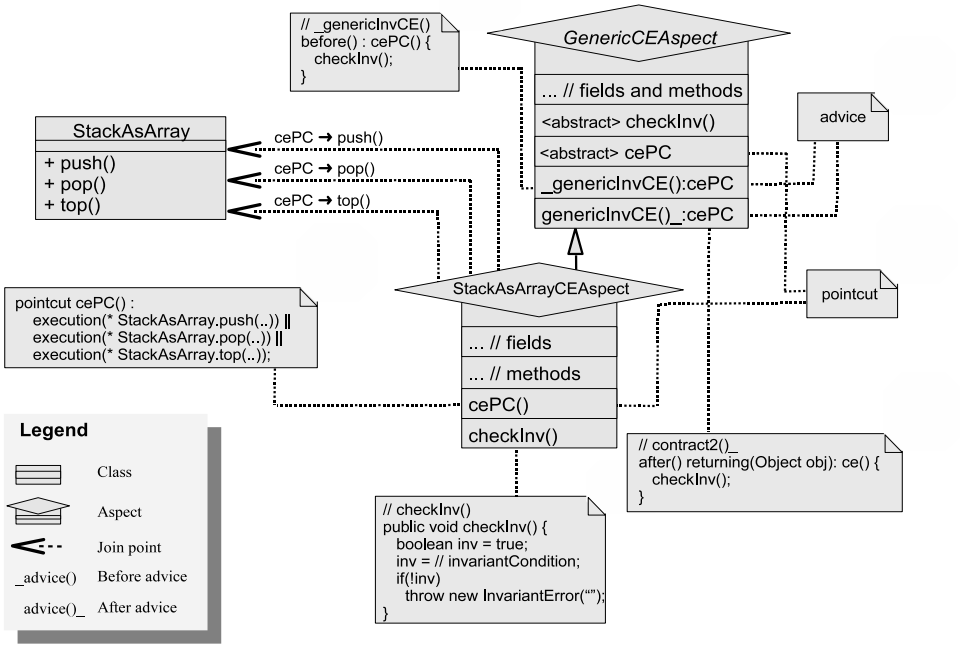

Figure 2. An example where the use of Contract Enforcement Aspect pattern avoids duplication, scattering, and tangling of contract code.

to show out the scattered and tangled nature of contract code.)

Solution

The Contract Enforcement Aspect pattern enables the explicit separation of contract code from business code. It uses aspect oriented programming (AOP) [Kiczales et al. 1997] features (in specific AspectJ [Kiczales et al. 2001]) to: (i) localize contract enforcement code within aspect units whose purpose is solely to implement it; (ii) reduce the amount of duplicated, scattered, and tangled contract enforcement code; (iii) improve the reuse of the contract code by other system units, modules etc, and (iv) reduce the maintenance effort of the contract enforcement code of the system.

The overall idea of the Contract Enforcement Aspect pattern is to use advice to implement contracts. We can use these "aspectized" contracts to affect several parts of a program by means of quantification and composition mechanisms provided by AOP languages.

Figure 2 illustrates how the pattern solves the problems discussed in the previous section. The design notation is based on an aspect-oriented modeling language, known as ASideML, which is used throughout this paper. This language extends UML with notations for representing aspects [Chavez and Lucena 2001].

The aspect `GenericCEAspect` defines the general structure to implement contracts in a modular way. As noted, such a generic aspect defines both a method (`checkInv`) and a pointcut (`cePC`) as abstract. Thus, they a supposed to be defined by a concrete aspect which extends it. Also, the generic aspect declares two advice (`before` and `after` advice) that crosscuts well defined points denoted by the abstract pointcut `cePC`. These advice are generic in the sense they are reused by the extending aspect. Hence, the con-

