Design by Contract in .NET Using Aspect Oriented Programming

Kai Köhne, Wolfgang Schult, Andreas Polze
Hasso-Plattner-Institut
Universität Potsdam
Postfach 900460
14440 Potsdam, Germany
(kai.koehne|wolfgang.schult|andreas.polze)@hpi.uni-potsdam.de

ABSTRACT

Most software is being implemented using imperative programming techniques. However, for certain problem domains declarative code has proven to be more expressive, easier to understand and more compact than its imperative counterpart.

Aspect Oriented Programming (AOP) seems to be a promising approach for intermixing imperative program code with declarative aspect information using an aspect weaver tool. The *dynamic Rapier LOOM.NET* weaver recognizes metadata in the form of .NET attributes, thus allowing the mixture of imperative and declarative programming styles using AOP techniques.

We discuss in this paper the declarative Design by Contract architectural concept, which is based on assertions, e.g. invariants, pre- and postconditions. We have developed an aspect representation of the concept's assertions in the form of a set of metadata for Microsoft's .NET languages. Our approach has been applied to a large real-world application for which we give an evaluation.

The paper aims to demonstrate the applicability of AOP techniques for describing the semantics of object-oriented interfaces based on the Design by Contract concept.

Keywords

Aspect Oriented Programming, Declarative Programming, Design by Contract, .NET

1. INTRODUCTION

Predictability, as an attribute of a component to consistently perform according to its specification, is a key concern in today's software development. As software systems become more and more complex, it is essential to guarantee the predictability of every subsystem component in order to ensure the predictability of the complete system. This is even more important when components are supposed to be reused.

Software components are commonly specified in terms of the interfaces they provide and the interfaces they require. Therefore, the first step to ensure predictability of a software component is to define its interfaces as precise as possible. While for instance the popular object oriented modeling language UML[1] provides support for extended interface descriptions [18, Object Constraint Language], most object oriented languages have only a very limited expressiveness concerning the definition of interfaces. Usually, advanced conditions for the proper use of components (or for assumptions of the component about its environment) either have to be written as a comment or need to be ensured in the specific implementation of the software component.

Design by Contract is a commonly known paradigm to extend the interface of a software component with boolean expressions. It helps to easily specify interface definitions and ensures compliance to the interface by both the using as well as the providing sides. In this paper we present a way to provide support for Design by Contract in .NET using Aspect Oriented Programming (AOP).

2. DESIGN BY CONTRACT

Design by Contract as a clearly defined and so-named concept was first provided as a part of the programming language Eiffel developed by Bertrand Meyer. The notion of the paradigm is that objects are providing services to other objects, and that the conditions for the use of this service, as well as the effects of its use, should be part of a formal contract between the service provider and the service consumer. Design by Contract extends the limited expressiveness of interfaces in object oriented languages. An example of an interface definition in Eiffel is shown in listing 1. The keywords `require` and `ensure` are used to mark pre- and postconditions, `invariant` identifies invariants. The semantics behind these conditions is given in the next section.

[1]Product and brand names used in this document may be trademarks or registered trademarks of their respective owners. Any such trademarks or registered trademarks are the sole property of their respective owners.

```
class interface DICTIONARY [ELEMENT]

feature
put (x: ELEMENT; key: STRING) is
  -- Insert x so that it will be retrievable
  -- through key.
  require
    not_full: count <= capacity
    key_valid: not key.empty
  ensure
    item_avail: has (x)
    item_stored: item (key) = x
    size_incr: count = old count + 1

  -- ... Interface specifications of other
  -- features ...

invariant
  not_under_minimum: 0 <= count
  not_over_maximum: count <= capacity

end
```

Listing 1: Definition of a contract in Eiffel

2.1 Eiffel Assertions

Pre- and postconditions as well as invariants are in Eiffel defined as different types of assertions. An assertion is here a boolean expression that specifies a condition of the contract. If the assertion proves to be true the condition is satisfied. If it evaluates to be false the contract is broken and the normal flow of control is interrupted, e.g. by raising an exception. The point where an assertion is tested in the program flow depends on its type.

