Your Java code often needs to interact with external components when you develop enterprise software. Whether your application must communicate with a legacy application, an external business system, or a third-party library, using components you don't control introduces the risk of unexpected results. IBM IT Specialist Filippo Diotalevi shows how aspect-oriented programming (AOP) can mitigate this risk by helping you design and define clear contracts between components while keeping your code clean and flexible.

**Design by Contract (DBC)** is a technique in object-oriented software design whose goal is to ensure software quality, reliability, and reusability. The key concept in DBC is that you can achieve this goal by:

- Specifying the communication between components as accurately as possible
- Defining the mutual obligations and expected results of the communication process

These mutual obligations are called **contracts**, and you use **assertions** to check whether an application complies with a contract. Simply put, an assertion is a boolean expression, inserted at a specific point in a program's execution, that must be true. Failure of an assertion is typically a symptom of a bug in the software, so it must be reported to the user.

DBC is especially valuable when you're dealing with external components or libraries and need to ensure that the data your application passes to them and receives from them is correct. This article will show you an abstract infrastructure that employs aspect-oriented programming (AOP) to implement DBC and then a sample application that establishes a contract with an external component.

**Assertions and the Java language**

DBC identifies three basic types of assertions:

- **Pre-conditions**: Obligations that the client must fulfill in order to call the external component correctly.
• **Post-conditions**: Expected results after the execution of the external component.
• **Invariants**: Conditions that must remain unchanged after the execution of the external component.

The Java language didn't provide native support for assertions originally. The `assert` statement was added in version 1.4. However, applying DBC in your everyday coding can be a challenge. In fact, the most common approach -- putting pre- and post-condition assertions directly into application code -- has serious drawbacks in terms of code modularity and reusability. This approach is a vivid example of *tangled code*: it mixes business-logic code with the nonfunctional code that assertions require. The code is inflexible because you can't change or remove assertions without altering the application code.

**The champion of DBC**

Bertrand Meyer's book *Object Oriented Software Construction* (see Resources) describes and formalizes Design by Contract. Meyer urges the adoption of DBC as a central and systematic approach to writing robust and reliable object-oriented code. He invented the Eiffel programming language, which provides excellent support for assertions. "Design by Contract" is a trademark of Interactive Software Engineering, the company Meyer co-founded to develop and distribute Eiffel.

An ideal solution to this problem would meet four requirements:

• **Transparency**: The pre- and post-condition code isn't mixed with business logic.
• **Reusability**: Most parts of the solution are reusable.
• **Flexibility**: Assertion modules can be added, deleted, and modified in a simple fashion.
• **Simplicity**: Assertions can be specified using a simple syntax.

**Using AOP for transparent Design by Contract**

When you're aiming for separation of concerns, transparency, and flexibility, aspect-oriented programming (AOP) is often the right answer. Pre-conditions, post-conditions, and invariants are *crosscutting concerns* -- commonly used functionalities often included in an application's various modules and, to some degree, mixed with application code. AOP's goal is to allow developers to code these functionalities in separate modules and apply them in a flexible and declarative way.

This article assumes that you are generally familiar with AOP under AspectJ and is not intended as an introduction to AOP. See the Resources section for a listing of introductory articles on this topic.

**The implementation infrastructure**

A solution that satisfies the four requirements I listed earlier consists of three parts, as diagrammed in Figure 1:

• The application code (which contains no elements relating to DBC)
• A contract implementation (with pre-condition, post-condition, and invariants checks)
• An object that acts as a *bridge* between the code and the contract, capable of applying the contract in the right part of the code and with the correct logic
The design illustrated in Figure 1 guarantees a highly flexible solution that lets you apply and remove contracts without changing either the contract implementation or the application code. And it's totally transparent to the application.

### Implementation details

The contract is a Java class that implements a specific interface, and the "bridge" is an AspectJ aspect. This aspect specifies the exact point where the contract is to be applied and the control logic necessary to apply the contract, shown in Figure 2.