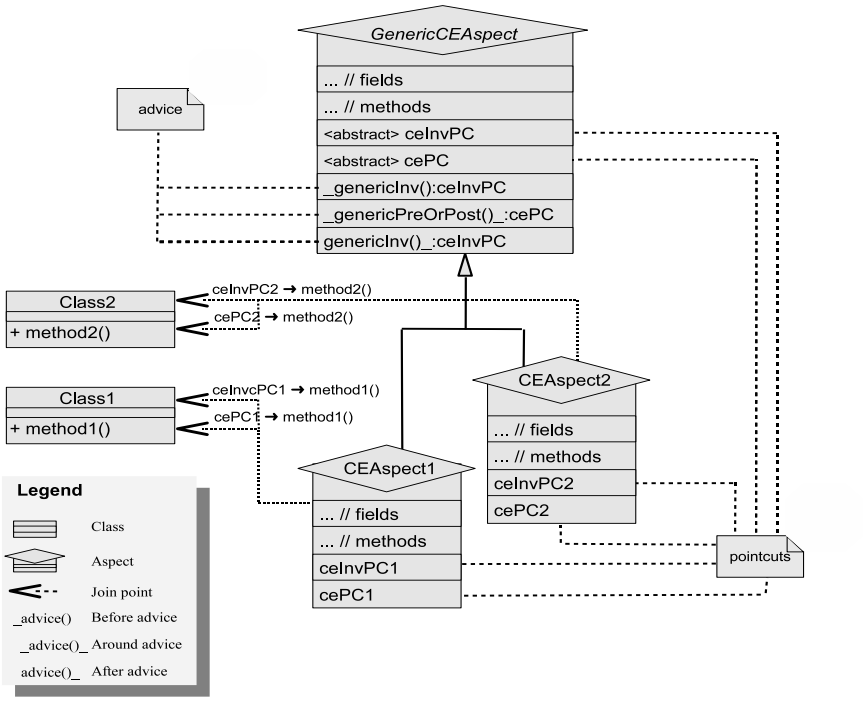

Figure 3. General structure of the Contract Enforcement pattern.

crete aspect `StackAsArrayCEAspect` is responsible for only provide the abstract members of its superaspect.

The pointcut `cePC` define the set of join points to be intercepted and the method `checkInv` provides the invariant's implementation. Once defined, the `cePC` and `checkInv`, the concrete aspect can now crosscuts the target class (which is denoted by `StackAsArray`) by injecting behavior (invariant checks) before and after the join points' execution. The methods `push`, `pop`, and `top` denote the set of join points which are affected by the concrete aspect `StackAsArrayCEAspect`.

In summary, code duplication, scattering, and tangling are avoided. The different contract enforcement mechanisms can be easily plugged/unplugged and reused by the business code of the system. Note that the example depicted in Figure 2 is a simplified version of the general structure of the pattern. We omitted pre- and postcondition templates for simplicity, since our concern is to solve the problem illustrated by the Figure 1. The complete structure of our pattern is discussed in the next section (Structure Section).

Structure

Figure 3 illustrates the structure of the Contract Enforcement Aspect pattern. The generic abstract aspect is denoted by the `GenericCEAspect`. Its structure declares a set of attributes and methods. The set of methods are those that must be implemented by concrete subaspects. Such methods comprehend the `checkInv` (illustrated in Figure 2), `checkPre`, and `checkPost`, which are responsible for checking the pre- and postconditions, respectively. Each method is called by its proper advice.

The aspects `CEAspect1` and `CEAspect2` are subaspects that extend the generic abstract aspect `GenericCEAspect`. The main role of these subaspects is to define two

abstract pointcuts (`ceInvPC` and `cePC`) from its superaspect. The former pointcut defines the set of join points in which the class invariants shoud be checked. Normally class invariants should be enforced by all methods of a class. Hence, to represent this quantification, the use of wildcards (‘`*`’) is usually adopted. The latter pointcut defines the set of join points in which pre- and postconditions should be enforced. Contrasting to the former one, the latter pointcut may not use wildcards since not all methods have pre- or postconditions to be verified. Finally, the classes `Class1` and `Class2` define one method each, `method1` and `method2`, respectively. These methods are intercepted by advice (e.g., `before` advice) of their corresponding concrete aspects in order to check contracts (e.g., preconditions).

In summary, The Contract Enforcement Aspect pattern has three participants:

- **Contract Enforcement Aspect**
  – defines the general contract enforcement structure with abstract pointcuts and generic advice.
- **Specific Contract Enforcement Subaspect**
  – implements the part of the contract concern that is specific to the business concern of a class.
- **Class**
  – implements the business code of an application, which has one or more methods that should be validated by contracts.

Dynamics

The following scenarios depict the dynamic behavior of the Contract Enforcement Aspect pattern.

Scenario I - Checking all Contracts, which is illustrated in Figure 4, presents the pattern behavior when the aspect `CEAspect` detects no contract violation:

- A client object calls the method `push` using an instance of class `Stack`.
- Before executing the method `push`, the control is transferred to the contract enforcement (CE) aspect `StackCEAspect`. The `StackCEAspect` attempts to verify the invariant conditions of class `Stack`, imposed on method `push`, using the CE advice `checkInv`. As with JML [Leavens 2006], we check all invariants before the preconditions and re-check them after postconditions.
- The CE advice ends its execution normally, without raising any errors (invariant violation).
- The control is again transferred to the CE aspect `StackCEAspect`, which now attempts to verify the preconditions of the method `push` using the CE advice `checkPre`.
- Control returns to the normal code, which resumes the execution of the method `push`.
- After executing the method `push`, the control is transferred to the CE aspect `StackCEAspect`, which attempts to check the postconditions of the method `push` using the CE advice `checkPost`.
- The control is again transferred to the CE aspect, which attempts to re-check the invariants conditions of class `Stack`, imposed on the method `push`, using the CE advice `checkInv`.

