Design by Contract in C++

Strategy and Implementation

Thomas Engelin
© Itancan Consulting AB 2003
1. Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean expression</td>
<td>A code statement that returns true or false.</td>
</tr>
<tr>
<td>C macro</td>
<td>Textual information processed by a C preprocessor.</td>
</tr>
<tr>
<td>class invariant</td>
<td>Describes a relation between state-relevant data members that must hold between client invocations.</td>
</tr>
<tr>
<td>client</td>
<td>A role taken by a software module when invoking a service on another software module.</td>
</tr>
<tr>
<td>communication party</td>
<td>A software module that communicates with another software module.</td>
</tr>
<tr>
<td>compiler flag</td>
<td>Controls how a C or C++ compiler generates executable code.</td>
</tr>
<tr>
<td>contract</td>
<td>A fictive contract between a client and a server implemented with runtime checks in the software.</td>
</tr>
<tr>
<td>defensive programming</td>
<td>A technique for writing code that is prepared to handle any normal or abnormal event in a safe way without having the software put to a halt. Can also be called maximum-tolerance design.</td>
</tr>
<tr>
<td>design by contract</td>
<td>A technique for writing code that detects the root of every fault and halts the system. Can also be called minimum-tolerance design.</td>
</tr>
<tr>
<td>exit()</td>
<td>Terminates the execution of a C or C++ program.</td>
</tr>
<tr>
<td>Firewall</td>
<td>A term for software that stops unwanted data to pass into a protected system.</td>
</tr>
<tr>
<td>fprintf()</td>
<td>Printout function in the C standard stream library.</td>
</tr>
<tr>
<td>one-definition rule</td>
<td>Every entity should be defined in only one place.</td>
</tr>
<tr>
<td>post condition</td>
<td>Specifies the server’s part of the contract.</td>
</tr>
<tr>
<td>precondition</td>
<td>Specifies the client’s part of the contract.</td>
</tr>
<tr>
<td>query</td>
<td>Returns information about an object without changing the state of the object, and that returns the same result if invoked twice.</td>
</tr>
<tr>
<td>released system</td>
<td>A system shipped to its intended end users.</td>
</tr>
<tr>
<td>server</td>
<td>A role taken by a software module when handling a request from another software module.</td>
</tr>
<tr>
<td>software module</td>
<td>A self-contained entity with a dedicated purpose. Examples include a class, a package, a component or a subsystem etc.</td>
</tr>
</tbody>
</table>
2. Summary

The intention with this document is to explain why and how Design by Contract can improve software quality, by showing a simple C++ implementation of a contract framework.

Using design by contract properly, your software will be more correct, more easily understood, and much better documented.

It is possible in C++ to mimic the built-in support for design by contract in Eiffel if we don’t set our expectations too high.

The suggested implementation of design by contract in this document is very simple, but has proven to catch numerous programming faults in several real commercial projects.

Once you get used to designing with contracts, you gain a higher scientific view of the development process, and you start to think more in terms of software correctness.
3. Problem Statement

As it always has been and still is, too many programming faults are ticking in our software. The faults are present for several reasons:

- The requirements are unspecified or vague.
- The actual design or implementation has serious flaws.
- The faults we see are not always the root of the problem, merely just symptoms.
- No attempt to prove that each software module is correct has been made.

Catching programming faults early in the design process is essential. Not only will this reduce cost for maintenance and redesign, but it will also increase the individual programmer’s confidence in his/her own software which leads to higher motivation.

And of course, a motivated programmer has a better chance to produce quality software.
4. A Piece of Solution

If we apply a technique termed Design by Contract, we can defeat the problems listed above.

On all levels of abstraction, we can set up intercommunication contracts. Communicating parties must conform to a contract, or else, a programming fault has been detected.

A contract specifies the responsibilities for each party. To explain this concept we can look at it from a client-server perspective.

A client wants to invoke a certain service on a server. To be able to invoke this service, the client must adhere to conditions set up by the server (preconditions), or else the invocation will fail, and the client has broken the contract.

If the client does follow its part of the contract, it is now up to the server to fulfill its part of the deal and deliver something that follows the contract (post conditions).

Before and after the invocation of the service, the server must guarantee to be in a valid state (class invariant). This is perhaps the most important piece of the contract.

When a contract gets broken it is easy to see the root of the fault:
1. Failed precondition: the client has made an invalid invocation of the service.
2. Failed post condition: the server has failed to deliver a correct result of the service.
3. Failed class invariant: the service execution has put the server in an invalid state.

