Using Aspect Oriented Programming to Implement Design by Contract: A Stack Example
Albert Noble

ABSTRACT
Design by Contract is a programming methodology that has been applied to Object-Oriented Programming to increase reliability. A contract is an agreement between a class and its clients. A contract is enforced on a class though postconditions, preconditions, and class invariants that is usually in the same functional part of the code as the class definition or implementation. This paper focuses on separating the contract and the class using Aspect Oriented Programming. Aspect-Oriented Programming (AOP) introduces separation of concerns that would allow a contract and class to exist as separate entities. Additionally, AOP supports constructs that can be used to implement preconditions and postconditions. Separating the contract from the implementation code will be demonstrated through the example of a stack.

1. INTRODUCTION
Design by Contract is a concept that is applied to object-oriented programming intending to increase the reliability of software. Bertrand Meyer pioneered this concept. Eiffel was his programming language of choice to support it. Design by Contract is “the relationship between a class and its clients as a formal agreement, expressing each party’s rights and obligations” [4]. The formal agreement is applied to the class programmatically.

In Object-Oriented programming, an object architect may ask a developer to add implementation code into a class; the class contains no code other than the definition of the class itself, called a skeleton class. Once the developer begins the implementation, the object architect cannot enforce constraints on the implementation other than code reviews and testing; i.e. these constraints are not programmatic. Design by Contract allows an object architect to add a “contract” to the skeleton code. This ensures that the implementation code does what it is supposed to do. The contract is part of the class definition and all children of the class will inherit the contract. Thus the agreement between the client and class is always guaranteed whether the client accesses the base class or the derived class.

Aspect-oriented programming (AOP) is a new programming methodology that enhances object-oriented programming (OOP). Besides the concepts of encapsulation, polymorphism, and inheritance from OOP, AOP introduces the concept of crosscutting-concerns. Concerns can be thought of as properties or areas of interest in a system [10]. For example, logging is a common concern throughout a system. In object-oriented programming (OOP), a concern that is located in multiple code blocks is considered a crosscutting-concern [11]. AOP allows one concern to span multiple code blocks. In other words, instead of repeating the same code block throughout the code base, AOP allows a developer to create one code block that is used in multiple sections of the code.

Contracts can be considered crosscutting-concerns [1]. They can be crosscutting, since the same concern, a contract, may be located in multiple code blocks. The result is a contract and class definition in separate pieces of code. This separation could achieve better reusability and adaptability [5]. In this paper, it will be demonstrated how AOP can be used to enforce Design by Contract using a common Design by Contract example: the stack.

2. APPROACH AND METHODOLOGY
This research will focus on implementing Design by Contract using aspect-oriented programming (AOP). To prove that Design by Contract can be implemented in AOP, it will be demonstrated that a contract implemented by traditional Design by Contract is functionally equivalent to the contract that has been implemented in AOP. By functionally equivalent, both contracts will need to implement the same post-conditions, pre-conditions, and invariants. The specific stack and contract that will be used is based on the STACK4 example from [4], as it contains class invariants, preconditions, and postconditions.

The equivalence will be done by ensuring that both classes contain and implement each precondition, postcondition, and class invariant from the STACK4 example. This will be performed in three stages.

1. Assign an ordinal number to each Design by Contract element in the STACK4 example (precondition, postcondition, and class invariant). Each element that is assigned a number is a specific assertion or statement that enforces the contract.

2. For each element identified in step 1, the same assertion will be applied to the contract for AOP stack and to the pure OOP Design by Contract stack; each element will reference the ordinal number in step 1. This will be used as a cross reference for step 3.

3. Cross reference each item that was identified in step 1 to the each stack that was created in step 2. This is a check to ensure that all of the elements on the contract in the STACK4 example have were fully implemented

The equivalence is important to ensure that both implementations are using the same contract.