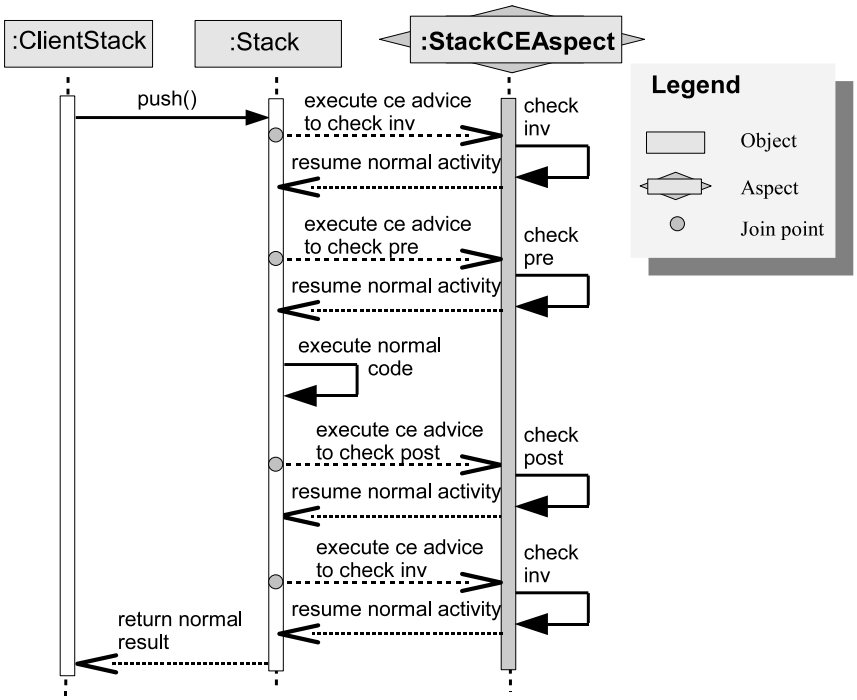

Figure 4. A scenario where no contract violation occurs.

Scenario II - Raising a precondition error, which is illustrated in Figure 5, presents the pattern behavior when the aspect `CEAspect` detects a precondition violation:

- A client object calls the method `push` using an instance of class `Stack`.
- Before executing the method `push`, the control is transferred to the contract enforcement (CE) aspect `StackCEAspect`, which attempts to verify the invariant conditions of class `Stack`, imposed on the method `push`, using the CE advice `checkInv`.
- The CE advice ends its execution normally, without raising any errors (invariant violation).
- The control is again transferred to the CE aspect `StackCEAspect`, which now attempts to verify the preconditions of method `push` by using the CE advice `checkPre`.
- CE advice `checkPre` raises a precondition error.
- Error `pre` is signaled to the instance of `Stack`, re-signaled by the latter, and finally received by the client object.

The Scenario II illustrated in Figure 5 can also be considered to describe the behavior when the aspect `CEAspect` detects an invariant violation before method's execution. As an invariant can be thought as an implicit pre- and postcondition of a method, the same flow of the sequence diagram presented in Figure 5 can be used. A similar behavior is provided if we have an invariant violation after method's execution. In this case, the violation only occurs after the execution of the constrained method.

Consequences

The Contract Enforcement Aspect pattern exhibits the following benefits:

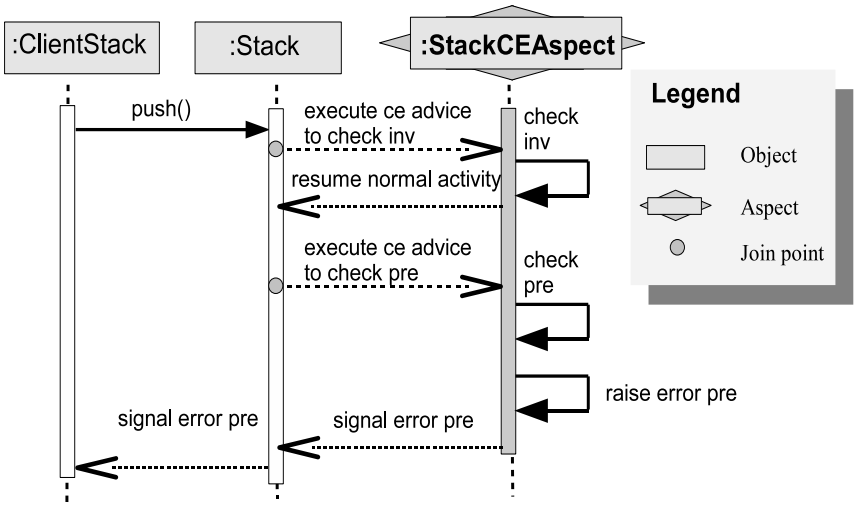

Figure 5. A scenario where a precondition violation occurs.

