

# Building Trust in Third-party Components using Component Wrappers in the .NET Frameworks

Christine A. Mingins, Chee Y. Chan

School of Computer Science and Software Engineering

Monash University

PO Box 197, Caulfield East 3145, VIC, Australia

cmingins@csse.monash.edu.au, cychan@csse.monash.edu.au

## Abstract

Software purchasers are often provided with very sparse information about how to correctly deploy software components. We describe a novel contract-based approach to building trust in third party components. In contrast to the usual approach where contracts specifying the software semantics are constructed at the time of source code generation, we retrofit existing components with Contract wrappers. Sets of client requirements on the component are expressed in terms of software contracts using the Design by Contract™ method. The targeted component is then wrapped in this executable specification. Subsequent calls to the component are made via the wrapper. Any client misunderstanding about the component's functionality will be picked up as contract violations. Contract wrappers also serve as the basis for test suites for integrating components into applications, and as documentation for future updates.

We demonstrate how such contract wrappers can be constructed and deployed using the cross-language metadata capabilities available in Microsoft's new .NET Framework.

*Keywords:* Design by Contract; software quality; wrappers, software components

## 1 Introduction

Many software developers experience difficulties with integrating third party components into systems. This is particularly the case for 'binary' components, with no accompanying source code. The problem springs from lack of information about detailed semantics of the operation of the component. Usually the only published information is a set of interface definitions, perhaps supplemented by written documentation. This problem exists not only for component clients at point of sale, but also for maintainers of component-based systems.

The problem of integrating existing components into systems can be expressed in the following way: The client's requirements may be only a subset of the component's advertised features; how can we employ the client's requirements as a specification set for

- a) determining that the component meets the requirements and
- b) acting as a mediator in the integration process?

From a system perspective, one solution is to "wrap" the component in an executable specification that expresses the client's requirements, using the wrapper as a proxy for the component proper. From a technical perspective, the proxy must intercept each client request, check its validity against the specified requirements for that method, delegate execution to the target component method, and manage any error conditions that may arise.

In this paper we describe how we have used the Design by Contract™ method to incorporate assertions representing client requirements into component wrappers.

## 2 The Design by Contract Method

Design by Contract™ (Meyer 1998) is a well-known approach to ensuring software quality. It is founded in Hoare logic (Hoare and Sheperdson 1985). Hoare triples  $\{p\}S\{q\}$  intuitively assert that if the program  $S$  is executed in a state  $\sigma$  such that  $p\sigma$  holds and  $S$  terminates, then it will produce a state  $\tau$  such that  $q\tau$  holds. The Eiffel programming language embeds such assertions in the class text as sets of executable specifications. Preconditions (*require* statements) declare the conditions under which a client may call a routine of an instance of a class. Postconditions (*ensure* statements) declare conditions that must hold for the instance after a routine has been executed. Invariants (*invariant* statements) state the consistency conditions for any valid instance of the class, immediately after creation and on return from any routine call. Assertions are compiled into executable code. At runtime assertion violations cause exceptions to be raised, either in the caller (precondition) or the implementing class (postcondition, invariant). Assertions can be inherited and modified in descendants, according to a set of rules that preserve the semantics of the classes and routines.

Assertions in Eiffel, then, represent a set of contracts between suppliers of classes and their clients, placing obligations on both parties in order to successfully use an

object. Developers typically generate assertions right at the beginning of the development lifecycle. Clear specification of the semantics of classes, their attributes and their routines lies at the heart of the concern for developing quality software expressed in the Design by Contract approach. Preconditions guide class users. Preconditions, postconditions and invariants guide class maintainers and extenders. They are used to guide the coding (implementer) and as a foundation for test suites for use by both the implementer and the client integrating the class into an application. Eiffel's "short-flat" form of a class, showing for a class only exported attributes and routines with their comments and assertions, forms a human-readable client specification document, extracted from the live code.

### 3 Contracting Existing Components

In our approach we use software contracting to extend component Interface Definitions by specifying and documenting requirements for the correct use and execution of publicly accessible methods, and specifying what it means for a component to be in a consistent state. These Software Contracts are embedded as executable specifications in component wrappers or proxies. They also provide a basis for generating test suites for integrating components into applications. Post-integration, they publish detailed descriptions of component semantics to guide system maintenance.

