Design By Contract in C++

Nicola Di Nisio

Design By Contract, a OO design technique first introduced by Eiffel, can be used even in C++. This papers introduces the Design By Contract as it is used in Eiffel and then shows how to reach comparable results in C++.

The author: Nicola Di Nisio has got a B.Sc. in Computer Science. Works as a software developer in C++ and Delphi on Windows 95/NT. www.dinisio.net/nicola , nicola@dinisio.net

Design By Contract (DBC from now on) is a design technique for Object Oriented software, first introduced with Eiffel. The aim of DBC is to ensure a higher quality for the software, a higher reliability and a greater reuse.

The key concept is that of designing a class starting from its interface and giving conditions that have to be satisfied before and after we perform each action over any object of that class. Those conditions are referred to as preconditions and postconditions. Furthermore we can provide invariant properties for the class, namely always true along the *public* life of any object of that class.

Those conditions, together with a well designed interface, should warrant that any instance of the class behaves always like we expect.

In this article we will see the support that Eiffel gives to the DBC and we will try to replicate that support in C++, as long as it is possible.

Design By Contract in Eiffel

DBC was first implemented in Eiffel, a strongly typed OO language (www.eiffel.com), or better still Eiffel was born just as an implementation tool for DBC. This language provides constructs to make

- preconditions
- postconditions
- class-invariants

A precondition is a boolean expression evaluated before the code of a method. This condition must evaluate always to *true*, otherwise an error is thrown.

A postcondition is a boolean expression evaluated after the end of the code of a method. This condition must evaluate always to *true*, otherwise an error is thrown.

A class-invariant is a condition that, each time is evaluated, it must evaluate to *true*, otherwise an error is thrown. The class invariant is evaluated immediately after the creation of each instance of the class and after each call to a public method of any instance of the class.

In Eiffel the key word **require** is to define preconditions. A precondition can be composed of many boolean expressions separated by semicolons ';'. The key word **ensure**, instead, is to define postconditions and **invariant** is to define class-invariants.

Here is a simple Eiffel class which shows those constructs at work:

```
class SquareRootable feature

value: Real;

sqrt is
  require
    value >= 0;
  do
    value := ....;
```

```
  ensure
    value >= 0;
    ((value <= old value) and (value>=1)) or
    ((value >= old value) and (value<1));
  end;

invariant
  value >= 0;

end; - class SquareRootable
```

The key word **old** in the postcondition indicates that we are referencing the values of the attribute *value* as it was before evaluating the precondition. That line of code tells us that the square root of a number is always less or equal the same number if greater of equal to 1 and viceversa if smaller. In case it was not true, then one of two cases should apply: either we have found a bug, or we have proved a new theorem :))

The class-invariant of this class is a property that expresses the necessary and sufficient condition for a number to be square-rooted without going into the complex field, that of being greater or equal to zero.

By means of a utility it is possible to extract the **short form** of that class, that is a sort of interface that omits the implementation of the methods and even private features (in the C++ terminology we would say private member data and functions), but including preconditions, postconditions and class-invariant.

The short form of the above seen class is the following:

```
class interface SquareRootable feature

value: Real;

sqrt is
  require
    value >= 0;
  ensure
    value >= 0;
    ((value <= old value) and (value>=1)) or
    ((value >= old value) and (value<1));

invariant
  value >= 0;

end; - class SquareRootable
```

It is evident that most of the information that we need to know to use an instance of SquareRootable is contained in the name of the methods (and in their public comments), in their preconditions and postconditions and in the class-invariant. These public warranties, make for us more reliable and simpler to use objects of the class SquareRootable, because being all explicit, we can always ensure that any request of any object is satisfied. All will go well until we respect this *contract*. This is why this technique is called Design By Contract.

