

UNCLASSIFIED

Defense Technical Information Center Compilation Part Notice

ADP023865

TITLE: Design by Contract: A Simple Technique for Improving the Quality of Software

DISTRIBUTION: Approved for public release, distribution unlimited

This paper is part of the following report:

TITLE: Proceedings of the HPCMP Users Group Conference 2004. DoD High Performance Computing Modernization Program [HPCMP] held in Williamsburg, Virginia on 7-11 June 2004

To order the complete compilation report, use: ADA492363

The component part is provided here to allow users access to individually authored sections of proceedings, annals, symposia, etc. However, the component should be considered within the context of the overall compilation report and not as a stand-alone technical report.

The following component part numbers comprise the compilation report:
ADP023820 thru ADP023869

UNCLASSIFIED

Design by Contract: A Simple Technique for Improving the Quality of Software

Mark Bolstad

US Army Research Laboratory (ARL)/ Raytheon, Aberdeen Proving Ground, MD
mbolstad@arl.army.mil

Abstract

At its heart, Design by Contract (DbC)^[Meyer97] is a technique for expressing the relationship between a software routine (the supplier) and the callers of that routine (the clients). DbC is inspired by commercial relationships and business contracts that formally express the rights and obligations binding a client and a supplier. DbC provides a clean, easy-to-implement technique that specifies the roles and constraints applying to a routine, and ultimately, will improve the quality of any software with minimal additional cost. In this paper we will define what DbC is, how it can benefit any software, and show several examples of software developed at ARL MSRC that use DbC.

1. Introduction

The quest for quality software rests with two basic principles:

Robustness: The ability of software systems to react appropriately to abnormal conditions.

Correctness: The ability of software to perform its exact tasks as defined by their specification.

While closely related, these two principles are not the same. Robustness is the principle of dealing with events that are beyond the control of the programmer, e.g., a failure in the network. The wise programmer will put in methods and code to handle these extra ordinary events. Correctness is the principle that the software is performing exactly as specified. An error in correctness is one that is within the programmer's purview to fix. For example, a sorted list class whose list somehow becomes unsorted is an error of correctness.

In the standard style of programming, sometimes called Defensive Programming, these two concepts are confused. In Defensive Programming every error and/or failure should be caught and handled by the software. There is no distinction between an error (a correctness problem) and a failure (a robustness problem). In DbC, there is a clear distinction between

these two concepts. It is the responsibility of the programmer to fix all correctness errors.

2. Design by Contract

As software developers, our goal is the construction of systems as structured collections of cooperating software elements, communicating on the basis of clear definitions of obligations and benefits. DbC is the mechanism that allows us to express these obligations and benefits in software.

Tony Hoare^[Hoare69] said that correctness is a relative notion and is a consistency of implementation with respect to a specification. To capture this relation, he created the concept known as a Hoare triple:

$$\{P\} A \{Q\}$$

Where P and Q are assertions and A is a set of instructions

The Hoare triple can be interpreted as:

Any execution of A started in a state satisfying P will terminate in a state satisfying Q .

For a simple example, the following arithmetic statement satisfies the Hoare triple:

$$\{n > 2\} \quad n := n + 10 \quad \{n > 12\}$$

The Q assertion in this example, $n > 12$, is not a mistake. Given that P is true, we guarantee that after the execution of the arithmetic statement that n will be greater than 12. While obvious on the surface, these types of assertions become of great value in specifying the correctness of a routine.

Now that we know the definition of a Hoare triple, we can let you in on a little secret: DbC is a method of implementing a Hoare triple in software. First, a little terminology from the definition above. P is called a precondition and Q is called the postcondition. The preconditions are the obligations that a caller must satisfy in order to receive the benefit of the work performed by the callee. The postconditions specify the benefit that the caller will receive if the preconditions are satisfied.