Invariants are assertions that have to be satisfied during the entire object's lifetime. These conditions are evaluated at the beginning and at the end of each method call to the object. They thereby ensure that the object is in a valid state. *During* a method call the condition might temporarily be violated because this temporary state will not be noticed by the client object.

In contrast to invariants, pre- and postconditions are always assigned to a specific method. Preconditions are assertions a client has to ensure before calling the method of the object implementing the contract. They can assure that a method is only called when the object is in a certain state or the method parameters are valid. Postconditions are assertions the object has to ensure after the method was called. By using postconditions, the called object can promise a certain state after the method was executed.

2.2 Inheritance of Eiffel Assertions

Design by Contract is an extension to the standard concepts for interface definitions in object oriented languages. It thus has to follow the common rules of interface inheritance. If a contract is inherited, the assertions are also inherited and must be fulfilled by both the object implementing the new interface and the object using it.

It is also possible to extend a pre- or postcondition in a derived interface or class. However, the new assertion must be more restrictive for postconditions, or less restrictive for preconditions.

2.3 Realizing Design by Contract for .NET using AOP

There are many attempts to support for Design by Contract in object oriented languages (some are described in section 6). All have to address two main problems: How to write down assertions conveniently, and how to check them during runtime.

This paper shows that Design by Contract is a first-class aspect, and should therefore be implemented using AOP. This solves the problem of code instrumentation using standard technology. In addition to that, .NET attributes are a powerful way to extend the source language with customized metadata. By using them to write Eiffel assertions these assertions can become an integral part of the interface descriptions.

3. THE RAPIER LOOM.NET WEAVER

This section gives a short overview about Rapier LOOM.NET, the aspect weaver we used for our experiments. The aspect weaver arises from the LOOM.NET project [22, 23]. Rapier LOOM.NET is a dynamic aspect weaver and provides its functionality through an assembly which has to be linked to the .NET project.

3.1 Aspects in Rapier LOOM.NET

In Rapier LOOM.NET an aspect is simply defined through an *aspect class*. An aspect class is a special .NET class with methods constructors and fields as well. At defined *connection points* an aspect class becomes interwoven with a target class. Interweaving, strictly speaking, means that an *aspect method* will be interwoven with a *target class method*. The aspect method itself contains the aspect code and is defined within the aspect class. It has a special connection point attribute applied. This connection point attribute declares a method in the aspect class as an aspect method. Not necessarily every method in an aspect class is a aspect method and has this attribute applied. Methods without this attribute will not considered for the weaving process.

Beside the connection point attributes, Rapier LOOM.NET defines nearly a dozen of interweaving attributes. These attributes are used to describe which methods should become interwoven with the aspect method. Examples for these attributes are *Include*, *Exclude* and *IncludeAll*.

A target class is a regular .NET class. The one and only restriction is that target class methods (which should become interwoven) either have to be virtual or to be defined via an interface. The weaving process will be initiated during runtime with a factory. Instead of using the *new* operator one uses the weavers factory method to produce interwoven objects. Figure 1 depicts this in detail.

3.2 Rapier LOOM.NET vs. AspectJ

Table 1 shows a comparison between AspectJ and Rapier LOOM.NET. As described above, Rapier LOOM.NET is a .NET library which is simply linked to the .NET project. This means that there is no need for a special compiler or an extra tool support. The interweaving process happens

	AspectJ	**Rapier LOOM.NET**
Interweaving	at compile time	at run time
aspect weaver	*ajc* compiler, used instead of the *javac* compiler	*RapierLoom* assembly, which is linked to the .NET project
aspect definition	`aspect MyAspect { ... }`	`public class MyAspect:Aspect { ... }`

Definition of interweaving points	With pointcuts:
	`pointcut TraceMethod():`
	`execution(* *.CalculatorClass.*(..));`
		With attributes on the aspect method and the method's signature:
		`[IncludeAll]`
		`[Call(Invoke.After)]`
		`object MyCode(object[] )`