Currently, there are no programming languages that have a native AOP implementation. Most AOP implementations have been open-source solutions directed at specific programming languages. Since there are no well-known implementations of AOP for Eiffel, a well-known implementation for Java called AspectJ will be used.
The selection of the stack to use is from [4] because it was written by the creator of Design by Contract, thus it is straight from the source. The STACK4 example is created as a generic class, however, since Java is being used, it does not currently support generic types. Thus the generic STACK4 will be constrained to a single type, an integer.

Table 1 outlines to constraints that were applied to this paper.

<table>
<thead>
<tr>
<th>Type of Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Tool</td>
<td>Eclipse</td>
</tr>
<tr>
<td>AOP Language</td>
<td>AspectJ</td>
</tr>
<tr>
<td>Java SDK</td>
<td>J2SE v1.4.2_06 SDK</td>
</tr>
<tr>
<td>Java Runtime Environment</td>
<td>J2SE v1.4.2_06 JRE</td>
</tr>
<tr>
<td>Stack Example</td>
<td>STACK4 example outlined in [4], specific to an integer stack</td>
</tr>
</tbody>
</table>

3. TRADITIONAL DESIGN BY CONTRACT

In Design by Contract, the contract is expressed in the code as runtime enforcement of assertions. The assertions in Design by Contract are expressed in three types: preconditions, postconditions, and class invariants. Preconditions and postconditions are described on the properties of individual routines whereas the class invariants enforce the integrity constraints characterizing a class [4].

3.1 Terminology

It is necessary to provide terminology of preconditions, postconditions, and class invariants in order to apply them to both Java and AOP.

A precondition is a constraint applied to a method before any calls to it, this includes any call to the method that occurs either from within the class or outside the class. The precondition ensures that both the class and method will run without errors [4]. It checks that the class is in a proper state, as well as enforcing that the client of the class is using the method properly. If the precondition fails, an assertion is thrown.

A postcondition is a constraint applied after a method has been executed. It ensures that the code within the method will execute properly and return the class to a proper state. It is a guarantee from the implementer that the method will perform as required. A postcondition will only execute if a precondition is valid. If the postcondition fails, an assertion is thrown.

A class invariant is a constraint applied to the entire class. The class invariant ensures that the class is in valid state. Class level variables are often checked in an invariant. All methods must satisfy the class invariant. For the invariant to be satisfied one of two conditions must hold true [4]:
- After instance construction
- before and after every remote call to a method of the class.

A class invariant constraint occurs before any precondition and after any postcondition.

3.2 Eiffel STACK4 Example

The Design by Contract stack used the STACK4 (figure 1) example from [4]. This stack is the primary example in [4] that is used to demonstrate Design by Contract, as it implements all necessary components: preconditions, postconditions, and class invariants. Table 2 has three columns associated to the right of the source code. These three columns represent the precondition count, the post condition count, and the invariant count. Next to each precondition, postcondition, or class invariant, is a reference number. Each number is of the following format:

X#, where X is the type of assertion and # is the count

‘R’ represents preconditions, ‘E’ represents postconditions and ‘I’ represent class invariants. The summed total from each of the three columns represents the total number of contract assertions.

Table 2. Snippet of Eiffel Stack References

<table>
<thead>
<tr>
<th>Eiffel Stack4</th>
<th>Precondition Ref #</th>
<th>Postcondition Ref #</th>
<th>Invariant Ref #</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>class STACK2 [G] creation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>make</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>feature -- Element change</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>put (x: G) is</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>require</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not_full: not full</td>
<td>R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>do</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count := count + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>representationput (count, x)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ensure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not_empty: not empty</td>
<td>E6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>added_to_top: item = x</td>
<td>E7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>one_more_item: count = old count + 1</td>
<td>E8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>in_top_array_entry: representation @ count = x</td>
<td>E9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>End</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>invariant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count_non_negative: 0 &lt;= count</td>
<td>I1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count_bounded: count &lt;= capacity</td>
<td>I2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>consistent_with_array_size: capacity = representation.capacity</td>
<td>I3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty_if_no_elements: empty = (count = 0)</td>
<td>I4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>item_at_top: (count &gt; 0) implies (representation.item (count) = item)</td>
<td>I5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. ASPECT ORIENTED DESIGN BY CONTRACT

The basic foundation of Aspect Oriented Programming is crosscutting-concerns. A concern that is located in multiple code blocks is considered a crosscutting-concern [11]. With AOP, these crosscutting-concerns are represented in a new programming element called an aspect. Instead of having one concern in multiple code blocks, the aspect can represent all these concerns in a single code block.