The aim of this technique is not to catch all the cases where a object is used without respecting some conditions, assertions would suffice for that, but to make it clear *a priori* which is the agreement between the object and its client. If the designer who has developed the class has made it explicit all the constraints and the user-developer of any object of that class has respected them, then no run-time messages will be thrown saying: “warning, you would haven’t done that to me!”. Those messages are intended for the developer and only for helping him during the phase of development and debugging in catching problem and finding solutions to them (this requires the ipothesis that pre and post and class-invariants are self-explanatory). It is like documenting very well every method of the class, but unlike paper documentation, it helps us even in finding bugs, saying us where the contract has not been respected.

Using the Design By Contract in C++

C/C++ programmers know the macro *assert()* which pretends the passed expression to be true each time it is evaluated. Someone could be tempted to think that an extensive and clever use of *assert()* could help the C++ developer in using the DBC in his projects. Unfortunately this is false and for several reasons:

- Every developer of your team could use his/her own style to place assertions into the code, hence many assertions could not be steadily identifiable as preconditions or postconditions: those would simply be expressions that must be true every time the CPU passes over them. The consequence is the impossibility to get the short form of a class in an automated way.
- In general it is difficult to simulate the service offered by the Eiffel key-word **old**, in the sense that we would always store a copy, at the beginning of each method, of every thing we would like compare its initial value with its final value.
- The evaluation of the class-invariant is one thing that should be hard coded by the developer of the class and this leads to write more lines of code for each class... Furthermore it is complex, in such "free" context, to ensure that an automated process for the generation of the short forms could extract pre, post and class-invariants from the code: we should force every developer of our team in following very strong constraints (often they not even follow those that seem to be elementary, let's figure these others out....)

All these points show that the only use of *assert()* cannot take us toward the same results as the Eiffel constructs permit us to reach using the DBC. To reach comparable results we should provide some coding conventions and a object which automate many things, like the class-invariant evaluation every time it is needed and the retrieval of the old-values of every attribute of the object.

Techniques for implementing the DBC in C++

The first obvious way to provide C++ programmers a easy way to follow the DBC is that of extending the C++ language with those constructs that Eiffel has introduced for that aim. This could be done via a preprocessor which maps this enriched C++ to the standard C++. Following this way there are some drawbacks, like the huge time required to implement such a preprocessor (it is substantially a compiler for a new language), the fact that his enriched C++ would not be recognized by our smart IDEs (hence thing like the syntax highlighting would not function any more) and least but not last the loosing of portability (we should "port" even the preprocessor, otherwise we should develop and preprocess on a platform and compile and debug on another, the dream of every C++ programmer.... someone is already tired of long compiling times, figure out of what he/she could say of such a work scheme).

Another possibility is that of putting in a class the needed code to automate the definition and the evaluation of preconditions, postconditions and class-invariants, with the semantics that we have already seen. This last approach is simpler, accomplishable by a single programmer and it is the one that we are going to see in the next paragraphs. An advantage of this approach is that of being "compatible" with the integrated development environments that we all love (me for first). This last observation stands for both aesthetical and functional aspects, for example, in the case of the precompiler-approach what about the debug at the code-level? We would like to debug looking at the enriched C++ source, while we are forced to debug the precompiled code....

The Verify class

All we need to use the DBC in C++ is the class template *Verify<T>*. An instance of that class, that by convention we call *verify* (along this paper names of instances always begin by lower case, while those of classes begin by capitol letters), is created in the private section of every object of the class T and it is initialized in the constructor of T, like in the following class definition:

```
class A {
```

```
public:
  A() {
    verify.init(this,"A");
  }
private:
  Verify<A> verify;
}//class A
```

In the constructor of the class *A*, we have first to initialize the object *verify*, passing him a pointer to the host-object and the class-name of the host-object: the first it is used to call the method that evaluates the class-invariant (see later) and to make copies for the retrieval of the old-values (see later), the second is used to make each error message which *verify* throws more readable.

This is all we need to add to our classes to get the services needed for the definition and the evaluation of preconditions, postconditions and class-invariants. Later we will see how to get those services and how *Verify<T>* implements them.