Interweaving Aspects	Implicit through pointcuts:
	`pointcut TraceMethod():`
	`execution(* *.CalculatorClass.*(..));`
		Explicit with class attributes or at instantiation:
		`[MyAspect]`
		`public class CalculatorClass { ... }`
		or:
		`Weaver.CreateInstance(typeof(CalculatorClass), null, new MyAspect());`

Definition of aspect code	In advices
	`after():TraceMethod()`
	`{ ... }`
		In aspect methods
		`...`
		`[Call(Invoke.After)]`
		`object MyCode(object[])`
		`{ ... }`

Table 1: AspectJ vs. Rapier Loom .NET

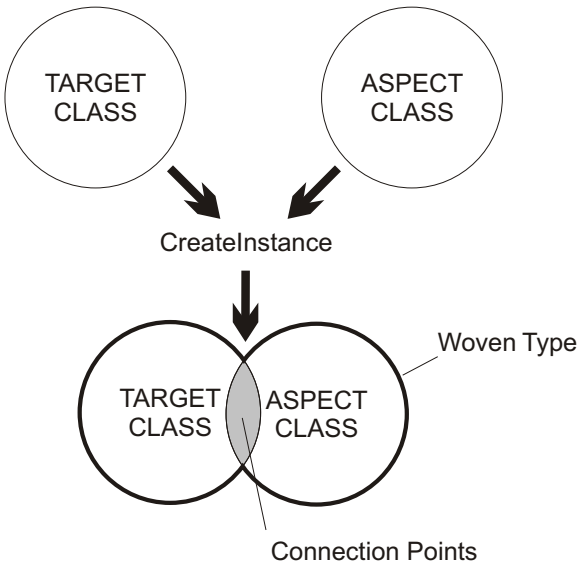

Figure 1: The weaving process

during runtime on intermediate language code. Through that, it is possible to define and interweave aspects weaver for all .NET languages.

4. A DESIGN BY CONTRACT ASPECT

The Rapier LOOM.NET weaver library presented in the last section allows us to intercept method calls and to perform checks before as well as after each method call. This, combined with the possibilities for .NET attributes, is the basis for a Design by Contract aspect, which we describe in this section.

4.1 Formulating Conditions as .NET Attributes

.NET genuinely provides a mechanism to add metadata like Design by Contract conditions[2] to source code and to executables. *Attributes* are keyword-like descriptive declarations that can be used to assign programming elements such as classes, fields or methods in a typesafe way [17].

We defined three different .NET attributes to be used for interface augmentation: `InvariantAttribute`, `PreconditionAttribute` and `PostconditionAttribute`. They are implemented as normal classes, but (indirectly) inherit from the `System.Attribute`, which allows them to be used in the way shown in listing 2. Each attribute stores the condition's boolean expression and an (optional) name.

[2]Unfortunately the term *assertion* is in .NET commonly connected with the usage of the `Debug.Assert` method. From now on we will use the more general term *condition* instead.

```
[Invariant("0 <= count")]
[Invariant("count <= capacity")]
public interface IDictionary
{
    // Insert x so that it will be retrievable
        through key.
    [Precondition("key.Length > 0")]
    [Postcondition("get(key) == x")]
    [Postcondition("count = old.count + 1")]
    void put (object x, string key);

    // ... Interface specifications of other
    // features ...
}
```