Through the use of the aspect, a contract can be implemented. Once the aspect has been created, the contract and class will be separate entities residing in different pieces of code. This is different than Design by Contract because the contract and class are usually in the same functional part of the code [5]. For example, if the contract needed to be removed, the AOP contract can be removed easily by removing the aspect from compilation. For Design by Contract, the code will need to be modified to remove the contract. The separation of code and contract is the defining element when AOP is used to implement Design by Contract.

4.1 Terminology

The Aspect contains three main elements, a join point, a pointcut, and an advice. A join point defines where the aspect will take place such as a method. A set of join points is called a pointcut. An advice is executed for each pointcut. The advice is code that can be executed before, around, or after a pointcut; the advice “advises” how the pointcut should act. The properties of an advice in AOP are similar to the preconditions, postconditions, and class invariants in Design by Contract.

To create the separation between the class and the contract, the individual parts of the class that the contact applies to will need to be identified. For the purposes of this exercise, every individual part is a method. In order to identify each method in AOP, a pointcut is created. The identification occurs in the pointcut through the method name and special wildcard symbols. A pointcut only defines a cross cutting concern, it does not perform any logic.

An advice can perform logic on a pointcut. There are three types of advices that can be applied to a pointcut: the before advice, the after advice, and the around advice. As the name suggests, the before advice executes before a pointcut and the after advice executes after a pointcut. The around advice allows execution to happen before a pointcut, after a pointcut, or skip the pointcut execution completely. Thus if a pointcut were a method, the around advice can perform precondition and class invariant checking, execute the pointcut (the method), and finally perform postcondition and class invariant checking. Each around advice is also required to declare a return type. The return type usually matches the same return type of the join point that is being advised [9].

A group of pointcuts and advices are defined by an aspect. All crosscutting concerns are implemented in the aspect. A privileged aspect allows an aspect access to private members of classes they are crosscutting. To help implement Design by Contract, the privileged aspect can access private variables that are needed for contract enforcement.

4.1.1 Advice Selection

The advice that most closely resemble Design by Contract first appears to be the before and after advice because the terminology is similar to the precondition and postcondition. However, using the before and after advice combination poses a drawback since the old keyword for postconditions in Eiffel cannot be implemented cleanly. Implementing cleanly is allowing the old variable to be applicable only to the method that needs it; i.e. no global or aspect member variables would be used. The old keyword in Eiffel is the value of an expression on the entry of a method [4]; this value is essentially saved off before the method operates on the expression. Another problem with the before and after advice is that neither one can capture the return value of a method.

To simulate the old keyword, the around advice will be used. First, an expression or variable that the old keyword will be applied to is identified. Next, that variable or expression will have the value saved before the execution of the pointcut or method. Then the pointcut, a.k.a. method, is executed; in AspectJ the proceed keyword in an around advice executes the pointcut. Finally, once the pointcut has finished execution, that value would be accessed in the postcondition for appropriate contract checking.

The around advice also allows the return value of the pointcut to be verified in the postcondition. The return value of the pointcut can be captured from the proceed keyword.

4.1.2 Accessing Private Variables & Method Arguments

In order to access private variables of a class, the aspect must be declared as privileged. The advice will need to access the private variables for postcondition, precondition, and invariant assertions. To access the private variables of the class in the advice, context information about the object will need to be exposed to the advice. The execution object pointcut captures the object for the method being called. The target pointcut is used to capture the object that passes the context information to the advice. The advice receives the object information similar to method arguments.