Let us note that it could have been possible to provide *Verify<T>* of a constructor that took the same two parameters of *init()*, so to make it possible to create and initialize the object in one shot. The drawback of this way of doing is that the *verify* object cannot be created in a static way into the class *A* (a constructor without parameters is needed for such a creation), we are forced to allocate it in the heap and even to deallocate it manually in the destructor of *A*. Together with the increase of encoding work for us and a new source of possible bugs, this technique leads to slower code, because allocating in the heap is slower than allocating in the stack. Furthermore in this way we force the operating system to allocate to areas of memory, one for the instance of *A* and the other for *verify*. If *verify* is created statically, instead, the operating system allocates only one block of memory for the instance of *A*, which also contains *verify* and the whole process is much faster.

THE DEFINITION OF A CLASS-INVARIANT

A class invariant is a piece of code that must have access to all the members of the object, both public and private, whose aim is to compute a boolean function of the attributes of the object. We expect this function always returning the logical value *true* in normal conditions.

The best place for such a piece of code is a public method, that by convention we could name *classInvariant()*. The signature of this method shall be

```
bool classInvariant() const {...}
```

Let us note that by this definition of class-invariant follows that a class-invariant is a boolean functions, not a boolean expression, hence it goes beyond the first order logic and this observation will be useful for some considerations at the end of this paper.

The **const** qualifier is used to stress and ensure the fact that the computation of the class-invariant has no side effects on the instance on which we compute it.

To let *verify* use this class-invariant, we have to pass a pointer to this member function e we do it by the method *useClassInvariant()*, used as follows:

```
verify.useClassInvariant(&A::classInvariant);
```

This action may be accomplished in any moment after the initialization of *verify*, even if the best choice is doing it immediately after the initialization of *verify*. We need to initialize *verify* first because at this stage we pass him the pointer to the host object and this pointer is needed to call the class-invariant on the host object.

The computation of the class-invariant is made after the evaluation of every precondition and postcondition in each method declared as public. *verify* does it for us silently and later we will see how to say him when a method is public.

The automatic evaluation of the class-invariant is not made for methods declared as protected or private, because we allow an instance to temporarily violate the class-invariant after the call to a private or protected method: what matters is that this does not happen for the public methods.

Anyway it is always possible to request explicitly the computation of the class-invariant in any moment, by the method *evaluateClassInvariant()*.

The class-invariant should be called even immediately after the creation of any object and we do it by calling *evaluateClassInvariant()* explicitly at the end of the constructor of the host object.

For what we have see until now, here is a template for the constructor of the class *A*:

```
A() {
  verify.init(this,"A");
  verify.useClassInvariant(&A::classInvariant);
  ...
  ...
  verify.evaluateClassInvariant();
}
```

LET'S USE PRECONDITIONS AND POSTCONDITIONS NOW

Here is the hot part of the paper, in which we are going to see how to define preconditions and postconditions in our code.

Preconditions and postconditions are implemented via calls to the methods *preCondition()* and *postCondition()* of the object *verify*. These methods take the following parameters

1. The boolean expression to be evaluated
2. A string message to be shown on the screen in case the condition evaluates to *false*
3. The name of the method from which we are calling the precondition
4. An indicator of the type of the method (*mtPublic*, *mtProtected* or *mtPrivate*)

The parameters 1 e 2 are always mandatory, the third is mandatory for preconditions only, while the forth is intended for preconditions only and is optional (the default value is *mtPublic*). In sum we can omit to communicate to postconditions the name of the method whose postcondition is going to be evaluated and the class of this method (public, protected or private), in that those information have already been given by the previous call to *preCondition()*. Obviously we make the assumption that a precondition has been defined and this is the normal situation, even if, as we are going to see later, *Verify<T>* gives the chance to release this and others constraints, in favour of above all performance. Anyway this chance has to be used carefully.