The following scenario shows a typical situation encountered by component purchaser, demonstrating the usefulness of retrofitting contracts to third-party binary components:

EPS Pty Ltd is a software development company that builds systems for auditing firms. To ensure on time delivery of quality system, they have a practice of acquiring quality software components from Quality Components Ltd. The package includes binary components and text documentation detailing their interfaces. EPS learns about the benefits of design by contract and would like to apply the concept in their system development process. They open a component with the Contract Tool and insert contracts into its interface interactively. These contracts represent EPS's understanding of the functional behaviour of the component, based on their reading of the documentation. They also serve to restrict the component interface to the functionality that the company requires. The tool builds a proxy that serves as the gateway to the original component. Access to the component is checked against the contracts in the proxy and any violation of the contracts triggers an exception. This greatly simplifies debugging, testing, system development and integration processes while ensuring quality of final product. As EPS does not want to include overhead of checking contracts in the final delivery, they remove the proxy with minimal effort by substituting a reference to the proxy with a reference to the initial component. The contracts in the proxy provide valuable documentation for future system maintenance and component upgrades.

## 4 Related Work

Software wrappers are a general-purpose mechanism for adding new functionality or to restrict access to existing functionality in the embedded software. The technique has been applied in areas as diverse as security and privacy (Fraser, Badger, and Feldman 1999) and database access (ODBC) (Microsoft 2001). Various techniques have been used, including pre-processors and class libraries.

Over the past 10 years researchers have developed mechanisms to graft assertions onto existing programming languages such as Java (Karaorman, Holzle, and Bruno 1999, Kramer 1998), C++ (Plösch and Pichler 1999) and Smalltalk (Brant, Foote, Johnson, and Roberts 1998). Techniques used range from new constructs inserted as special comments in the source code and preprocessed before compilation, class libraries and modified development environments. Most efforts to add contracts have been at the source code level. Development environments have until now lacked the reflection capabilities to interrogate binary components, retrieving enough detail about their functionality to support wrapping them in contracts.

## 5 .NET Metadata and Component Wrappers

All programs that compile in Microsoft's .NET Frameworks (Microsoft 2001) generate an 'executable', comprising intermediate code called Common Intermediate Language (CIL) and metadata about the module. The CIL and the metadata are placed in files that extend the Portable Executable (PE) format used for .exe and .dll files. The metadata describes a component's publicly accessible types, members and signatures. In fact enough information is provided by the metadata to support the dynamic instantiation, invocation and even the sub-classing of a type. Furthermore, no matter what the source language, once a program has been compiled, as long as the Common Language Specification has been adhered to in constructing it, then the resulting module or component is accessible across all the other languages that compile to the .NET Frameworks.

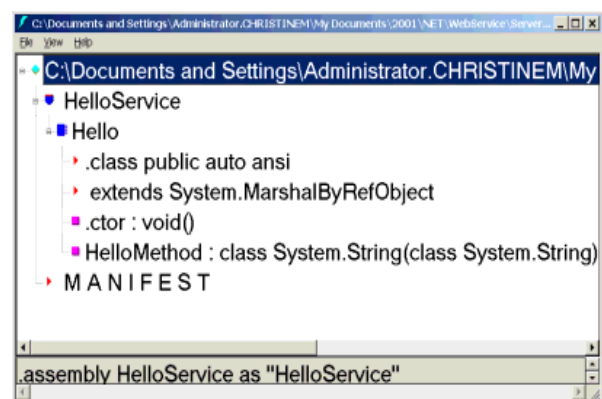


Figure 1: .NET Metadata

Figure 1 shows the output of the .NET tool ILDASM, displaying the structure of a library called HelloService. The assembly contains a class

Hello which inherits from MarshalByRefObject. The members of Hello are a constructor method, and method HelloMethod which takes one String argument and returns a String. Once compiled, HelloService can be loaded and compiled to native code and executed on any platform supporting the .NET runtime. It is now in a language-independent format and can be 'discovered', interrogated dynamically, instantiated and even subclassed by programs written in any language targeting .NET. This makes the .NET environment ideal for taking a component compiled from any source language implemented in .NET, accessing its public interface, and generating a contract wrapper or proxy that captures input messages, checks the relevant assertions and redirects the call to the embedded component.

## 6 Contract Tool Implementation

The Contract Tool makes extensive use of the metadata available in a .NET component. The tool extracts information from the metadata to build a type hierarchy from the component. Types are shown as tree nodes in a browser and type members (methods, properties and attributes) are shown as sub-nodes of the tree view browser. The view is much the same as shown in Figure 1 above. The person building the contracts uses the browser to navigate through the type hierarchy in the component. There are two ways to define contract definitions for the wrapper or proxy. The first method is to add preconditions, postconditions, and invariants with the contract tool browser. Contract entries, including base entries are shown in the browser for clarification. The second method is to make use of .NET Custom Attributes. Custom attributes provide a means for developer to insert descriptive information into the metadata of a .NET component. Using this approach, contract definitions are defined as custom attributes in the component. The tool extracts this information from the component metadata at the time the component proxy is generated.