- *Improved Separation of Concerns*. The contract concern is entirely modularized in the aspects. The basic classes do not implement any contract behavior and do not need to be changed (changes in the conventional contract enforcement strategy is required). The aspect-oriented constructs support the separate definition of contract behavior that affect several units of the system. This separation of concerns allows better modularity, avoiding tangled code and scattered code over those units.
- *Reusability*. The basic contract code is modularized in a generic contract enforcement aspect, which can be reused by concrete aspects.
- *Reduced Number of Advice*. The idea to modularize and check contracts with aspects is not new. However, existing approaches [Diotalevi 2004, Briand et al. 2005, Rebêlo et al. 2008b] do not use a generic aspect which severely reduces the number of advice responsible for inserting the contract enforcement behavior at the join points. The generic advice reduces the number of advice because the `around` advice declared to check pre- and postconditions is reused for each method of a particular type. On the other hand, the other approaches [Diotalevi 2004, Briand et al. 2005, Rebêlo et al. 2008b] that do not consider a generic aspect (with a generic advice) must need at least to write an `around` advice per method to check pre- and postcondition.
- *Reduction of duplicated contract enforcement code*. The pattern supports the isolation of the contract enforcement as aspects, minimizing the code replication. Class invariants are an example of code duplication that can appear into several units (Figure 1 illustrates an example of duplicated code caused by invariant checks). The use of aspects to modularize contracts mitigate this problem.
- *Ease of Evolution*. Contract Enforcement (CE) developers need only to augment the existing pointcuts to include new join points. Additionally, whether necessary, the contract methods (e.g., `checkPre`) can also be augmented to support new contract checks related to the new join points. The well known problem of pointcut fragility can be a concern by a CE aspect only in the case where a rename refactoring does not take into account the renamed join point. Adding or removing a particular join point is not a problem because the impact is only in the invariant

checking level. Since, an invariant checking is a global property for a particular type, it does not matter if we add or remove a join point.

- *Maintainability*. The improved separation of concerns provided by the contract pattern also contributes to increase system maintainability. Since contract code is localized, a developer does not have to search through an entire program to change a certain contract checking code.
- *Pluggability*. A Contract enforcement (CE) aspect can be easily replaced by another CE aspect implementing different contract verification strategies. This feature makes it easy to reuse the normal code of an application or part of it across different systems. Moreover, since the DbC technique is commonly used during the development phase, it can be easily dropped in the production code due to the plug-and-play capability added by the use of aspects.

However, this pattern solution has the following drawbacks:

- *Increased Size and Complexity in Contract Checking Methods*. Unless the methods that check invariants (i.e., `checkInv`), the others (e.g., `checkPre`) have a significant increase in their sizes when there are several methods with pre- and postconditions to be enforced (checked). This is due to the generic nature of such methods. For example, the contract checking method `checkPre` is used to verify the preconditions of all methods. The more methods with preconditions, the more lines of code the method `checkPre` will have. In addition, the contract checking method such as `checkPre` can have several `if` statements do handle various join points that a class can have. Hence, the more methods with preconditions, the more `if` statements the method will have. In the implementation section, we illustrate these drawbacks with source code samples.
- *No Integration with Constrained Environments*. There are other patterns related to the Contract Enforcement Aspect pattern, for instance [Rebêlo et al. 2008b] (refer to Related Patterns section). The main difference is that the work reported in [Rebêlo et al. 2008b] provides the use of Aspect constructs, compliant to both Java SE and Java ME applications. Since our pattern depends on special variables of AspectJ [Kiczales et al. 2001, Laddad 2003], such as `thisJoinPoint`, we cannot apply it in constrained environments like Java ME. This is due to the need for using reflection, which is not supported by Java ME for various reasons (e.g., performance degradation).
- *The Aspect dependency*. The use and knowledge of aspects are mandatory to apply the contract enforcement pattern. Aspects provides more idioms than that covered by a traditional OO language. Such idioms are responsible for providing a clean well-modularized implementations of crosscutting concerns (contract enforcement in our case). We recommend that programmers read the AspectJ [Kiczales et al. 2001, Laddad 2003] tutorials and examples available in the literature before deciding to adopt AspectJ and consequently the CE aspect pattern into a project. The use of AspectJ [Kiczales et al. 2001, Laddad 2003] is required since the proposed pattern is used to modularize the crosscutting contract enforcement concern of the Java applications. However, we claim that the solution is general enough (independent of AspectJ) and can be adapted to other languages that also have corresponding aspect-oriented extensions.

```
public void meth() throws ArbitraryException {
  try{
    int r = -10;
    constrainedMethodCall(r); // throws a precondition error
  }
  finally{
    throw new ArbitraryException(); // masks the previous precondition error thrown
  }
}
```

Figure 6. An overridden assertion violation by exception handling code.