Now let us see how is the class *SquareRootable* in C++

```
class SquareRootable {
public:
  float value;

  SquareRootable() {
    verify.init(
      this,
      "SquareRootable"
    );
    verify.useClassInvariant(&A::classInvariant);
    verify.evaluateClassInvariant();
  }//SquareRootable

  void sqrt() {
    verify.enableOld();
    verify.preCondition(
      value>=0,
      "NOT value>=0"
      "sqrt()"
    );

    value = ....;

    verify.postCondition(
      value>=0 &&
      (value>=1)?(value<=verify.old->value): (value>verify.old->value),
      "NOT value>=0 or NOT value>=1?value<=old.value: value>old.value"
```

```
    );
  }//sqrt

  bool classInvariant() const {
    return value >= 0;
  }//classInvariant

private:
  Verify<A> verify;

};//* class SquareRootable
```

We can see in the call of method *verify.postCondition()* that the old-values are retrieved by the *old* pointer, which points to a copy of the host object, as it was at the moment of the call to the method *verify.enableOld()*. This last method has made a copy of the host object and that copy is going to be destroyed at the end of the postcondition evaluation.

THE SHORT FORM OF A CLASS

As it happens in Eiffel, we can implement a program that parses our class and automatically retains only the signatures of public methods, their preconditions and postconditions, the class-invariant, public attributes and published comments, that is comments that begins, for example, with a sequence like '//*'. The output of such a program for the C++ version of *SquareRootable* should look like the following:

```
class interface SquareRootable {
  float value;

  SquareRootable();

  void sqrt()
    verify.enableOld();
    verify.preCondition(
      value>=0,
      "NOT value>=0"
      "sqrt()"
    );

    verify.postCondition(
      value>=0 &&
      (value>=1)?(value<=verify.old->value): (value>verify.old->value),
      "NOT value>=0 or NOT value>=1?value<=old.value: value>old.value"
    );

  bool classInvariant() const {
    return value >= 0;
  }

};//* class SquareRootable
```

INHERIT CLASS-INVARIANTS, PRECONDITIONS AND POSTCONDITIONS

The class-invariant of an ancestor can be inherited simply recalling it in the class-invariant of the derived class. This should always be done, in that a derived class is a specialization, hence it should satisfy all the conditions of the ancestor class, plus perhaps another set of additional conditions. An example of inherited class-invariant is in the following code:

```
class Base {
public:
  ...
  bool classInvariant() const {return ...;}
  ...
};//class Base

class Derived: public Base {
```

```
  bool classInvariant() const {
    return Base::classInvariant();
  }//classInvariant

};//class Derived
```

As regard preconditions and postconditions, inherited methods continue to be covered by preconditions and postconditions defined by the ancestor, but it is not possible to derive preconditions and postconditions in virtual methods: such conditions should entirely be rewritten in the methods of the derived classes, which overwrite the corresponding methods in the ancestor class. Alternatively, we can provide protected member functions to evaluate preconditions and postconditions of each virtual method and recall those functions both in the ancestor class and in the derived class. Let's have a look at an example of this technique:

```
class Base {
public:
  virtual void dummy() {
    verify.preCondition(
      dummyPre(),
      "dummy message for precondition",
      "dummy()"
    );
    ...
    ...
    verify.postCondition(
      dummyPost(),
      "dummy message for postcondition",
      "dummy()"
    );
  }//dummy

protected:
  bool dummyPre() {...}
  bool dummyPost() {...}
};//class Base

class Derived: public Base {
  void dummy() {
    verify.preCondition(
      Base::dummyPre() && dummyPre(),
      "dummy message for precondition",
      "dummy()"
    );
    ...
    ...
    verify.postCondition(
      Base::dummyPost() && dummyPost(),
      "dummy message for postcondition",
      "dummy()"
    );
  }//dummy

protected:
  bool dummyPre() {...}
  bool dummyPost() {...}
};//class Derived
```

THE RETRIEVAL OF THE OLD-VALUES

Unfortunately the retrieval of the old-values in this framework is not as efficient as it could be in a Eiffel implementation. In fact, to be able to retrieve the old value of each attribute, we must do a copy of the entire instance, even when we need of only the value of a couple of integer attributes... This is clearly inefficient, but it is the only simple and general way to accomplish the job and it is there, available for all those cases in which our instances are not so big and the price paid in terms of memory consumption and CPU-time is acceptable over the diminished cost of the trace activity

for bugs introduced by our hard-coded retrieval of old-values. In all other cases we have to implement the storing and the retrieval of the old-values by hand, managing even the allocation and deallocation of memory, if needed.