This is how design by contract addresses the problems listed in the previous section:

- The requirements are unspecified or vague

Using contracts pushes the requirements to the front. The contract for a service is the requirement for the service. Programming with contracts means focusing on requirements.

- The actual design or implementation has serious flaws

The contracts will help you detect design faults, programming faults, logical faults, and faulty assumptions during run time.

- The faults we see are not always the root of the problem, merely just symptoms

A contract acts as a firewall between two communicating parties. Contract violations are caught immediately. No fault can propagate through the system and suddenly show up disguised as some other fault.

- No attempt to prove that each software module is correct has been made

It is hard to prove that a software module indeed is correct, but by using well-defined contracts we have the ability to express what it means for a software module to be correct.

The Eiffel programming language has built-in support for design by contract, which is one of the reasons for its rumor as the best object-oriented programming language invented.

Next, we will see how two different implementations of design by contract can be made in the C++ programming language.
5. Requirements

We can state a number of requirements for any implementation:

- It must be possible to switch off the contract mechanisms in the system.
- When a contract is broken it must be possible to see who has broken the contract.
- A contract specified by a class must be inherited by its subclasses.

These requirements are fully supported by Eiffel, but we won’t be able to completely satisfy them especially not if we aim for a simple yet decent solution.

*It must be possible to switch off the contract mechanism in the system*

We will make use of C macros so that we can switch on and off the contract mechanism by using different compiler flags. These compiler flags will be supported:

- CONTRACT_PRECONDITION_ON
  - Preconditions will be checked.
- CONTRACT_POSTCONDITION_ON
  - Post conditions will be checked.
- CONTRACT_INVARIANT_ON
  - Class invariants will be checked.
- CONTRACT_ALL_ON
  - All the above conditions will be checked.

*When a contract is broken it must be possible to see who has broken the contract*

Once again we will use C macros to indicate where in the code a contract was broken, but since we aim for a simple solution we will have a slight problem with failed preconditions.

The preconditions will be checked in the server’s code, and that means that we won’t be able to catch the violation at the invocation in the client, but only after the invocation has been made.

Thus, a failed precondition will point to a code line in the server, and it might require the use of some tracing or debugging tool to see the source of the invocation.

However, writing some inline assembly code it is possible to unwind the stack and point out the failing client’s call. This is outside the scope of this document.

*A contract specified by a class must be inherited by its subclasses*

We have now reached a point where it is very hard to mimic how contracts work in Eiffel. With our simple implementation presented next, there is no way to let a subclass inherit the contracts specified by its super class except by doing source code copy-and-paste.

Relying on copy-and-paste introduces a difficulty of keeping contracts in super- and subclasses in sync with each other during development, so some kind of contract inheritance mechanism is desired.

In chapter 7 an approach to inherit contracts is proposed. This approach works very well even though it has some drawbacks.
6. Implementation

We will put our contract implementation in a standard C++ class, called Contract. We will also use the same keywords as in Eiffel, but mapped to C macro names:

✔ A precondition check uses the keyword require in Eiffel.
✔ A post condition check uses the keyword ensure in Eiffel.
✔ A class invariant check uses the keyword invariant in Eiffel.

When the contract mechanism is switched on, the Contract macros will be mapped to a validation operation in the Contract class.

When the contract mechanism is switched off, the Contract macros will be mapped to an empty inline operation which we assume that the compiler will be able to optimize away.

For example, an integer Stack class could make use of the contract mechanism in this way (a couple of operations are missing here):

```cpp
#include "Contract.h"
Stack::Stack(int size) : _size(size)
{
    Contract::Require(valid_size, size > 0);
    _representation = new int [size];
    _topOfStack = 0;
    Contract::Invariant(ClassInvariant());
    Contract::Ensure(empty_stack, IsEmpty());
}

void Stack::Push(int item)
{
    Contract::Require(not_full_stack, !IsFull());
    _representation[_topOfStack] = item;
    _topOfStack++;
    Contract::Invariant(ClassInvariant());
    Contract::Ensure(not_empty_stack, !IsEmpty());
    Contract::Ensure(item_on_top, Top() == item);
}

bool Stack::ClassInvariant() const
{
    return (0 <= _topOfStack && _topOfStack < _size);
}
```

And so on. The preconditions and post conditions require both a label and a Boolean expression. The class invariant requires a boolean expression only.

When the contract mechanism is switched on, the boolean expression will be evaluated by the Contract class, and if it evaluates to false a contract has been broken. The labels are just for tracing purposes.

To follow the one-definition rule the class invariant expression is put into a separate query, and it is called from all non-query operations (creation and command operations).