Listing 2: Definition of an contract in C#

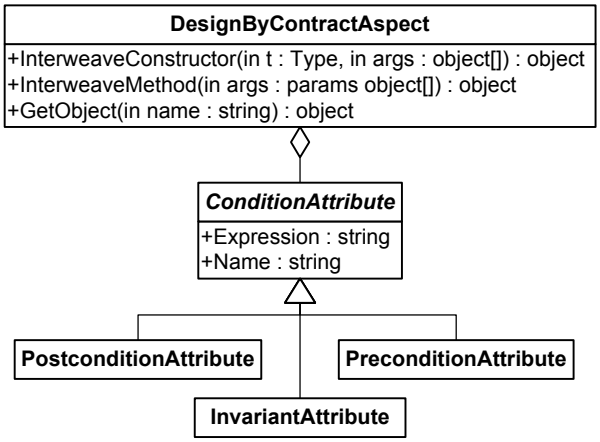

Figure 2: UML 1.4 class diagram of the Design by Contract aspect

While instances of the `PreconditionAttribute` and `PostconditionAttribute` classes (the attached "Attribute" can be omitted) can only be attached to methods or properties, objects of the `InvariantAttribute` class can be assigned to a class or to an individual field. This reflects the different scope of the conditions: Invariants must always be tested for each method call, while pre- and postconditions are dedicated to one method.

4.2 The Aspect Class

The attributes just discussed allow us to write Design by Contract conditions as integral part of the source code. But they are merely passive: They can't enforce that the expressions stored as conditions are syntactically or semantically correct, nor that the conditions are kept during runtime. This has to be ensured by aspect code that is woven to each class and method to be checked. This aspect code is formulated in a so called aspect class (see also figure 2).

The aspect class contains two methods to be interwoven: `InterweaveConstructor` is called whenever a new target class instance should be created, `InterweaveMethod` whenever a method of the target class is invoked.

4.2.1 Creating interwoven objects

As already described in section 3, a factory method `Weaver.CreateInstance(...)` has to be used for creating interwoven objects. This factory method takes as parameters the type of the target object to create, the potential parameters for the target object constructor and the aspect objects (in our case an instance of the DesignByContractAspect class) to interweave with:

```
object[] constructorArgs = {};
Aspect aspect = new DesignByContractAspect();
IDictionary dict = (IDictionary)Weaver.
    CreateInstance(typeof(DictionaryImpl),
    constructorArgs, aspect);
```

Instead of imperatively setting the aspect, it could have also been attached as an attribute to the `DictionaryImpl` target class.

4.2.2 InterweaveMethod semantics

The method `InterweaveMethod` is the central part of the aspect class. It's task is to collect on each target method call all conditions that have to be checked, and to find out wether they all evaluate to true. Both steps are described in detail in the next two sections.

We try to minimize the performance penalty of collecting and evaluating the conditions by the extensive use of buffers and caching.

4.2.3 Collecting Conditions

The collection the Design by Contract attributes to be checked is done using standard .NET reflection mechanisms (see also [17]). Whenever a method is called for the first time the target class and each interface it's implementing have to be examined for Design by Contract attributes. This is due to the fact that – in opposite to the case of class inheritance – attributes assigned to interfaces don't become automatically part of the class implementing this interface. In addition to that, names of parameters used in a condition's expression might have changed in the hierarchy. Therefore parameter names have to be renamed to a standardized form.

In the case of Postconditions, boolean expressions have also to be checked for the use of the "`old`" keyword. As described in section 4.3, this keyword is used to allow access to the state of variables *before* the method call. Therefore, the variables referenced in that manner have to be copied in advance to the call of the actual target method.

4.2.4 Evaluating Conditions

The boolean expression of a condition is stored in the corresponding attribute object as a text string. This is necessary because the expressions should not be evaluated during compile time. However, this makes compilation/interpretation during runtime necessary.

One possibility to achieve runtime evaluation is to built a parser and an interpreter for conditional expressions. This is a complex and error-prone task; it would be much simpler to use an existing compiler, which the Design by Contract aspect also does. The .NET framework allows the programmatic usage of the C# compiler, but only for valid complete source classes, and not for singular expressions. Therefore, the Design by Contract aspect has to dynamically generate a full valid class for every condition.

```
public class ExpressionEvaluator
{
    System.Int32 key_Length;
    public bool Evaluate(Delegate getObject)
    {
        try
        {
            object[] xParam = { "key.Length" };
            key_Length = (System.Int32)
                getObject.DynamicInvoke(xParam)
                ;
        }
        catch (TargetInvocationException ex)
        {
            throw ex.InnerException;
        }
        return (key_Length > 0);
    }
}
```

Listing 3: Dynamically generated code for `key.Length > 0`

The generated code for an expression `x==0` is shown in listing 3. The only method `ExpressionEvaluator.Evaluate` takes a delegate (a typesafe function pointer) with the signature "`object GetObjectDelegate(string objName)`". This delegate is provided by the aspect class and used to initialize the variables referenced in the condition's expression with their current value. Finally, the method simply returns the result of the boolean expression. When compiled, the generated assembly is loaded into memory and can be used to evaluate the condition whenever necessary.