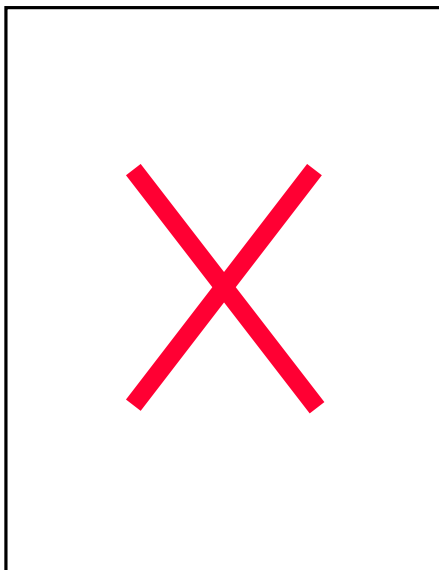


Figure 2: Contract Tool User Interface

Figure 2 shows the Contract Tool User Interface. Developers open the file containing a component, target a type to enter an invariant expression, or type member to enter a precondition or postcondition. Any existing assertions are automatically displayed.

The contract tool generates a proxy class for each of the public classes in the targeted component. This proxy class has a same name as its direct counterpart, but within a different namespace. The proxy class contains an instance pointing to its reference class. The inheritance type hierarchy is maintained, including interface implementation. The proxy class exposes all the public members that is, the interface of the initial type. In general, there is a direct mapping between members of the initial class and its proxy class. However there is an exception with the *field* members. Field members are exposed as *properties* in the proxy. In the .NET common language infrastructure, properties encapsulate fields, allowing for *getter* and *setter* functions, thus allowing contract validation in field accesses. In any case direct attribute exposure is generally not a good practice in object-oriented programming. The following code gives a simple example to illustrate the mapping relationship. Figure 3 shows the original C# source code for the class.

```
// assertion member name
{
    if (!(condition1))
        throw new Contract.InvariantException
            ("tag/description");

    if (!(condition2))
        throw new Contract.InvariantException
            ("tag/description");
}
```

Figure 3: C# Source Code

Figures 4 and 5 show source code generated by the Contract Tool for the proxy or wrapper class. Note that although the classes have the same name, A, they are distinguishable by the namespace that they reside in. The original resides in namespace Example and the proxy in namespace Example.Contractured. Instance-reference points to the target class.

The comments in the body of the methods indicate where the C# source code for the assertions are placed. Note that the public attribute attribute\_a now has accessor methods, hopefully also with contracts, in order to guarantee the consistency of the component state. Note also that private\_function is not present in the wrapper; private members are not subject to contracts.

```

//Contract Wrapper
namespace Contracted.Example
{
class A : Example.A //inherit from Example.A
{
public A() // Constructor
{
// precondition checking. Invariants
// not checked.
instance_reference = new Example.A();
// postcondition and invariant checking.
}

public void function_a()
{
//preconditions and invariant checking
// call the reference method function_a
instance_reference.function_a();
// postconditions and invariant checking
}
...
}
}

```

Figure 4: Generated C# Source Code (1)

```

Public int property_a // property
{
get
{
// invariant checking
return instance_reference.property_a;
}
set
{
// precondition and invariant checking
instance_reference.property_a = value;
// postcondition and invariant checking
}
}

public int attribute_a // here is a property
{
// accessors go here, including contract checking.
}
//Reference to the enclosed class
Example.A instance_reference;
}
}

```

Figure 5: Generated C# Source Code (2)

The Contract Tool is also faithful to the Design by Contract method in preserving assertion semantics in inheritance hierarchies. Preconditions are weakened - parent's assertions are ORed with those of the inheriting

class. Postconditions and invariants are strengthened - parent's assertions are ANDed with those of the inheriting class.

Of course contractors do not have to hand-code the proxies. The Contract Tool provides a user interface for browsing assemblies and components. Developers generate the wrappers by entering assertions for targeted classes and their members.