- *Limited Integration with Error Handling Code*. A subtle interaction between error handling code and contract checker can cause the latter to fail to report errors. This problem is due to the well-known capability of finally clauses to implicitly override exceptions. Another problem is when a catch block hides a thrown contract error. Besides the pattern we describe in this work, these problems also occur in other approaches that use or not aspect-orientation to check contracts.

Figure 6 illustrates the forth drawback discussed (limited integration with error handling code). Let us assume that the method `constrainedMethodCall` has a precondition which states that its argument must be greater than zero. Hence, when we execute the method `meth` and it calls the method `constrainedMethodCall` passing `-10` as argument, we have a precondition violation. However, since the body of the `finally` clause is always executed, we have the precondition violation overridden by other exception. This happens due to the exception thrown (`ArbitraryException`) in the body of the `finally` clause. Thus, Figure 6 depicts a classical example where contract enforcement can fail to report violations in the presence of exception handling code. For a complete discussion about the interaction between error handling code and contract checking, please refer to [Huisman 2009].

Implementation

In what follows we describe some guidelines for implementing the Contract Enforcement Aspect pattern. We give AspectJ [Laddad 2003] code fragments to illustrate a possible implementation of the pattern, describing details of the Stack example.

```
1  public abstract aspect GenericCEAspect {
2     public interface TargetType {}
3     declare parents : TargetType implements java.io.Serializable;
4
5     protected abstract pointcut withinType();
6     protected abstract pointcut targetInvariantPC();
7     protected abstract pointcut targetPC();
8     private pointcut invariantPC() : targetInvariantPC() && withinType();
9
10    protected abstract void checkInv(Object thisObject);
11    protected abstract void checkPre(Object thisObject, String sig, Object[] args);
12    protected abstract void checkPost(Object oldThisObject, Object thisObject, String sig,
13     Object[] args, Object returnValue);
14
15    before():invariantPC(){
16       checkInv(thisJoinPoint.getTarget());
17    }
18
19    Object around():targetPC() && withinType(){
20       Object result = null;
21       checkPre(thisJoinPoint.getTarget(), ...);
22       Object oldThisObject = DeepCopy.copy(thisJoinPoint.getTarget());
23       result = proceed();
24       checkPost(oldThisObject, ...);
25       return result;
26    }
27
28    after() returning(Object result):invariantPC(){
29       checkInv(thisJoinPoint.getTarget());
30    }
31 }
```

Figure 7. The abstract aspect `GenericCEAspect`.

Step 1: How to define a Contract Enforcement aspect?

↪ The Contract Enforcement (CE) aspect (Figure 7) is declared as abstract (line 1) since it needs to be redefined in different subaspects. Some abstract pointcuts and methods are declared in the CE aspect to be implemented by its subaspects. For instance:

- `withinType()` pointcut (line `5`) – is used to define the specific type which is affected by the generic aspect;
- `targetInvariantPC()` pointcut (line `6`) – is used to define the set of join points (of a specific type denoted by the abstract pointcut `withinType`) that the invariant conditions are enforced;
- `targetPC()` pointcut (line `7`) – identifies the set of join points (of the type denoted by the abstract pointcut `withinType`) that contain pre- and/or postconditions to be enforced;
- `checkInv()` method (line `10`) – is used to define the invariant checking code to be executed in the set of join points defined by the abstract pointcut `targetInvariantPC`;
- `checkPre()` method (line `11`) – is used to define the precondition checking code to be executed in the set of join points defined by the abstract pointcut `targetPC`;
- `checkPost()` method (line `12`) – is used to define the postcondition checking code to be executed in the set of join points defined by the abstract pointcut `targetPC`.

The Contract aspect uses the *Marker Interface* [Hanenberg and Unland 2003] AspectJ idiom in its implementation (line 2). This idiom is used to define a generic interface to be attached to a type (e.g., class or interface) (denoted by `TargetType` in Figure 7). The main advantage of using a *Marker Interface* is that no internal knowledge is required except the name of the *Marker interface* (`TargetType`). Thus, this idiom is useful when, for example, everything in the pointcut is defined except the classes where such crosscutting contract enforcement occurs. In addition, to the *Marker Interface*, the contract aspect successfully applies the *Abstract Pointcut*, *Composite Pointcut*, and *Template Advice*. The *Abstract Pointcut* idiom (lines 6 and 7) increases the reusability of pointcuts to other aspects, which are not restricted to a certain type (e.g., `TargetType`). This happens because the whole pointcut definition is moved to the concrete aspect. The *Composite Pointcute* idiom (line 8) is used to compose a complete pointcut in terms of other component pointcuts which are independent and reusable. This alows one to modify a single component pointcut without knowing the complete (composite) pointcut. Finally, the *Template Advice* (lines 10 to 30) idiom is very similar to the *Template Method* [Gamma et al. 1995]. By using *Template Advice* one should only implement the abstract methods (lines 10 to 12) inherited from the abstract superaspect. This is useful to specify variabilities (which are implemented by the concrete methods in subaspects) invoked inside a template advice. Such an advice is defined within the abstract aspect (`GenericCEAspect`) and is inherited by subaspects with no need to re-define it.