Preconditions and postconditions are implemented as assertions. An assertion is a statement that must be true at certain points in the execution of the software. For programmers familiar with the C programming language, an assertion can be implemented with the *assert* function in the C standard library. Routines may, and usually do, have multiple assertions for both their preconditions and their postconditions. For a routine to be correct, all of the assertions must be true, i.e., the assertions are AND'd together.

Preconditions and post-conditions describe the properties of individual routines. In an object-oriented language, there is also a need for expressing the conditions of a class as a whole. These conditions that describe the semantic meaning of the class are called the invariant. In the context of the Hoare triple, the invariant is evaluated before the precondition and after the postcondition, i.e., using the notation below

{invariant and P} A {Q and invariant}

For example, the invariant of a class that maintains a sorted list is that the list is sorted. So in the definition above, the list must be in a sorted state before and after the execution of A, but A is allowed to violate the invariant and reorder the list during its execution as long as it is restored upon exit of the routine.

Another type of instruction used in DbC is the check instruction. This assertion allows the software developer to express their conviction that a particular condition must be true at a certain point in the software. For example, after performing a calculation, say a matrix inversion, a check statement can be inserted to see if the matrix is truly inverted.

One of the features that make DbC so attractive is that the control of assertions is a compile-time option. During the development of the software, all assertions are enabled. Depending on the extent of the assertions used, full assertion checking may result in a slowdown of the system by a factor of 2–20 or higher. Typically, when the software is ready to be beta tested, the developer will then choose to deploy a version with just precondition checking enabled. Upon release of the final version, all of the assertions can be disabled for maximum system performance.

2. Implementing Design by Contract

Let's start off this section with a simple example to show the principles behind DbC. First, a note about the notation we will be using for several of the examples. The examples are written in the programming language Eiffel^[Meyer92]. Eiffel was chosen for several reasons:

- Eiffel was the first language to support DbC. As a consequence, Eiffel has the most robust support for DbC.

- Eiffel was designed with readability in mind. Think of Eiffel as pseudo-code that executes.

For this example, we will examine a couple of routines of a class that manipulates vectors of arbitrary size. This class is part of a scene graph library and rendering code developed at ARL. Specifically, we will look at the creation of a vector, and adding elements to it.

```
class VECTOR
feature -- Initialization
  make (the_count: INTEGER) is
    -- Make a zero vector with given
    -- number of components.
  do
    if the_count > 0 then
      -- Make a vector
    else
      -- Indicate error
    end
  end

feature -- Element change
  put (value: DOUBLE;
      index: INTEGER) is
    -- Assign 'value' to component
    -- at 'index'.
  do
    if index > 0 and
       index <= count then
      -- Add the value
    end
  end
```

This is “typical” for the defensive style of programming. Since the possible values for the input parameters are unknown, code is added to protect the critical sections from bogus values. In addition, we may have redundant checking of the input. The client checks the values to ensure that they are correct, and then the supplier checks again to determine how to handle the input. Note that there is no specification that tells the client what set of values constitutes a valid range of input. If the client provides a negative count to the *make* routine, is it an error? The advantage of DbC is that it makes the responsibilities of the client and the supplier clear.

Now let's look at this class again using DbC and assertions. In Eiffel, preconditions are listed in a *require* statement, and postconditions in an *ensure* statement. Also, this example assumes the presence of other routines to help check with the postconditions. These routines are not listed here, but their function should be fairly obvious from their names.

```

class VECTOR
  feature -- Initialization
    make (the_count: INTEGER) is
      -- Make a zero vector with given
      -- number of components.
      require
        positive_length: the_count > 0
      do
        -- Make a vector
      ensure
        count_is_length: count =
          the_count
        is_zero: is_zero
      end

  feature -- Element change
    put (value: DOUBLE;
        index: INTEGER) is
      -- Assign 'value' to component
      -- at 'index'.
      require
        non_nan_value: not is_nan
      (value)
        valid_index: index >= 1 and
          index <= count
      do
        -- Add the value
      ensure
        assigned: item (index) = value
      end

```

The first thing to notice is that the benefits and obligations are now clearly defined. For the make routine, if *the_count* is positive, then this routine guarantees a vector of the correct size and initialized to zero. The second item to notice is that the executable statements are no longer "protected" as in the defensive programming example. Since we are guaranteed that the input values meet the specification, there is no need to check these values, thus simplifying our code. Also, since we will incur no runtime penalty for the final code (all assertions are off), we can make the assertions more elaborate and extensive.