To generate the code just described the aspect has to find out which variables are used in the expression, and which type they have. In the given example the string `x==0` uses the variable x, and its type in the target class is `int`. Right now this is done by regular expressions and reflection on the target class.

The compiled and loaded condition is ready to use and now has to be called at the correct locations in the control flow. In the case an condition fails – either because the boolean expression is malformed or because the condition is not satisfied – the contract was violated or is malformed. Therefore either a `PreconditionException` or a `PostconditonException` or an `InvariantException` is raised.

4.3 Expressiveness of Conditions

As we use the Microsoft C# compiler to compile the given conditions, the condition can use every feature for expressions specified in the C# language specification [28]. Using other .NET languages for the conditions is of course possible, but would require changing the implementation of the aspect and the dynamic code it generates.

The variables that can be used in conditions are limited to the scope of the target class. Only (static and non-static) class variables and – for pre- and postconditions – method arguments can be evaluated, while the usage of other variables result in a runtime exception. Postconditions can also retrieve the state for a class variable or parameter, before the method execution takes place. This is done by prepending an "old." to the variable name.

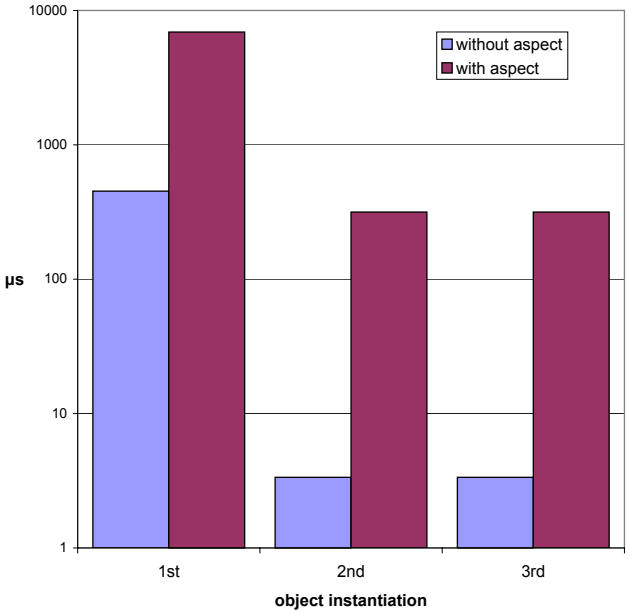

Figure 3: Execution time of a class instantiation with and without Rapier LOOM.NET

5. EXPERIMENTAL EVALUATION

We have tested the runtime overhead of using Rapier LOOM.NET and the Design by Contract aspect on a PC with a single Intel Celeron/800 Mhz CPU and 256 MB RAM. The tests were conducted using the Microsoft .NET framework 1.1, running on Windows XP. By using an high resolution performance counter of the Win32-API, we could make measurements with a resolution of about four microseconds.

The results of the measurements for a simple aspect as shown in listing 2 are depicted in figure 3 and 4. While the first creation of a weaved object needs roughly tenth as much time as a simple creation via `new`, this runtime overhead is near the factor 100 when it come's to the second or third weaving of the target object with the same aspect.

While the performance penalty for object instantiation is caused solely by the dynamic Rapier LOOM.NET weaver, figure 4 shows the overhead for a method call that is caused by both the Rapier LOOM.NET weaver technology and our aspect. The used method had a simple precondition aspect attached, and took two parameters. When called, the method immediately returns. Especially the first call takes significantly longer because the Design by Contract condition has to be compiled and loaded into memory. For consecutive calls the already compiled condition method can be reused. However there is still the need to call this method and the target method dynamically, which results in a runtime overhead of factor 100.