## 7 Error Handling

Three assertion exception types have been defined to report assertion violations. They are `Contract.InvariantException`, `Contract.PreconditionException` and `Contract.PostconditionException`. Implementation of all assertions follows the pattern for invariants shown in figure 6 below:

It is important to note that exceptions thrown as a result of assertion violations do not necessarily denote errors in the embedded component. They may also indicate misunderstandings by the client, as represented in the contracts, about how the component operates. Errors discovered in the component itself could possibly be patched by building work-arounds into the wrapper, but this use of wrappers is beyond the scope of this discussion.

```

// assertion member name
{
if (!condition1)
throw new Contract.InvariantException
("tag/description");
if (!condition2)
throw new Contract.InvariantException
("tag/description");
}

```

Figure 6: Assertion implementation

## 8 Review of the Contract Tool

There are two limitations to the Contract Tool as it is currently implemented. The first is that the types enclosed within a component must reside within a namespace. This is because of the naming conventions we have adopted for the proxy class. All proxy classes are sited in the `Contracted` namespace. If a class does not have a namespace, the compiler generates an error message as it cannot differentiate between the class and its proxy. The second problem is that the tool can only analyse a single component at a time. If the component references types in other components, these types will not be contract bound. However, the future releases of the tool should address this limitation.

## 9 Conclusion and Future Work

The Contract Tool supports a method for retrofitting contracts onto binary components compiled into

Microsoft's .NET Frameworks. This allows component purchasers to 'build' trust into components by constructing Contract Wrappers. These wrappers also establish a basis for debugging and testing, significantly ease the burden of component integration and provide excellent documentation for future revisions. The design and construction of the Contract Tool was straightforward, thanks to the powerful reflection capabilities of the .NET metadata. This metadata comes 'free' with any executable or library compiled into the .NET runtime. Thus even 'black box' components for which no source code is available may be interrogated, their type structure retrieved, and wrappers generated for them.

Of course, as stated earlier, wrapping techniques have been applied in other domains such as security, component functionality restriction and extension. Our Contract Tool could be modified to generate such wrappers.

We are exploring the use of .NET *custom attributes* [10] for encoding contracts directly into C# source code and that of other .NET-enabled languages supporting custom attributes. This will provide another method building contract wrappers, as well as allowing programmers to add assertions directly into source code. Custom attributes add user-defined annotations to the metadata. They allow an instance of a type to be stored with any element of the metadata. This mechanism can be used to store application specific information at compile time and access it at runtime. Custom attributes are used to store (document) contract entries. Three attribute types are included in the Contract namespace: *Require*, *Ensure* and *Invariant*. These attributes can only be applied to constructors, methods, properties, and fields. The *Invariant* custom attribute is applied to classes and interfaces. The constructor of the attribute class takes a string of form "description: validation code" where 'description' describes the entry and 'validation code' is the actual assertion condition. It is also possible to specify multiple custom attribute entries for any class or member.

## 10 References

- BRANT, J., FOOTE, B., JOHNSON, R.E., ROBERTS, D. (1998): Wrappers to the Rescue. *Proc. European Conference on Object Oriented Programming*, ACM Press.
- CARRILLO-CASTELLON, M., GARCIA-MOLINA, J., PINENTEL, E. (1994): Eiffel-like assertions and private methods in Smalltalk, *Proc. 26<sup>th</sup> Conference on Technology of Object-Oriented Languages and Systems*. Prentice Hall.
- FRASER, T., BADGER, L., FELDMAN, M. (1999): Hardening COTS components with generic software wrappers. *Proc. 1999 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press.
- HOARE, C.A.R., SHEPERDSON, J.C., (eds) (1985): *Mathematical Logic and Programming Languages*. Prentice Hall.

KARAORMAN, M., HOLZLE, U., and BRUNO, J. (1999): jContractor: A Reflective Java Library to Support Design By Contract. *Proc. Second International Conference on Metalevel Architectures and Reflection (Reflection'99)*, Saint Malo, France, 1616, Springer-Verlag.

KRAMER, R. (1998): IContract – The Java™ Design by Contract™ Tool, *Proc. 26<sup>th</sup> Conference on Technology of Object-Oriented Systems*, IEEE Computer Society Press.

MEYER, B. (1998): *Object Oriented Software Construction*. Prentice Hall.

Microsoft (2001): ODBC Web Site  
<http://www.microsoft.com/data/odbc/default.htm>

Monash (2002): ECMA Draft CLI Standard  
<http://lightning.csse.monash.edu.au/.net/CLI/>

PLÖSCH, R., PICHLER, J. (1999): Contracts: From Analysis to C++ Implementation, *Proceedings of the 30th Conference on Technology of Object-Oriented Systems, Santa Barbara, USA*, IEEE Computer Society Press.