The logic diagrammed in Figure 2 is common to all contracts, so you can develop a common abstract aspect that specifies it. Figure 3 shows a class and aspect diagram of the solution:
Figure 3. Basic components of a contract-checker system

The AbstractContract aspect specifies the control logic declared in the activity diagram in Figure 2. It leaves unexpressed (that is, abstract) the point in the program’s execution where the contract -- defined by an implementation of the ContractManager interface -- is applied. A ConcreteContract aspect (extending the AbstractContract aspect) is responsible for indicating:

- The exact point of execution of the contract checking, through the targetPointcut pointcut
- The class responsible for checking the contract, through the getContractManager() method

The class containing the code necessary for checking the contract between the application and the external module is an implementor of the ContractManager interface. ContractManager, shown in Listing 1, is a simple Java interface defining the basic behavior of a contract-checker class:

Listing 1. ContractManager interface

```java
public interface ContractManager {
    /**
     * Check the preconditions
     */
    public void checkPreConditions(Object thisObject, Object[] args)
        throws ContractBrokeException;

    /**
     * Check the postconditions
     */
    public void checkPostConditions(Object thisObject, Object returnValue, Object[] args)
        throws ContractBrokeException;

    /**
     * Check the invariants
     */
    public void checkInvariants(Object thisObject) throws ContractBrokeException;
}
```

The ContractManager interface defines a different method for each kind of assertion you want to check. Each method has access, through the thisObject parameter, to the Java object that is calling the function whose contract is to be ensured. The pre-condition and post-condition methods can also see the values passed as function arguments (args). Only the post-condition method
receives the eventual return value, through the `returnValue` parameter. By using these three methods in combination, you can check almost all common conditions.

The `AbstractContract` aspect performs the control logic necessary for executing the contract checking. This logic is expressed in the `around(): targetPointcut()` advice. Listing 2 shows the `AbstractContract` aspect:

**Listing 2. AbstractContract aspect**

```java
public abstract aspect AbstractContract
{
    /**
     * Define the pointcut to apply the contract checking
     * MUST CONTAIN A METHOD CALL
     */
    public abstract pointcut targetPointcut();

    /**
     * Define the ContractManager interface implementor to be used
     */
    public abstract ContractManager getContractManager();

    /**
     * Perform the logic necessary to perform contract checking
     */
    Object around(): targetPointcut()
    {
        ContractManager cManager = getContractManager();
        System.out.println("Checking contract using:" +
                            cManager.getClass().getName());

        if (cManager!=null)
        {
            System.out.println("Performing initial invariants check");
            cManager.checkInvariants(thisJoinPoint.getTarget());
        }
        if (cManager!=null)
        {
            System.out.println("Performing pre-conditions check");
            cManager.checkPreConditions(thisJoinPoint.getTarget(), thisJoinPoint.getArgs());
        }
        Object obj = proceed();
        if (cManager!=null)
        {
            System.out.println("Performing post conditions check");
            cManager.checkPostConditions(thisJoinPoint.getTarget(), obj, thisJoinPoint.getArgs());
        }
        if (cManager!=null)
        {
            System.out.println("Performing final invariants check");
            cManager.checkInvariants(thisJoinPoint.getTarget());
        }
    return obj;
    }
}
```

The `AbstractContract` aspect presents two abstract methods that must be overridden when you implement a concrete contract-checker aspect:
• public abstract pointcut targetPointcut() expresses the pointcut where the advice must be applied. The pointcut must be a method call.
• public abstract ContractManager getContractManager() must return an instance of a ContractManager implementing the correct contract checking.

It is important to note that the invariant checking is performed twice, both before and after the service execution. This lets you check that the execution of the service doesn't affect the value of some external fields.

The failure of a contract causes a ContractBrokeException, which stops the execution of the advice.