We also experimentally applied the aspect during the development of a large real-world software system completely written for the .NET platform. The system's task is to handle distributed and time-critical business processes that form the backend for a network of national branch offices. A part of the project that deals with the development of the dialog flow control and the business logic is conducted by a

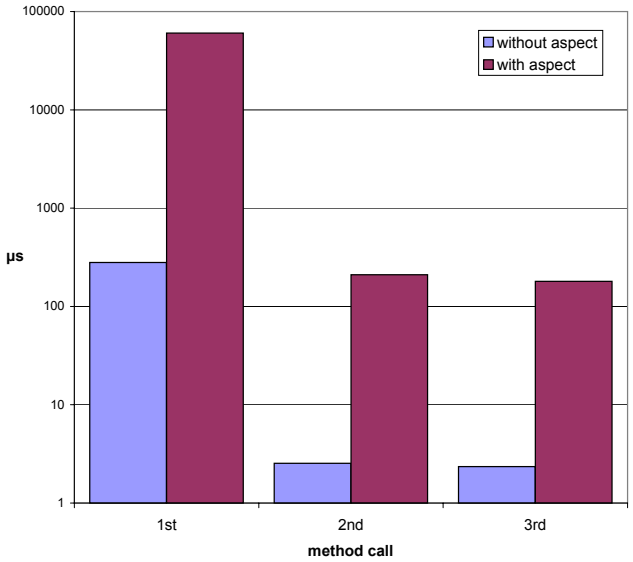

Figure 4: Execution time of a method call with and without the Design by Contract aspect

research partner of our institute. About 140 different business processes have to be mapped to user interface masks, occupying about 20 developers for several months.

We decided to analyze the usefulness of the Design by Contract aspect for the interfaces between this part of the project and the infrastructural parts provided by other companies. We were hoping that using Design by Contract would make the interface semantics more explicit, helping to avoid problems in the integration phase.

One encouraging result was that the aspect could actually save around five per cent of lines of code in an average module. Especially a lot of range tests for method parameters were until then implemented explicitly in each interface implementation, often building the overwhelming part of the actual method body. By specifying these conditions as Design by Contract conditions, these range checks could be made an explicit part of the interface, allowing the method only to contain actual business logic.

However, the evaluation also showed some restrictions of the current implementation, which in part are due to the use of the Rapier LOOM.NET weaver and it's nature as a class library. While for instance the need for using the factory method to create woven objects has the advantage that you can control programmatically wether contract checking should happen, it is nevertheless in most cases an invasive necessity. One could also wish that the syntax of the contracts should be checked at compile time, or that runtime overhead is only caused by the actual checking of conditions, and not the sole usage of the weaver.

These are all weak points that aren't caused by the idea of using AOP as such, or by the specific implementation of the aspect, but by the weaver technology we've chosen. To circumvent that, we are currently working on a static weaver for .NET called Gripper LOOM.NET[3], which will allow the usage of exactly the same aspect, while circumventing the given limitations.

6. RELATED WORK

Design by Contract is a very pragmatic declarative approach to describe and ensure the behavior of objects. While it has – in contrast to more formal approaches of program verification – a limited expressiveness, it's main advantage is to be easily understandable and usable for everyday programming. Barnett et al. provide in [2] a good overview of more advanced possibilities for behavioral interface specifications.

There has been a lot of work done to add support for the Design by Contract paradigm to all sorts of languages and frameworks such as Perl [5], Python [20], Ruby [11], Ada [15], Lisp [10], Smalltalk [4] and C++ [9], as well as Java and .NET. All these tools have to deal with two main challenges: The conditions have to be integrated somehow in the source code, and tested during runtime. How existing approaches manage to do both shall be presented in the next two sections especially for Java and .NET.