Another problem with this "easy way" to the retrieval of the old-values is the assumption that the copy constructor of the host object make a recursive deep-copy of the object, meaning that each pointer or reference into the host object is in turn deep-copied and so forth. If this would not be true, then we could have pointers and references copied into other pointers and references, at a certain level of the containment hierarchy, but the object the point to are always the same and when we change one of them, even the corresponding object in the copy get changed. The result is that we cannot keep track of some old-values.

To ensure that a class has a **recursive deep-copy constructor**, we must require that it makes a deep-copy of the object and that in turn all its sub-objects have a copy constructor which makes a recursive deep-copy. This is a recursive definition, that when unrolled simply means that each object contained into a host object, at every level of the containment hierarchy, must have a copy constructor which performs a deep-copy.

Even if it does not appear, the above assumption is very strong, in sense that are not rare the classes that do not satisfy that condition, in that they does not have a recursive copy-constructor. An example is given by the class *Tform* of the Borland's VCL, provided with the C++ Builder 1.0, compiler on which the code of this paper has been tested (it has been successfully tested even with the g++ version b18 for Win32 by Cygnus).

NOTES ON EFFICIENCY

As we have seen, the automation of the retrieval of the old-values can be very expensive in terms of time-space resources needed. Sometime this cost might be unacceptable, leading us to hard-code the storing and the retrieval of the old-values we need.

Other inefficiency are dued to the fact that this framework operates at a language level and for that reason no optimization can be done to decrease the computational cost of class-invariants and conditions evaluations. For example, in the C++ version of the class *SquareRootable*, the class-invariant is a factor of the conjunctive forms of the boolean expression of both the precondition and the postcondition. Being the class-invariant evaluated after the evaluation of the precondition, it is obvious that if the precondition evaluates to *true*, then even the class-invariant evaluates to *true*, hence the evaluation of the class-invariant in this case is useless, a waste of resources. The same happens for the postcondition. Unfortunately *verify* does not know this all, because it cannot analyze both the class-invariant and the pre-post conditions, it simply receives their values, *true* or *false*. Not to count the fact that the class-invariant could, in this framework, even be a boolean function of the class members, much more than a boolean expression of them, for example, it can have loops and jumps.

The optimization of the evaluation of boolean expressions is instead possible in Eiffel, because in its case it can operate at a compilation level, where it is possible to analyze the boolean expressions in both the pre-post conditions and the class-invariant (in Eiffel the class-invariant is a boolean expression).

Other considerations are to be made as regard the overhead dued to the use of *Verify*<*T*> and these have to be all investigated, above all because one of the characteristic of the C++ is that it makes never loose the control over the resources to the programmer, at contrary it helps him in controlling them at the highest degree, for the aim of the maximum efficiency.

Looking at the **List 1**, which is the source code of the file verify.h, a first consideration regards the switch *Verify_RELAX_PRE_POST_CLOSURE*, that when defined relax the constraint of defining a postcondition in a method every time that a precondition as been defined in it. Furthermore it prevents us from defining a postcondition in a method without having previously defined a precondition in the same method. The pre-post closure is just a constraint of the framework which may be relaxed when needed, for efficiency reasons, but to the detriment of the

strict adherence to the DBC paradigm (there are cases in which this could be desirable, even if they are not so many as one could think).

