elj.com

A Website dedicated to Innovative Language Technologies

The Eiffel Contract for C++ Programmers

by Paul Johnson <Paul@treetop.demon.co.uk>

What Is a Bug?

Take a look at this class definition:

```
template  class Foo {
public:
   Foo();
   void put (const T &new_item);
   void remove ();
   T item ();
   int empty ();
};
```

This class is used in the following piece of code:

```
Foo <char *> my_foo;
my_foo.put ("Hello World");
my_foo.remove ();
cout << my_foo.item ();
```

Now answer the following questions:

- Is it a bug?
- How do you know?

You can see that the code does a *"put"*, and then a *"remove"*, and then calls *"item"*. But what is the relationship between these calls? *"Remove"* obviously takes something away, but what is left afterwards? Was there something there before the *"put"*, and if so does the *"remove"* put it back? Do these features have anything to do with *"item"* anyway? You can guess at the answers to these questions, and then try to decide if there is a bug or not.

The problem is that the class definition only tells you about the names of the features and their arguments. It doesn't tell you what those features actually do. Traditionally programmers have had three ways of dealing with this situation:

- Read the documentation. If you have paid good money for a third party library then the odds

are you have some good documentation. Otherwise not. When schedules are tight its the first thing to go.

- Read the code. But what guarantee do you have that future maintenance won't change the answer and break something?
- Write a little test program to find out what happens. This suffers from the same problem as (2), plus the risk of hidden side effects that don't show up in your script.

What you need here is a way to determine what a class does in a way that is independent of how it does it, but is still guaranteed to be kept up to date.

Interface Semantics

In the code at the beginning of this article *"Foo"* means *"Stack"*. Does this make the problem easier?

Of course it does. We know what stacks are and how they behave. In particular we know that:

- Its an error to try to get the top item off an empty stack
- A stack starts off empty.
- A number of puts followed by an equal number of removes will leave a stack in its original state.

From these three facts we can deduce that the code fragment is indeed a bug.

Here is another definition of a stack class, written in an imaginary dialect of C++. This allows us to write down the information about what a class does as a collection of assertions.

```
template <class T> class Stack {
public:
   Stack()
      ensure empty ();

   virtual void put (const T &new_item)
      ensure
         count () == old count () + 1 &&
         item () == new_item;

   virtual void remove ()
      require
         !empty ()
      ensure
         count () == old count () - 1;

   virtual T item ()
      require
         !empty ();
```

```
      int empty ();

      int count ();

   invariant:
      empty() == (count() == 0);
      count () >= 0;
   };
```

The new keywords in this dialect are:

- require: An assertion that must be true when the feature is called. Otherwise there is a bug in the caller.
- ensure: An assertion that must be true when the feature exits. Otherwise there is a bug in the feature.
- old: A unary operator used in *"ensure"* clauses. Returns the value its argument had before the routine was called.
- invariant: Assertions that must be true before and after each call.

Taken together, these *"interface assertions"* tell us the key things about stacks:

- The constructor says that a new stack is created empty.
- The *"put"* operation increments the count, and the *"remove"* operation decrements it.
- The *"put"* operation also says that when you put something on the stack it becomes the new top item.
- The *"item"* operation says that you can't get the top item of an empty stack, and the *"remove"* operation says you can't remove it either.
- The invariant says that the stack is empty when the count equals zero, and that the count can never be negative.

Eiffel and Software Contracting

In fact there is a language with the *"require"*, *"ensure"*, *"old"* and *"invariant"* keywords. Its called *"Eiffel"* and was invented around the same time as C++ by Bertrand Meyer. Eiffel compilers have been commercially available ever since. It looks more like Pascal or Ada, but underneath the syntax the features offered by the languag are fairly similar to Java and C++. Rather than explaining the Eiffel syntax in this article I'll carry on using my imaginary C++ dialect.

Eiffel was designed around the concept of *"Software Contracting"*. In this view a class offers a service under particular terms which are listed in its interface. Client code must accept these terms to use the service, thus forming the *"contract"*.

Just like a legal contract, the software contract describes the rights and duties of both partners. The client must conform to the conditions listed in the *"require"* clauses, and in return the class guarantees to perform the service according to the conditions in the *"ensure"* and *"invariant"* clauses. If a class cannot meet its obligations under the contract then it must inform the client by throwing an exception. Silently ignoring a problem is not allowed.

Inheriting Assertions

Where assertions really come into their own is in managing inheritance. In *"The Liskov Substitution Principle"* (C++ Report, March 1996) Robert Martin wrote:

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

This obviously affects the way that function signatures are inherited, but it also affects the way in which behaviour is inherited. By using interface assertions we can state the rules for inherited behaviour very simply:

- *"Require"* clauses may be weakened, allowing the class to accept a greater range of inputs than its ancestor. We cannot make a *"require"* clause more restrictive because then the descendant would not be able to accept all the inputs that were legal for the base class.