6.1 Design by Contract for Java

Java (originally invented by Sun Microsystems, [8]) is nowadays one of the most popular object oriented languages and execution environments. Adding Design by Contract to Java is in the top ten list of official requests for enhancements on the Sun website [25], but it seems unlikely that direct support will be added to the official standard of the language in the near future. Therefore there are numerous interesting approaches to add support to Java: iContract, jContractor, Handshake, JMSAssert, Jawa, Jass and Barter.

Most of the tools embed conditions in Javadoc comments, marked with special tags. Such comments will be ignored by the Java compiler, therefore an external pre-processor has to extract the information from the source files. In contrast to that the jContractor [12] requires conditions to be formulated inside methods following certain naming conventions. These methods can be part of the normal class or placed in a special contract class, which allows also the description of contracts for interfaces. Handshake [6] also does not access the source files: Contracts are formulated in separate files.

iContract [13], Jawa [16], Jass [3] and Barter [26] are precompilers that generate new sources for each class implementing a contract. The new sources, containing checks for the conditions in the method implementations, can be compiled by every Java compiler. Handshake delays the instrumentation of the classes until class loading time: It provides an augmented standard C library that intercepts every call to the file system by the Java virtual machine and generates new extended byte code for the classes to load. JMSAssert uses a special library to register in the JVM and to assign condition triggers to the corresponding methods. This unfortunately seems to work with certain virtual machines only. Finally, jContractor provides two methods to instrument classes during runtime: The default method uses Java's possibility to provide a specialized class loader. If this

[3]A very early version is available at our website: `http://www.gripper-loom.net`

is not possible, the user can also create instrumented objects through a special object factory, which creates an extended child of the original class.

Beside these special tools Martin Lippert et al. in [14] shows that AOP weavers like AspectJ can be used to implement a Design by Contract aspect. In fact, Barter is just a preprocessor that generates aspects from the conditions stored in the source file comments and uses AspectJ to weave these aspects to the original source files. Takashi Ishio et al. also presented in [27] an approach to write conditions in form of normal Java comments inside method bodies, and use AspectJ to extend them to Java statements.

6.2 Design by Contract for .NET

The Microsoft .NET platform [19] is an object-oriented execution environment that can be used with any language conforming to the Common Type System and adhering to some standards. The most popular of these is C#, which is especially designed to make full use of the framework.

One of the languages also targeting the .NET platform is Eiffel.NET, which instantly provides support for Design by Contract [24]. Alternatively an extended version of C# by ResolveCop named XC# [21] can be used, which allows the specification of conditions through .NET attributes[4].

However, the decision for a specific programming language depends on many different constraints, and often people want to stick to their favorite .NET language. If so, Arnout and Simon from Interactive Software Engineering propose in [1] a so-called "Contract Wizard" to make the extended support for Design by Contract in Eiffel.NET available to all .NET languages.

The Contract Wizard works by applying the Decorator Pattern (as defined by Erich Gamma et al. in [7]) to .NET assemblies, generating proxies in Eiffel.NET which check assigned conditions before and after each call propagated to the original assembly. The Contract Wizard allows the user to interactively add conditions to interfaces and classes which are already compiled to .NET assemblies. These conditions are added to generated proxy classes in Eiffel, which should be used – after being compiled to another .NET assembly – as a replacement for the original ones.

7. CONCLUSIONS

Within this paper we have shown that AOP techniques together with customizable metadata could be easily used to provide declarative programming concepts like Design by Contract in imperative/object oriented languages. The Design by Contract aspect we have presented is both powerful and was easy to implement. Pre- and postconditions as well as invariants can be expressed by language-inherent means. By using the dynamic Rapier LOOM.NET runtime library, we furthermore could avoid an external tool or compiler.

Our evaluation in a large real-world project has shown that the aspect can significantly reduce the lines of code and could help to enforce and clarify the semantics of interfaces. The presented aspect has the potential to improve both code quality and readability.

[4]Attributes are further described in section 4.1

8. REFERENCES

[1] K. Arnout and R. Simon. The .NET contract wizard: Adding Design by Contract to languages other than Eiffel. In *Proceedings of TOOLS USA*. IEEE Computer Society, 2001.