To disable the pre-post closure it is sufficient to define the symbol constant before including verify.h

```
#define Verify_RELAX_PRE_POST_CLOSURE
#include "verify.h"
```

Let us note that when the pre-post closure constraint is relaxed, being not mandatory to call *verify.postCondition()* at the end of the method, it is not warranted the automatic destruction of the copies of the host object made by *verify.enableOld()*, for the old-values retrieval. Such destructions should be hand made in this case, by a call to the method *verify.freeOld()*, which may be called without any problem even if the host object has not been copied.

A second switch, *Verify_ENSURE_INITIALISATION*, checks for the initialization of the *verify* object. This check is made before the copy of the host object and before or after the evaluation of both pre-post conditions and the class-invariant. This check may sense during the development and the debug phase, this is why the default condition is that it is disabled. It may be enabled simply putting a *#define* directive before the inclusion of verify.h

```
#define Verify_ENSURE_INITIALISATION
#include "verify.h"
```

Another useful switch is *Verify_INLINE*, that when defined let the methods *preCondition()*, *postCondition()* and *evaluateClassInvariant()* be in-line expanded. This three methods are rather long, but when the pre-post closure and the initialization constraint are disabled, the remainder code reduces to two comparison and two function call, or even one comparison and one function call in the case of *evaluateClassInvariant()*, hence in some cases it may sense to remove the function call overhead for these methods, suggesting to the compiler to expand them in line (yes, suggest, because at the end, it is up to Him, the compiler, to establish whether it is convenient to expand those methods in line). Too many in line expansions may lead to the so-called code-bloat, that it an explosion of the actual code compiled, dued to the in line expansion of many function calls, each one very long. At this stage the wisdom of both the programmer and his/her faithful compiler will evaluate pros and cons of each decision in its context, what that matters is that they have the power to choice...

The last switch we are going to examine is *Verify_ENSURE_POST_WITH_METHODNAME*, which imposes the name of the current method to *postCondition()*. This switch is disabled by default, because the pre-post closure is active by default and this leads to this information being passed via a previous call to *preCondition()*, at the beginning of the method. In case of relaxation of the pre-post closure, this assumption could not be true any more, and for that some sadistic team leader could say "ok, I do not force you to use always the preconditions, just because our program is too slow, but woe betide you if you do not declare the name of the method in the post conditions, I will prevent you to compile!". This is the aim of this switch...

The switches we have seen are all enablable independently one from the other, hence we could have even a situation like the following

```
#define Verify_INLINE
#define Verify_ENSURE_INITIALISATION
#define Verify_ENSURE_POST_WITH_METHODNAME
#define Verify_RELAX_PRE_POST_CLOSURE
#include "verify.h"
```

A LOOK TO THE CODE

As you have noted during the paper I have limited myself in showing only the use of the class template *Verify<T>* and I have never talked about how things are really implemented, this to stress the objective more than the implementation of *Verify<T>*, that is much less interesting. What that matters is the idea exposed in the paper, more that the class template or its implementation details.

Anyway, the source code of *Verify<T>* is given on **List 1** (verify.h), **List 2** (verify.i) and **List 3** (verify.cpp).

The **List 1** shows the interface of the class template with all its methods, that we have already seen at work. At the end of the list, verify.i is included, which is shown in the **List 2** and contains the implementation of the methods of *Verify<T>*, together with a simple smart pointer class *VerifySmartPointer<T>*, used only to export the *old* pointer in a **const** way and to prevent that when *old* is referenced before its initialization (made by *enableOld()*) a sharp "access violation" message could arise, but a smarter and more explicative

class: A
method: aMethod
old value evaluation without having previously called the method enableOld()

After all, this framework has been built up to facilitate the bug trace and it would have been absurd if it could have potentially introduced other meaningful messages like "access violation".

In the last file, verify.cpp in **List 3**, there is the implementation of the function *Verify_throwMessage()*, whose only aim is to print out error messages the more explicative as it is possible, in a portable way and in such a way that follows the conventions of the target platform. To this aim, the conditional compilation has been used, to print out messages on a dialog error when the target platform is Windows or on *cerr* on every other target platform or on the same Windows for console applications. This is a function and not a private method of the class template, because I found not correct to force to include iostream.h or windows.h together with verify.h even in code that has nothing to do with those files.