- *"Ensure"* and *"Invariant"* clauses may be strengthened, so that the descendant guarantees to do more than the base class. We cannot weaken "ensure" or *"invariant"* clauses because then the client cannot depend on the contract made by the base class.

To put it another way, the contracts of the ancestors shall be honoured by the descendants, yea even unto the Nth generation.

Much the same rules govern maintenance. A maintenance programmer can relax *"require"* clauses or strengthen *"ensure"* and *"invariant"* clauses safe in the knowledge that the changes won't break any existing clients.

For example, suppose we wanted to create a descendant of Stack with a maximum capacity. Its interface would look like this (underlines show new or changed text from the original Stack).

```
template <class T> class Bounded_Stack: Stack {
public:
   Bounded_Stack(int cap)
      ensure
         empty () &&
         capacity () == cap;

   void put (const T &new_item)
      require
         ! full ()
      ensure
         count () == old count () + 1 &&
         item () == new_item;
```

```
      void remove ()
         require
            !empty ()
         ensure
            count () == old count () - 1;

      T item ()
         require
            !empty ();

      int empty ();

      int count ();

      int full ();

      int capacity ();

   invariant:
      empty() == (count() == 0);
      count () >= 0;
      count () <= capacity ();
      full () == (count () == capacity ());
   };
```

The problem appears in the new require clause on *"put"*: it bans the client from adding a new item to a full stack. This means that a Bounded Stack cannot be used in circumstances where a normal Stack would be legal, and hence that Bounded Stack cannot inherit from Stack.

The Long and the Short and the Flat

The Bounded Stack example above shows off a nice trick that Eiffel environments have, known as the *"short flat"* form of a class.

C++ has class declarations in its *".h"* files, and the associated implementation in the *".c"* files. Eiffel does things a little differently. The entire class definition is held in a single file, but the public interface, including the interface assertions and any associated comments, can be extracted by a filter known as *"short"*. The result is similar to the declarations held in a C++ *".h"* file.

However C++ programmers will be familiar with the problem of checking back through ancestors to find out where a routine is defined. Eiffel avoids this by providing the *"flat"* form of a class with all the ancestor routines incorporated. All the redefinitions and renaming is applied, so the flat form of a class shows you the whole thing as it would be if there was no inheritance.

Putting both of these filters together we get the *"short flat"* form of a class which shows the complete interface, including all the inherited features and their inherited assertions, but without any implementation detail. That is what was presented for Bounded Stack above. The actual ".h" file for Bounded Stack is much less friendly.

The really useful thing about the short-flat form is that it acts as the reference document for the class, so you don't need to keep a separate reference manual up to date. You still need to write some general documentation about how the classes in the framework fit together, but this kind of thing is

much less sensitive to changes in the class than the reference manual. So using interface assertions saves you the work of writing and updating the reference manual, and also removes the risk that it will be allowed to get out of date.

Design By Contract

Because assertions state what a class does, they form a key component in the design process. Instead of writing pseudo-code or a vague description for the features, the designer writes the assertions. This gives the programmer clearly defined requirements that the code must meet.

Checking Assertions

Interface assertions document what the code does, but they are not just fancy comments. At run time they work in the same way as *"assert"*: if the assertion is false then the program stops so that you can debug it.

Normally assertions are used in a reactive way: a piece of code seems buggy so we scatter a few assertions in there to see if we can find the fault. Software Contracting uses assertions proactively to define the operation of the software. So if there is a bug then the interface assertions will detect it immediately. This makes debugging quick and easy.

Doing Software Contracting

Doing software contracting in C++ is not easy. Lots of people have tried, but nobody has come up with a really satisfactory solution. In order to do it you need the following:

- Macros called *"require"* and *"ensure"*. These are called at the beginning and end of each routine.
- A virtual function called *"invariant"* which must also be called at the beginning and end of every routine, possibly by the *"require"* and "ensure" macros. The coding standard should require that this routine calls its predecessors when it is redefined.
- A way to call a function within the same object without checking the invariant. This allows you to write functions which can work when the object is in an *"inconsistent"* state (i.e. the invariant conditions are not satisfied) because the object is half way through some operation. Invariants form part of the external interface to an object. The object does not need to obey them in its own implementation.

Of course you can simply write a private routine which does not check the invariants. However a public routine which is also called internally presents a problem.

- A filter which extracts the assertions from the implementation and then prints out the short flat versions of the class definition. In addition to recognising the require, ensure and invariant assertions this filter must also handle the C++ inheritance mechanisms so that the interface assertions are inherited in the right way.

Or you could just try using Eiffel instead. For more information see The Eiffel Liberty web site at

http://www.elj.com/. This contains articles on *" Design by Contract"*, reports of successful projects that have used Eiffel, and pointers to tutorials and free compilers.

Page Last Modified: 06 Feb 2000
Page created: 06 Feb 2000
Homepage: http://www.elj.com/eiffel/cpp/eiffel-contract/