**Contract checking in practice**

Now that you understand the necessary infrastructure for implementing Design by Contract with AOP, you can put it to work. Suppose you need to query an external customer relationship management (CRM) system to retrieve customer's data. Your invocation of the CRM system might look like this:

```java
Customer cus = companyCustomerSystem.getCustomer("Pluto");
```

From the developer point of view, the implementation of the getCustomer function is insignificant, because getCustomer is an external component. But it is extremely important to check that it doesn't return corrupt results. It's equally important to be sure that your application doesn't pass wrong or meaningless input to the CRM system. You can address both contingencies by developing a concrete aspect that extends AbstractContract. The concrete aspect overrides two methods:

• targetPointcut(), to define the pointcut where to apply the contract checking
• getContractManager(), to define the ContractManager implementation responsible for executing all checks

Listing 3 shows the concrete aspect for the sample application:

**Listing 3. Concrete contract aspect**

```java
public aspect CcCompanySystem extends AbstractContract {
    public pointcut targetPointcut(): call(Customer CompanySystem.getCustomer(String));
    public ContractManager getContractManager()
    {
        return new CompanySystemContractManager();
    }
}
```

The CcCompanySystem aspect specifies that the contract checker called CompanySystemContractManager will be invoked on the pointcut represented by the invocation of the CompanySystem class's getCustomer method. You don't need to define the control logic of the
Contract enforcement with AOP

The contract-checking operation, because it's inherited from the ancestor AbstractContract abstract aspect in Listing 2.

The last step is to develop a Java class to do the contract checking. As I said earlier, this class must implement the ContractManager interface. Listing 4 shows a sample CompanySystemContractManager class:

Listing 4. ContractManager implementation for the sample application

```java
public class CompanySystemContractManager implements ContractManager {
    /**
     * Check preconditions
     */
    public void checkPreConditions(Object thisObject, Object[] args)
            throws ContractBrokeException
    {
        Object arg = args[0];
        if (arg == null)
        {
            throw new ContractBrokeException("PRECONDITION ERROR: " +
                    "Argument of getCustomer shouldn't be null");
        }
    }
    /**
     * Check postconditions
     */
    public void checkPostConditions(Object thisObject, Object value, Object[] args)
            throws ContractBrokeException
    {
        if (value == null)
        {
            throw new ContractBrokeException("POSTCONDITION ERROR: " +
                    "Return value of getCustomer shouldn't be null");
        }
    }
    /**
     * Check invariants
     */
    public void checkInvariants(Object thisObject) throws ContractBrokeException
    {
        //invariants check
    }
}
```

The CompanySystemContractManager class in Listing 4 checks only if arguments or return values are null, but it's possible to enhance it to add extremely sophisticated checks.

An important point to note: Every contract check instantiates one CompanySystemContractManager object, so you can check invariants by storing data in private fields during the first invariant check and verifying that they aren't changed after the execution of the CRM system invocation.

Congratulations! You've developed a simple contract between your application and the CRM system. After you compile the application using the AspectJ compiler, the contract will be applied to every invocation of the CompanySystem class's getCustomer method and will check the consistency of your application's interactions with it. Furthermore, if the
CompanySystemContractManager is sufficiently generalized, you can reuse it; you need to redefine only the targetPointcut to apply it to other contract checking.

The sample solution completely fulfills the four desiderata I listed at the beginning of this article:

- It is **transparent** because the business-logic code contains no reference to contract checking; it is absolutely unaware of it.
- It is **reusable** because it relies on a simple infrastructure (an interface and an abstract aspect) and lets you reuse a single ContractManager in more than one situation.
- It is **flexible** because you can use the AspectJ compiler facilities to choose which aspect to apply and therefore which contracts you want to check.
- It is **simple** because it is made up of only a few classes.

**Conclusion**

This article has described a possible approach for adopting Design by Contract in Java application development using AspectJ and AOP techniques. The proposed solution guarantees a clean and flexible solution, because it relies on an extremely simple and well-modularized design that lets you code contracts separately from your business logic and apply them declaratively.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-ceaop-source.zip</td>
<td>25KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2004  
 Trademarks  
(www.ibm.com/developerworks/ibm/trademarks/)