Similar to private variables, the advice needs to access method arguments for preconditions and postconditions. The argument pointcut captures method arguments. The context information about the argument is exposed to the advice through the args keyword. The advice receives argument values as advice arguments.
4.2 Creation of the AOP Stack
An aspect is merely a cross cutting concern; it cannot be instantiated. The aspect needs to be applied to a concrete concern. In this case, the concrete concern is a stack class. The stack class is similar to the STACK4 class but without the assert statements; the class name is \textit{IntStack}. This is a class that contains no error checking and only applies to a stack of integers.

Once the stack class has been created, the cross cutting concerns can be defined. The creation of the AOP stack contract, named \textit{StackAspect}, first began with the pointcut definitions. Each pointcut definition has a one to one correlation to each method in the \textit{IntStack} class. The created pointcuts only apply to the \textit{IntStack} class, thus there is a tight coupling between the \textit{StackAspect} class and the \textit{IntStack} class.

After each pointcuts was created, an around advice was applied to that pointcut. Each around advice, except the \textit{make} routine, followed the same basic format:
1. Check class invariants
2. Check preconditions if applicable
3. Run the method defined by the pointcut
4. Check postconditions if applicable
5. Check class invariants

The create routine \textit{make} is a special case where all the above are executed except for first step. This case follows the class invariant rules as defined in section 3.1.

4.3 AspectJ STACK4 example
The AOP implementation is in Java using AspectJ; the implementation is shown in table 3. Similar to STACK4 example in table 2, there are three columns that identify preconditions, postconditions, and class invariants. For each assertion in the AspectJ stack, there is a corresponding reference number in the appropriate column. This reference number coincides with the same reference number from the STACK4 example from table 2.

<table>
<thead>
<tr>
<th>X-Ref</th>
<th>Precondition</th>
<th>Postcondition</th>
<th>Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert !stack.full();</td>
<td>R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int _oldCount = stack.count; proceed(stack, x);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assert !stack.empty();</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assert stack.item() == x; assert stack.count == (_oldCount + 1); assert stack.representation[stack.count-1] == x; CheckInvariants(stack);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>void around(IntStack stack, int x) : putPoints(stack, x) { CheckInvariants(stack); }</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. AspectJ Contract for the Stack

5. RESULTS
5.1 Differences
There were some language dependent differences with respect to the overall functionality. In the Design by Contract STACK4, the \textit{make} routine is equivalent to a constructor in Java. In order to keep things simple, the \textit{IntStack} did not use the constructor to implement the make routine, but rather a separate method. This allowed both construction routines to follow the same naming convention.

The most notable difference between the AOP implementation and the Design by Contract implementation is that the AOP implementation has a separation between the contract and the class. For Design by Contract implementation, the stack implementation code and contract are in one unit of code, the STACK4 example. For the AOP implementation, the stack implementation code is in the \textit{IntStack} class and the contract is represented in the \textit{StackAspect} aspect. As opposed to having one class implement both the contract and functional code, the AOP implementation contains a separation between contract and functional code. This is a positive difference that could help lead to the reliability [7, 6], reusability [3, 5], and adaptability [3, 5, 6].
For the AOP implementation, class invariants were applied to every method in the `IntStack` class through the `StackAspect`. These invariants are added in each around advice, however they are needed to be applied manually. Thus, for the AOP implementation, whoever is implementing the contract will need to manually code each invariant because they are not applied automatically. Class invariants in the Design by Contract implementation on the other hand are applied to every method in class automatically. In Eiffel, the compiler guarantees that the class invariants are applied to these methods automatically. Manually applying class invariants in the AOP implementation has the obvious drawbacks as there is no “contract” to guarantee that new methods will include the proper class invariant code checking.