In DbC languages (e.g., JML [Leavens 2006]) and approaches, one can refer to old expressions in postconditions. An old expression refers to an expression before method's execution (known as pre-state). In order to allow postconditions to refer to expressions evaluated on the entry of a method (pre-state), we use an around advice (lines 18 to 25) to properly refer to old values. Hence, The methods `checkPre` and `checkPost` are used within the around advice. The pre-state constitutes the beginning of the around advice just as before the call to the method `proceed` (line 22). Such a call denotes the call to the original method. In turn, the post-state comprehends any point after the call to the method `proceed` (lines 23 and 24). As a consequence, we keep a copy of the object's states in the variable `oldThisObject` (line 21). As noticed, this variable is assigned in the pre-state. Once the pre-state values are saved, we can refer to them in the post-state. In this case, we pass the variable `oldThisObject` as an argument to the method postcondition (`checkPost`). Note that all the advice defined within the generic abstract aspect do not need to be implemented by the concrete subaspects. Hence, we gain in reusability. □

Step 2: How to define a specific Contract Enforcement aspect?

↪ Contract subaspects define specific implementations of the Contract Enforcement aspect. We can specify a different contract subaspect to each type (class or interface) in type system. The subaspects must implement the abstract pointcuts and methods of the Contract aspect. To exemplify, in the stack examples these elements are implemented in the `StackAsArrayCEAspect` (Figure 8) aspect, as follows:

- `withinType()` pointcut (line 4) – defines the type `StackAsArray` that is intercepted;
- `targetInvariantPC()` pointcut (line 5) – defines the interception of all non-static methods of the class `StackAsArray`, as possible execution points where

```
1  public privileged aspect StackAsArrayCEAspect extends GenericCEAspect {
2     declare parents : StackAsArray implements TargetType;
3
4     protected pointcut withinType() : within(StackAsArray);
5     protected pointcut targetInvariantPC() : execution(!static * StackAsArray.*(..));
6     private pointcut ConstructorExec() : execution(public StackAsArray.new(int));
7     private pointcut pushExec() : execution(public void StackAsArray.push(Object));
8     private pointcut popExec() : execution(public void StackAsArray.pop());
9     private pointcut topExec() : execution(public Object StackAsArray.top());
10    private pointcut isEmptyExec() : execution(public boolean StackAsArray.isEmpty());
11    private pointcut lengthExec() : execution(public int StackAsArray.length());
12
13    protected pointcut targetPC() : ConstructorExec() || pushExec() || popExec()
14     || topExec() || isEmptyExec() || lengthExec();
15
16    protected void checkInv(Object thisObject){
17       boolean inv = true;
18       StackAsArray stack = (StackAsArray) thisObject;
19       assert (0 <= index)&& (index < array.length);
20    }
21    protected void checkPre(Object thisObject, String sig, Object[] args){
22       boolean pre = true;
23         ...
24       assert (pre);
25    }
26    protected void checkPost(Object oldThisObject, Object thisObject, ...){
27       boolean post = true;
28       ...
29       assert (post);
30    }
```

Invariant checking code

Figure 8. The concrete aspect `StackAsArrayCEAspect`.

the invariants are enforced;

- `targetPC()` pointcut (line `13`) – defines the interception of the specific methods (lines `6` to `11`) `push`, `pop`, `top`, `isEmpty`, `length`, and the constructor `StackAsArray`, as possible execution points where the pre- and/or postconditions are enforced;
- `checkInv()` method (lines `16` to `23`) – defines the implementation of the invariant checking code for the join points specified in the pointcut `targetInvariantPC`;
- `checkPre()` method (lines `24` to `30`) – defines the implementation of the precondition checking code for the join points specified in the pointcut `targetPC`;
- `checkPost()` method (lines `31` to `37`) – defines the implementation of the postcondition checking code for the join points specified in the pointcut `targetPC`.

The Contract Enforcement subaspects must also specify which types implement the interface `TargetType` (step 1). The aspect `StackAsArrayCEAspect` defines that the class `StackAsArray` implements the interface `TargetType`. It specifies a `declare parents` AspectJ construction as presented in Figure 8 (line `2`).

Note that the line `19` of Figure 8 represents the code which enforces the invariant conditions in the join points specified by the pointcut `targetInvariantPC`. All that duplicated, scattered, and tangled invariant checking code illustrated by Figure 1 is now modularized in a single place of the method `checkInv` (line `19`). □

Known Uses

Lipert and Lopes [Lippert and Lopes 2000] were the first work in the literature to show that this pattern can be used to modularize contracts. Even though their work focus on exception detection and handling, they also discussed how AOP was used to handle pre- and postconditions that are interwined and scattered in a conventional way (using OOP). The authors conducted a study that restructured a Java-based object-oriented framework for interactive business applications, called JWAM [Breitling et al. 2000]. According to the authors, such a framework uses contracts (pre- and postconditions) to ensure that callers do not misuse the methods, and that the methods' implementations preserve some of their basic specifications. Due to the crosscutting nature of such contracts, the authors successfully extracted the contract code from classes into separate aspects. As a result, the authors decreased the 2120 preconditions (without aspects) of the JWAM framework to 620 preconditions (with aspects).