Another particularity of the source code, is the extensive use of c-style strings, that is **char***, instead of a smarter string class. The reason for this choice is that it does not exist a standard implementation for a string class in every C++ compiler (both AT&T and ANSI). Looking at the objective of *Verify<T>* and the use it makes of **char*** strings, it has not seemed to me right to introduce portability problems with the choice of a string class.

COPY CONSTRUCTION AND ASSIGNMENT

We have already talked about copy constructors for the host objects during the examination of the mechanism for the retrieval of the old-values. There are other problems that may arise when the host object does not provide a custom copy constructor, but relies on that generated by the compiler. Let us consider the case of a host class *A* and the following C++ fragment of code

```
A a;
A b(a);
```

The copy constructor for *A* generated by the compiler has constructed *b* by doing a binary copy of the object *a*, hence the pointers to the host object that *a.verify* and *b.verify* have, point to the same object, they point to *a*. The same happens if we do not define a custom assignment operator for the host class *A*, in that a default one is generated by the compiler, which makes a binary copy and in the fragment of code that follows

```
A a,b;
a=b;
```

the results are exactly the same as before. To avoid such a problem we have to provide both a custom copy constructor and a custom assignment operator, which initialize the object *verify* during its construction and assignment

```
A(const A& a) {
  verify.init(this,"A");
  ...
}
```

```
void operator=(const A& a) {
  verify.init(this,"A");
  ...
}
```

As it is not rare the possibility for someone to forget this particular, to avoid unpleasant consequences, the class *Verify<T>* has declared its own copy constructor and its own assignment operator as private members functions, preventing in this way the compiler to generate default versions for these member functions in the host classes: at this stage the compiler will require the explicit implementation of these methods, if they are used in your code.

A LITTLE PHILOSOPHICAL NOTE ON DBC

Along the paper I have stressed that the aim of DBC is not that of filling our code of assertion ready to go off at every step (the main critic that someone moves against this technique of design and development), but that of proving as many information it is possible to give to the users of instances of a class, so make them able to write more reliable applications. A signal of the fact that often a developer needs of such information is in the widespread belief that the availability of the source code of a class is a condition to use objects of that class in a safe way... this means that often developers want to look better to things, the information provided by the interface of the class and a hasty user's manual are not sufficient to prevent some problems. It is not said that the addiction of pre and post conditions to these information is sufficient, all relies on the wisdom of who writes them, but as a matter of fact, in case they result violated, they lead us in the piece of code where the problem is, hence the bug-trace problem is half solved, it suffice to execute the application in as many contexts as it is possible, waiting for dialog errors. The alternative, without this framework, is that of stopping at each step to evaluate the correctness of the numbers printed out on the screen before going on to another test and this is much slower to do that saying to a tester program "run" and see whether, when and why a dialog error comes out. Furthermore, when such a dialog error comes out, it does not say "access violation at the address 0x012AFE44", or "abnormal program termination", but a who has caused the problem, where and why and this makes a great differences for that poor programmer who has to debug the code.

CONCLUSIONS

This little framework helps the C++ developer to adhere to the DBC still using his own programming language and in a orderly fashion. This, together with the adoption of some conventions, like that of always placing a precondition as first line of code in each public method and a postcondition as last line, leads to clearer and more reliable code and above all self-documenting in terms of pre and postconditions in the class short form. Classes short form are generated by a simple program, even because the framework standardize the definition of preconditions, postconditions and class-invariants and this all facilitate the use of DBC in large C++ developing teams, where it is not simple to impose to many conventions for the coding activity...

BIBLIOGRAPHY

[1] Bertrand Meyer. *Introduction to the Eiffel language and methods*, www.eiffel.com/doc/manuals/language/intro/index.html, 1985-1998