The AOP implementation had more lines of code than of the Design by Contract implementation. The additional statements were not required in the Design by Contract implementation. For a method in the class, the following were required to be added to the AOP Contract in addition to the postcondition and precondition:

- Create a pointcut
- Create an advice
- Call class invariants at the beginning of the method
- If the post condition requires the use of the `old` keyword, save the value off into a local variable in the advice
- Call the proceed method to execute the method in the class
- Call class invariants at the end of the method
- If the original method returns a value, the advice must return a value

Thus there were between 4-7 lines of additional code for to implement Design by contract in each method in AOP. The count was calculated as follows:

- Excluding assert statements, white space, and line with curly brackets.
- Including pointcut definitions, advice definition, and statements.

The number of lines of code is only a difference; it should not be assumed that this is an advantage, disadvantage, or neutral disparity.

5.2 Similarities

Functionally, both contract implementations had the same assertions. Each assertion in the AOP implementation referenced the same assertion the Design by Contract implementation. The references were checked via the reference numbers in the Design by Contract stack. By checking the reference numbers, it ensured that each precondition, postcondition, and class invariant was implemented.

With both the Design by Contract and the AOP implementation, all the subclasses will inherit the contract. However, as noted in the differences, any new methods in the AOP implementation will not inherit the invariant assertions. The contract for the AOP stack will only apply to any overridden methods in the AOP Stack.

5.3 Breaking Encapsulation

The Design by Contract `STACK4` example uses private variables to check preconditions, postconditions, and class invariants. In order to ensure that a method’s or classes (invariant) contract is not violated, these private variables are used. Some examples are `E9`, `I3`, and `I5`.

In order to implement the same contract as outlined in Design by Contract `STACK4`, the AOP contract needs to access these variables. However, accessing these variables does break encapsulation. The AOP contract does access the private members of the `IntStack`. This creates a tight coupling between the contract and class.

Although, AOP allows encapsulation to be broken, it should not be construed as a negative aspect if it used correctly [9]. In order to adequately implement all postconditions and class invariants according to the `STACK4` example, this was necessary.

6. RELATED WORK

There has been some work on using AOP to implement Design by Contract, however, most of these implementations did not implement all Design by Contract assertions. [3] defines the rules for checking a contract in an aspect with the implementation of the assertions defined in another class. Thus, for a contract to be applied to a class, there are three parts: the aspect, the contract class, and the class itself. This provides a separation of class and contract; however, private variables are often used for contract checking. Private variables are used to ensure that the variables are of the correct state [4]. The use of a separate class to implement the contract will not allow access to private variables. Accessing the private variables would always guarantee that the contract is checking the correct variables.

[2] separates the contract and the class completely; however it focuses only on preconditions and postconditions. This is incomplete as class invariants are not addressed. It cannot be implied that class invariants apply to every precondition, postcondition, or method, since the rules for invariants as outlined in section 3.1 do not apply in every case.

[5, 8, 7] make the implication that AOP can be used to implement Design by Contract, however, each examine it at an abstract level. For these papers, implementing Design by Contract through AOP is not examined in depth.

7. CONCLUSION AND FUTURE WORK

In traditional Design by Contract, the contract and class definition are often in the same functional piece of code. Aspect Oriented Programming can provide an approach to separate functional code such as the class definition and implementation from the contract. This paper addressed implementing Design by Contract through Aspect Oriented Programming through a specific example: the stack.

When a specific contract is used on a specific class, it was demonstrated that AOP can be used to implement Design by Contract. This separation has been viewed as a positive characteristic. From a pure loose coupling perspective, this is valid. However, the question remains if there are any negative
differences. Further, if there are differences, do they outweigh the positive characteristics of separation of contract and class.

Some differences that could not be determined neither positive nor negative to this example were:

- the number of lines of code increase using the AOP contract
- the OOP encapsulation being broken and tight coupling of private variables to the AOP contract
- the adaptability of the AOP contract and how the contract applies to a class when new methods are added, what happens to class invariants.
- A better implementation to apply class invariants

Before the AOP implementation of Design by Contract can be deemed as a suitable alternative for the traditional Design by Contract, both the positives and negative characteristics will need to be further examined.

8. REFERENCES