Kiczales [Kiczales et al. 2001] discusses and demonstrates how to implement this pattern in a modular form. Briand *et al*. [Briand et al. 2005] and Diotalevi [Diotalevi 2004] show how to use the Design by Contract Aspect Pattern. The authors considered the modularization of pre-, postconditions and invariants into aspects. The solution allows one to write contracts of the application separately (untangled) from one's business logic. They also discuss how the use of aspects provides transparency, better reusability, and flexibility.

Leavens *et al*. [Leavens 2006] conceived a DbC language known as Java Modeling Language (JML). JML constructs are written in a special notation like Java comments. As JML is a superset of Java with DbC capability, it provides a more expressive way (e.g., behavioral subtyping [Liskov and Wing 1994]) to write contract enforcement concern of Java applications than by using simple Java `assert` statements [1]. The use of JML, as a DbC tool, minimizes the symptoms of code tangling and scattering of Java applications. However, since the nature of JML specifications still are crosscutting, the use of the contract aspect pattern provides a better solution to separate the contract concern from the base code, and therefore gaining in attributes such as maintainability. Moreover, as AspectJ, a programmer must present knowledge in how to use the language and tools in order to specify DbC properties (e.g., preconditions) on the Java code. In this sense, the adoption of AspectJ is more straightforward since the programmers can use a plugin that can be added in the well-know Eclipse IDE. JML still lacks support to a common environment that can be used to apply all the JML tools.

Rebêlo *et al*. used this pattern to implement the JML features [Leavens 2006] in a new JML compiler, known as ajmlc [Rebêlo et al. 2008b, Rebêlo et al. 2008a]. Such a compiler generates aspects that are suitable to enforce preconditions, postconditions, and invariants. Also, the ajmlc compiler generates, unlike the standard JML compiler (jmlc) [Cheon and Leavens 2002], an instrumented bytecode compliant with both Java SE and Java ME applications. The authors also empirically analyzed the impact of the pattern in some applications.

[1] Programming with assertions – `http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`

Related Patterns

The Contract Enforcement Aspect pattern is strongly related to the Design by Contract (DbC) pattern, proposed by Rebêlo *et al*. [Rebêlo et al. 2008b, Rebêlo et al. 2008a]. The main difference between DbC and Contract Enforcement Aspect is that, in the latter, the use of an abstract generic aspect reduces significantly the number of advice used to check the contracts.

The Contract Introduction pattern [Laddad 2003] uses aspect-oriented programming to introduce implementations of contract methods (e.g., checkPre) in a transparent way to the base code of the system. The Contract Introduction uses Contract Enforcement Aspect to combine the introduced contract methods with advice (which refer to these contract methods) in order to intercept the well points in the program to check contracts in a modular form.

Several authors [Lippert and Lopes 2000, Filho et al. 2006, Filho et al. 2007] propose the use of AOP to modularize exception handling into aspects. For instance, the Error Handling Aspect pattern [Filho et al. 2007] aims at using AOP for separating exception handling code from the business (core) code. This pattern can be used in combination with Contract Enforcement Aspect. As a result, both error detection and error handling code become localized within aspects. The resulting code (of the application of both patterns) is not tangled by error detection (part of the contract concern) and error handling concern. Such an Error Handling Aspect pattern can define a unique point for handling contract violations raised during the execution of a system. Once a contract is violated the result is unpredictable and thefore can not guaranteed. Hence, such a pattern can be used to handle all the situations when a contract is violated.

The AspectJ [Laddad 2003] implementation of the Contract Enforcement Aspect pattern uses advanced AspectJ idioms, such as *Marker Interface*, *Abstract Pointcut*, *Composite Pointcut*, *Template Advice* [Hanenberg and Unland 2003]. Such idioms facilitate the separate definition of the base system and aspects. The way in which aspect-oriented features are applied has a direct impact on how these idioms are reusable in other object-oriented programs. In the context of Contract Enforcement Aspect, the used idioms make the pattern successfully reusable in other object-oriented programs.

Acknowledgements

We would like to give special thanks to Tiago Massoni, our shepherd, for his comments, helping us to improve our pattern.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 is also supported by FINEP and Cenpes/Petrobras.

References

Breitling, H., Lilienthal, C., Lippert, M., and Züllighoven, H. (2000). The JWAM Framework: Inspired by research, reality-tested by commercial utilization. In *OOPSLA 2000 Workshop: Methods and Tools for Object-Oriented Framework Development and Specialization*.