[2] M. Barnett and W. Schulte. Contracts, components and their runtime verification on the .NET platform. Technical Report MSR-TR-2002-38, Microsoft Research, One Microsoft Way, Redmond WA, 98052-6399, USA, Apr. 2002.

[3] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with assertions. *Electronic Notes in Theoretical Computer Science*, Jan. 2004.

[4] M. Carrillo-Castellón, J. García-Molina, E. Pimentel, and I. Repiso. Design by Contract in Smalltalk. *Journal of Object-Oriented Programming*, 9(7):23–28, November/December 1996.

[5] D. Conway and G. C. Goebel. *Contract - Design-by-Contract OO in Perl*, Feb. 2001. `http://search.cpan.org/~mdupont/Introspector-0.04/lib/Class/Contract.pm%`.

[6] A. Duncan and U. Hölzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA 93106, USA, Dec. 1998. `http://www.cs.ucsb.edu/labs/oocsb/papers/TRCS98-32.pdf`.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[8] J. Gossling, B. Joy, and G. Steele. *The Java language specification*. Addison-Wesley Publishing Company, 1996.

[9] P. Guerreiro. Another mediocre assertion mechanism for C++. In *Proceedings of TOOLS 33*, 2002.

[10] M. Hölzl. Design by Contract in Common Lisp. `http://www.muc.de/~hoelzl/tools/dbc/dbc.lisp`.

[11] A. Hunt. Design by Contract in Ruby, Aug. 2002. `http://www.pragmaticprogrammer.com/ruby/downloads/dbc.html`.

[12] M. Karaorman, U. Hölzle, and J. Bruno. jContractor: A reflective Java library to support Design By Contract. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2nd Int'l Conf. Reflection*, volume 1616 of *LNCS*, pages 175–196, Berlin, 1999. Springer Verlag.

[13] R. Kramer. iContract – the Java Design by Contract tool. In *Proceedings of TOOLS USA*. IEEE Computer Society, 1998.

[14] M. Lippert and C. V. Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427. ACM Press, 2000.

[15] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe. ANNA – a language for annotating Ada programs. *Lecture Notes in Computer Science*, 16, 1987.

[16] D. Meemken. *User Manual for the JaWA-Precompiler, Version 1.0*. `http://theoretica.informatik.uni-oldenburg.de/~jawa/doc.engl.html`.

[17] The Microsoft Developer Network. `http://msdn.microsoft.com`.

[18] Object Management Group. *Unified Modeling Language specification – Version 1.4*, Sept. 2001.

[19] D. S. Platt. *Introducing Microsoft .NET*. Microsoft Press, third edition, 2003.

[20] R. Plösch. Design by Contract for Python. In *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, pages 213–219, 1997.

[21] ResolveCorp. eXtensible C#, Nov. 2002. `http://www.resolvecorp.com/XCSharp.ppt`.

[22] W. Schult. Rapier-Loom.Net homepage. `http://www.rapier-loom.net`.

[23] W. Schult and A. Polze. Aspect-oriented programming with C# and .NET. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 241–248, Crystal City, VA, USA, April 29 - May 1 2002.

[24] R. Simon and E. Stapf. Full Eiffel on the .NET framework. *MSDN library*, July 2002. Available at `http://msdn.microsoft.com/library/en-us/dndotnet/html/pdc_eiffel.asp`.

[25] Sun Microsystems. Bug database – top 25 RFE's (Request For Enhancements), Mar. 21, 2004. Available at `http://developer.java.sun.com/developer/bugParade/top25rfes.html`.

[26] V. Szathmary. Barter – beyond Design by Contract, 2002. `http://barter.sourceforge.net`.

[27] S. K. Takashi Ishio, Toshihiro Kamiya and K. Inoue. Assertion with aspect. In *International Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT2004) in conjunction with AOSD2004*, Mar. 2004.

[28] Technical Committee 39, Task Group 2. *C# language specification – Standard ECMA-334, 2nd edition*. ECMA International, 2002.