The assertions clearly delineate the responsibilities of both the client and the supplier. A failure of the precondition is an error in the client. A failure of the postcondition or the class invariant is an error of the supplier.

For completeness, the listing below shows the class invariant for the VECTOR class we've been examining. When testing the VECTOR class with full assertion checking, each of the statements below is executed before and after each externally called routine in the class. This excerpt below is typical of the invariants that are utilized in this library:

```

invariant
  no_nan_items: is_nan_free
  count_strictly_positive: count >= 1
  is_zero_is_correct: is_zero implies
    (magnitude_squared = 0.0 and
     magnitude = 0.0 and two_norm =
     0.0 and
     one_norm = 0.0 and max_norm = 0.0)
  magnitude_is_non_negative:
    magnitude >= 0.0
  magnitude_squared_is_non_negative:
    magnitude_squared >= 0.0
  two_norm_is_magnitude:
    two_norm = magnitude
  one_norm_is_non_negative: one_norm >=
  0.0
  one_norm_is_large_enough: ((count = 1
    implies (one_norm + Tolerance >=
      absolute_value (item (1))))
    or else
      (one_norm + Tolerance >=
        (absolute_value (item (1)) +
         absolute_value (item
          (count))))))
  max_norm_is_non_negative: max_norm >=
  0.0
  max_norm_is_large_enough: max_norm >=
    absolute_value (item (1)) and
    max_norm >=
      absolute_value (item (count))
  max_norm_is_small_enough:
    max_norm <= one_norm
  magnitude_is_correct:
    about_equal (magnitude,
      square_root
        (magnitude_squared))
  is_unit_is_correct:
    is_unit implies
      within_range (magnitude,
        1.0 - Tolerance,
        1.0 + Tolerance)
  pound_with_self_is_magnitude_squared:
    about_equal (Current #
      Current,
      magnitude_squared)
  dot_with_self_is_magnitude_squared:
    about_equal (dot (Current),
      magnitude_squared)
  zero_is_correct: zero.is_zero
  unit_is_correct: magnitude /= 0.0
    implies unit.is_unit

```

3. Design by Contract in HPC Codes

Now that we have seen how useful assertions can be, we would like to add them into new and existing HPC codes. Since most HPC codes are not written in Eiffel (maybe they should be, but that would be a different paper), we need a way to implement assertions in the context of frequently used languages of HPC codes, C, C++, and FORTRAN. Todd Plessel, while under contract for the US Environmental Protection Agency, developed a set of tools and techniques that allow the use of DbC in FORTRAN90, C, and C++. Since these languages do not have built-in support for DbC, they lack some of the capabilities of Eiffel;

however they are expressive enough for most of the situations that are encountered in HPC codes.

To illustrate how we can add DbC to existing code, we will show a simple example that adds assertions to the Extensible Data Modeling Format (XDMF) developed by Jerry Clarke at ARL. XDMF is a C++ code for describing data formats and facilitates the linking of computational models. For clarity, we will show an original routine, and then the same routine modified with assertions. Listed below is the original routine that takes a string as input and returns the corresponding XdmfArray.

```
XdmfArray *
TagNameToArray(XdmfString TagName) {
    char c;
    XdmfInt64 i, Id;
    istrstream Tag(TagName, strlen(TagName));

    Tag >> c;
    if(c != '_') {
        XdmfErrorMessage("Invalid Array Tag
Name: " << TagName);
        return(NULL);
    }
    double d;
    Tag >> d;
    Id = (XdmfInt64)d;

    for( i = 0 ; i <
        XDMFArrayList.ListLength;
        i++){
        if (XDMFArrayList.List[i].timecntr ==
            Id){
            return(XDMFArrayList.List[i].Array);
        }
    }
    XdmfErrorMessage("No Array found with Tag
Name: " << TagName);
    return(NULL);
}
```