Let’s move on to the actual Contract class. First, the header file Contract.h:
#ifndef CONTRACT_INCLUDED
#define CONTRACT_INCLUDED

#if defined(CONTRACT_PRECONDITION_ON) || defined(CONTRACT_ALL_ON)
#define Require(L,X) Check(#L, X, #X, __FILE__, __LINE__)
#else
#define Require(L,X) Noop()
#endif

#if defined(CONTRACT_POSTCONDITION_ON) || defined(CONTRACT_ALL_ON)
#define Ensure(L,X) Check(#L, X, #X, __FILE__, __LINE__)
#else
#define Ensure(L,X) Noop()
#endif

#if defined(CONTRACT_INVARIANT_ON) || defined(CONTRACT_ALL_ON)
#define Invariant(X) Check("invariant", X, #X, __FILE__, __LINE__)
#else
#define Invariant(X) Noop()
#endif

class Contract
{
public:
    static void Check(
        const char* label,
        bool expression,
        const char* expressionString,
        const char* fileName,
        long line);
    static inline void Noop() {}
};

#endif

The macros add on textual information, but also the filename and line number for the place of the call to any Contract macro. This is how the implementation file Contract.cpp could look:

```cpp
#include "Contract.h"
#include <stdio.h>
#include <stdlib.h>

void Contract::Check(
    const char* label,
    bool expression,
    const char* expressionString,
    const char* filename,
    long line)
{
    if (! expression)
    {
        fprintf(stderr, "Broken contract at %s:%ld [%s: %s]\n", 
            filename, 
            line, 
            label, 
            expressionString);
        exit(1);
    }
}
```

The calls to fprintf() and exit() could be changed to whatever suits your platform. The system should stop and the contract violation printout should be made visible.
7. To Inherit Contracts

To force contracts to be inherited without having to do any copy-and-paste, it is possible to use C++ in the following way.

✓ Have an abstract class specifying the public interface as non-virtual operations.
✓ Put all contracts into these non-virtual operations.
✓ Force subclasses to implement corresponding virtual representations of these operations.
✓ From each public interface operation, call the corresponding subclass implementation.

Consider the Stack class again. Following the guidelines above, create the abstract class that holds the public interface (most operations are left out). First, IStack.h:

```cpp
#ifndef ISTACK_INCLUDED
#define ISTACK_INCLUDED

class IStack
{
public: // Note 1
    void     Ctor(int size); // Note 2
    virtual ~IStack();
    void     Push(int item);
    int      Pop();
    int      Top() const;
    bool     IsEmpty() const;
    bool     IsFull() const;

protected:
    IStack(); // Note 3
    protected: // Note 4
    virtual void _Ctor(int size) = 0;
    virtual void _Push(int item) = 0;
    virtual int _Pop() = 0;
    virtual int _Top() const = 0;
    virtual bool _IsEmpty() const = 0;
    virtual bool _IsFull() const = 0;
    virtual bool _Invariant() const = 0; // Note 5
};
#endif
```

Some comments are needed to explain what is happening here.

Note 1: In this public section, the actual interface for objects of this class and of its subclasses is listed. These very operations will be used by clients to the class (and subclasses).

Note 2: To be able to use contracts during creation, the creation process is done in two steps. The standard default constructor is called followed by a call to this operation. This is all handled in the subclasses via a factory pattern.

Note 3: To prevent clients from bypassing the factory mechanism, the default constructor is made inaccessible to clients.

Note 4: In this protected section the actual (abstract) implementation of the class interface is listed. Subclasses do have to implement all of these operations.

Note 5: Also, a class invariant check operation has to be implemented by subclasses.

The important thing comes now; all contracts should be put in the operations listed under the public section. Let's see how this is done in IStack.cpp:
#include "IStack.h"
#include "Contract.h"

void IStack::Ctor(int size)
{
    Contract::Require(valid_size, size > 0);
    _Ctor(size);
    Contract::Invariant(_Invariant());
    Contract::Ensure(empty_stack, _IsEmpty());
}

IStack::IStack() {}
IStack::~IStack() {}

void IStack::Push(int item)
{
    Contract::Require(not_full_stack, !_IsFull());
    _Push(item);
    Contract::Invariant(_Invariant());
    Contract::Ensure(not_empty_stack, !_IsEmpty());
    Contract::Ensure(item_on_top, _Top() == item);
}

int IStack::Pop()
{
    Contract::Require(not_empty_stack, !_IsEmpty());
    return _Pop();
}

int IStack::Top() const
{
    return _Top();
}

bool IStack::IsEmpty() const { return _IsEmpty(); }

bool IStack::IsFull() const { return _IsFull(); }

There are no physical implementation details in this code. That is handed over to subclasses. Now, let's implement the obvious subclass. Here's Stack.h:

#ifndef STACK_INCLUDED
#define STACK_INCLUDED

#include "IStack.h"

class Stack : public IStack
{
public:
    static IStack* Instance(int size);

protected:
    void _Ctor(int size);
    ~Stack();
    void _Push(int item);
    int _Pop();
    int _Top() const;
    bool _IsEmpty() const;
    bool _IsFull() const;
    bool _Invariant() const;

private:
    int _size;
    int _topOfStack;
    int* _representation;
};
#endif

The only thing publicly provided here is the factory mechanism. Apart from that, the actual implementation of the IStack interface is done in a protected section. Let's look at Stack.cpp:
#include "Stack.h"

IStack* Stack::Instance(int size)
{
    IStack* s = new Stack();
    s->Ctor(size);
    return s;
}

void Stack::_Ctor(int size)
{
    _size = size;
    _topOfStack = 0;
    _representation = new int [size];
}

Stack::~Stack()
{
    delete [] _representation;
}

void Stack::_Push(int item)
{
    _representation[_topOfStack] = item;
    _topOfStack++;
}

int Stack::_Pop()
{
    _topOfStack--;
    return _representation[_topOfStack];
}

int Stack::_Top() const
{
    return _representation[_topOfStack - 1];
}

bool Stack::_IsEmpty() const
{
    return (_topOfStack == 0) ? true : false;
}

bool Stack::_IsFull() const
{
    return (_topOfStack == _size) ? true : false;
}

bool Stack::_Invariant() const
{
    return (0 <= _topOfStack && _topOfStack < _size) ? true : false;
}

Clients use the Stack class in this way:

#include "Stack.h"

...  
IStack* s = Stack::Instance(MAX_SIZE);
s->Push(42);
delete s;

We have separated contracts from implementation, and made it possible to inherit the contracts. The contracts in IStack can also be complemented with additional contracts in the subclasses.

This approach has some drawbacks apart from the more complex design:
✓ The construction needs a factory mechanism.
✓ Each new operation added by a subclass must also be added to the super class.
✓ There’s no easy way to just create a Stack on the stack (did you get that?)

My advice is to use the simpler approach in the previous chapter due to less complexity.
8. Design by Contract and Testing

When exercising the production code through a unit test framework we would like to have the ability to test the actual contracts. For example we would like to write test code such as this:

```c++
void TestCase::Run()
{
    IStack* s = Stack::Instance(MAX_SIZE);
    BREAK(int x = s->Pop());
}
```

This piece of test code is supposed to break the precondition in `IStack::Pop()`:

```c++
Contract::Require(not_empty_stack, !_IsEmpty());
```

In other words, if the contract is violated the test code works. This means that the unit test framework has to be able to handle test of contracts through the macro BREAK. But most importantly - as soon as a contract is broken the execution must immediately bail out from the tested code and be passed back to the unit test framework!

Why? Well if the execution continues after a contract has been broken we run the risk of executing code under test with totally invalid data (such as NULL pointers which will crash the unit test framework anyhow), or the code under test might be put in an invalid state.

By writing some innovative code using C macros together with setjmp() and longjmp() it is possible to create a unit test framework that can handle this very smoothly. This is outside the scope of this document but please contact the author if you are interested in such code.
9. Comments

Eiffel's contract implementation also supports the old and check keywords [1, 2]. It is pretty easy to implement support for these as well in C++ with a little help from macros.

In most Eiffel environments, there is a tool called short. This tool can extract interface information from a class definition file, ignoring the actual implementation details. In Java environments, there is a tool called javadoc, which also can extract interface information about a class by searching for special tags in the comments.

It shouldn't be too difficult to make a similar tool for C++, and in that tool incorporate functionality for extracting contract information for each operation.

Note that the contract mechanism should be seen as a debugging aid. In a released system, the contract mechanisms should be switched off.

Contracts should not be used to check for valid user input (or input from another system). In those situations, use a defensive programming approach instead. The whole purpose of contract based programming is to find design and programming faults, not to deal with faults that occur while interacting with some external entity.

A lot of improvements can be made to this example, but this simple approach actually works very well, and will definitely help you write better software, once you grasp the concept of how to express a contract between two modules.
10. References

[1] Object-Oriented Software Construction

[2] Design by Contract


A lot of information on this subject can also be found on the Internet.