Briand, L. C., Dzidek, W. J., and Labiche, Y. (2005). 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. IEEE Computer Society.

Chavez, C. and Lucena, C. (2001). Design-level support for aspect-oriented software development. In *OOPSLA 2001 Workshop: Proc. of the Workshop on Advanced Separation of Concerns at OOPSLA'2001*.

Cheon, Y. and Leavens, G. T. (2002). A runtime assertion checker for the Java Modeling Language (JML). In Arabnia, H. R. and Mun, Y., editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press.

Constantinides, C. and Skotiniotis, T. (2002). Reasoning about a Classification of Cross-cutting Concerns in Object-Oriented Systems. In *Second Workshop on Aspect-Oriented Software Development (Workshop Aspektorientierte Softwareentwicklung der GI-Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung)*, Bonn, Germany.

Diotalevi, F. (2004). Contract enforcement with AOP: Apply Design by Contract to Java software development with AspectJ. Avaliable at http://www.ibm.com/developerworks/library/j-ceaop.

Filho, F. C., Cacho, N., Figueiredo, E., Maranh ao, R., Garcia, A., and Rubira, C. M. F. (2006). Exceptions and aspects: the devil is in the details. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–162, New York, NY, USA. ACM.

Filho, F. C., Garcia, A., and Rubira, C. M. F. (2007). The error handling aspect pattern. In *SugarLoafPLop '10: Proceedings of the 6th Latin American Conference on Pattern Languages of Programming (SugarLoafPLop'10)*, pages 22–45.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Hanenberg, S. and Unland, R. (2003). Aspectj idioms for aspect-oriented software construction. In *Proc. of the 8th European Conference on Pattern Languages of Programming and Computing (EuroPlop'03)*.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Huisman, M. (2009). On the interplay between the semantics of java's finally clauses and the jml run-time checker. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, pages 8:1–8:6, New York, NY, USA. ACM.

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

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Jean-MarcLoingtier, and Irwin, J. (1997). Aspect-oriented programming. In *European Conference on Object-*

Oriented Programming (ECOOP) , Jyvskyl, Finland, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag.

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

Leavens, G. T. (2006). JML's rich, inherited specifications for behavioral subtypes. In Liu, Z. and Jifeng, H., 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. Springer-Verlag.

Lippert, M. and Lopes, C. V. (2000). A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA. ACM.

Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16:1811–1841.

Marin, M. (2006). Formalizing typical crosscutting concerns. *CoRR*, abs/cs/0606125.

Marin, M., Moonen, L., and van Deursen, A. (2005). A classification of crosscutting concerns. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 673–676, Washington, DC, USA. IEEE Computer Society.

Meyer, B. (1992). Applying "design by contract". *Computer*, 25(10):40–51.

Rebêlo, H., Soares, S., Lima, R., Borba, P., and Cornélio, M. (2008a). JML and aspects: The beneifts of instrumenting JML features with AspectJ. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 11–18, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362. School of EECS, UCF.

Rebêlo, H., Soares, S., Lima, R., Ferreira, L., and Cornélio, M. (2008b). 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. ACM.

Appendix A – Aspect Terminology

In this appendix, we provide a brief overview of the terminology associated with aspect-oriented programming. We draw on the terminology described by Laddad [Laddad 2003]. Such a terminology is adopted by many aspect-oriented programming languages, such as AspectJ [Kiczales et al. 2001, Laddad 2003], a general purpose aspect-oriented extension to Java.

Aspects. Aspects are modular units responsible for modularizing crosscutting concerns. An aspect can affect one or more types (classes and interfaces) and/or objects in distinct manners.

Join Points and Pointcuts. Join points are well-defined points in the execution of a program (e.g., method calls and method executions).

Advice. Advice are related to dynamic crosscutting mechanism of AspectJ. An aspect can specify an *advice* that is used to add behavior when a join point is reached. Such a behavior is provided by extra code defined by the advice. There are different kinds of

advice: (i) before advice executes whenever a specific join point is reached and before the original computation proceeds; (ii) after advice executes whenever a specific join point is reached and after the original finishes; (iii) around advice run whenever a specific join point is reached, and this advice has total control under the intercepted join point.

Inter-Type Declarations. Inter-type declarations are a static crosscutting mechanism that allows one to introduce new methods and fields to an existing class, convert checked exceptions into unchecked exceptions, and changes the class hierarchy.

Weaving. Weaving is a process responsible for composing classes and aspects. It can be performed either in compile-time or in runtime.

Appendix B – Online Appendix

We invite researchers to replicate our example used in this paper. Source code of the Stack example with and without the pattern and the generic Contract Enforcement Aspect code are available at:
`http://www.cin.ufpe.br/~hemr/sugarloafplop10`.