In order to properly utilize DbC and assertions, we need to add several utility routines to facilitate the checking of parameters to the routine. For clarity we will not list the routines, but briefly describe them here. *Has(arrayname)* returns a boolean if arrayname is in the list of arrays (an internal data structure). *XdmfArrayList.array(tag)* returns the array associated with *tag*.

```
#include <Assertions.h>
// The header file to enable DbC
...

XdmfArray *
TagNameToArray(XdmfString TagName) {
    // Use the appropriate macro for the
    // number of preconditions
    PRE3(TagName != 0,
        strlen(TagName) > 0,
        XDMFArrayList.has(TagName) );
    char c;
    XdmfInt64 i, Id;
```

```
istrstream Tag(TagName, strlen(TagName));

    Tag >> c;

    double d;
    Tag >> d;
    Id = (XdmfInt64)d;

    XdmfArray* val=XDMFArrayList.array(id);

    POST2(val != 0,
        strcmp(val->GetTagName(),
            TagName) == 0 );
    return(val);
}
```

It is possible to simplify the routine further by pushing the computation of the Id into the array routine of XDMFArrayList. Using the tools that were developed along with the header files, it is possible to create a documentation of the assertions for distribution to a client of the routine. In this way, it is possible to use information hiding and encapsulate the implementation, while still providing the client with the complete contract. Shown below is the output from *short*, a script tool that extracts comments and assertion information from the source code to create a set of documentation, similar to doxygen^[doxygen].

```
XdmfArray *
TagNameToArray(XdmfString TagName) {
    // Use the appropriate macro for the
    // number of preconditions
    PRE3(TagName != 0,
        strlen(TagName) > 0,
        XDMFArrayList.has(TagName) );
    POST2(val != 0,
        strcmp(val->GetTagName(),
            TagName) == 0 );
}
```

For completeness, we show a simple Fortran90 example with assertions.

```
! Set DENOMINATOR to NEW_DENOMINATOR.
SUBROUTINE FRACTION SET_DENOMINATOR
    (SELF, NEW_DENOMINATOR)
    TYPE(FRACTION), INTENT(OUT):: SELF

    INTEGER(KIND=8), INTENT(IN)::NEW_DENOMINATOR
    PRE(NEW_DENOMINATOR/=0_8)

    SELF%DENOMINATOR = NEW_DENOMINATOR

    POST(DENOMINATOR(SELF) ==
        NEW_DENOMINATOR)
    RETURN
END SUBROUTINE
FRACTION_SET_DENOMINATOR
```

4. Conclusion

In this paper we have provided a brief introduction to the concepts behind DbC, and showed how it is relatively

simple to add assertions to existing HPC code. Using assertions will have a remarkable effect on the quality of software being developed, and can improve the runtime performance by removing excess defensive programming code that is no longer necessary.

And since the assertions do not affect the performance of the final run-time, more elaborate types of error checking can be added. This will allow the software developer to add improved correctness information earlier in the development cycle. The earlier errors are caught in the development cycle, the easier they are to fix, which leads to our original statement: DbC is a step on the road toward quality software.

References

[doxygen] <http://www.doxygen.org>

[Hoare69] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", in *Communications of the ACM*, vol. 12, no. 10, October 1969.

[Meyer92] Meyer, B., "Eiffel: The Language", Prentice Hall, 1992.

[Meyer97] Meyer, B., *Object-Oriented Software Construction*, Second Edition, Prentice Hall, Upper Saddle River, NJ, 1997, pp. 3-19, 39-64.