



[Home](#) | [EventStudio System Designer 2.5](#) | [VisualEther Protocol Analyzer 1.0](#) | [Real-time Mantra](#) | [Cor](#)
[Home](#) > [Real-time Mantra](#) > [Object Oriented Design](#) > Design by Contract Programming in C++

Design by Contract Programming in C++

The Eiffel programming language introduced "design by contract" to object oriented programming to model interfaces between classes as contracts. In this article, we will be applying this powerful programming.

Interfaces as Contracts

In legal terms, a contract is a binding document that describes the responsibilities and expectations entering into the contract. Interfaces between classes can be modeled in the same way. When a method of an other object, this interaction should be viewed as a contract between the caller and the object. A contract consists of the following conditions:

- **Pre-conditions.** The caller of the method needs to pass parameters that meet the requirements of the method. Pre-conditions check if the caller of the method is keeping its side of the contract parameters.
- **Post-conditions.** The called method needs to return results that meet the expectations of the caller. Post-conditions check if the called method is keeping its side of the contract, i.e. returning the results of the interface.
- **Consistency checks.** In addition to the pre-conditions and post-conditions, all the objects involved in a transaction should be left in a consistent state.

Design by Contract Framework

We will consider an example here to see how "design by contract" is implemented in C++. We will present a basic framework for design by contract. The framework consists of the following macros that are only active in debug mode. The macros are defined to blank in the release build.

- **ASSERT:** This macro aborts the program if the parameter passed to the macro does not evaluate to true. If this macro is used in the code, the programmer is asserting that the condition specified in the macro is true. If it is not so, further execution of the program cannot proceed.
- **IS_VALID:** This macro is used to check the consistency of an object. The programmer invokes this macro acting on an object pointer. This macro invokes the virtual function IsValid to perform the consistency check associated with the class.
- **REQUIRE:** This macro should be used to check if the pre-conditions for the invoked method are met. If the caller does not keep its part of the contract, this macro will assert (as it is just another red flag).
- **ENSURE:** This macro checks if the post conditions have been met. Here the called function checks its part of the contract. This macro will assert if there is a contract breach.
- **IsValid Method:** This method is defined in a base class that will act as the parent for all classes. The method is defined as an abstract function. This forces all programmers to define the IsValid method in their inheriting classes should check for consistency of the objects internal state. Also note that this macro is only active in the debug build, thus it does not add to code bloat.

Design by Contract Framework

```
// Object class is the base class for all
// objects in the system. All classes inheriting from this class need
// to define a method IsValid. This method should perform a
// consistency check on the state of the object. Note that
// this method needs to be defined only when a debug build is made
class Object
{
public:
#ifdef _DEBUG
    bool IsValid() const = 0;
#endif
};

#ifdef _DEBUG

// The debug mode also defines the following macros. Failure of any of these macros
// program termination. The user is notified of the error condition with the right f
// and line number. The actual failing operation is also printed using the stringizi

#define ASSERT(bool_expression) if (!(bool_expression)) abort_program(__FILE__, __LI
#define IS_VALID(obj) ASSERT((obj) != NULL && (obj)->IsValid())
#define REQUIRE(bool_expression) ASSERT(bool_expression)
#define ENSURE(bool_expression) ASSERT(bool_expression)

#else

// When built in release mode, the _DEBUG flag would not be defined, thus there will
// in the final release from these checks.

#define ASSERT(ignore) ((void) 0)
#define IS_VALID(ignore) ((void) 0)
#define REQUIRE(ignore) ((void) 0)
#define ENSURE(ignore) ((void) 0)

#endif
```

Debug and Release Builds

The "design by contract" framework can work only if the programmers can introduce aggressive worrying about their performance implications. The main idea here is that lab testing of the proc debug builds (i.e. `_DEBUG` flag is defined) where all the "design by contract" macros are enable allow you to zero in on the faults very quickly as all breach of contract conditions are being che dramatically lower the debugging time in a large project. This means you will have less of those sessions when bugs have to be isolated.

When the product is ready to be shipped, the release build can be made. This build will disable the "design by contract" framework. Thus you will obtain complete performance. If CPU perform the initial deployment you may decide to retain the macros and replace the exit condition in the exception throwing. Thus you have complete control on the level of debugging you wish to have

An important thing to note here is that the "design by contract" framework is not a replacement programming. You still write defensive code which handles error conditions even in the release contract checking as a diagnostic programming technique that allows you to be extra suspiciou. when the implementation of the contract is new and untested. When you get comfortable with tl implementation, you turn off the extra suspicious checking.

Another benefit of "design by contract" technique is that it gives you extra diagnostic capability & readability. There is no need to add redundant if statements for diagnostic programming. In fact they actually improve the readability as they clearly state the expectations of the caller and the callee.

An Example of Design by Contract Programming

Here we have taken an example from the [STL Design Patterns](#) article and added support for the framework. All the additions to the original [Terminal Manager](#) are shown in bold.

Terminal Manager

```
#include <map>           // Header file include for map
using std;              // STL containers are defined in std namespace

class TerminalManager : public Object
{
    // The map is keyed with the terminal id and stores pointers to Terminals.
    // terminal id is an integer, terminal ids can be in the entire range for
    // an integer and they will still be efficiently stored inside a map.
    typedef map<int, Terminal *> TerminalMap;
    TerminalMap m_terminalMap;
    int m_managerType;
    FaultManager m_faultManager;

public:

#ifdef _DEBUG
    // IsValid methods play an important role in checking the consistency
    // of objects in the debug. IsValid is defined as a pure virtual function
    // in Object class, thus it needs to be overridden in all inheriting classes.
    // The inheriting class should perform defensive checks to make
    // sure that it is in a consistent state/
    // Also note that this method is only available in the debug build.
    virtual bool IsValid() const
    {
        return (m_terminalMap.count() <= MAX_TERMINALS_PER_MANAGER &&
                m_managerType < MAX_MANAGER_TYPES &&
                m_faultManager.IsValid());
    }
#endif

    Status AddTerminal(int terminalId, int type)
    {
        // Checking Preconditions
        REQUIRE(terminalId < MAX_TERMINAL_ID);
        REQUIRE(type >= TERMINAL_TYPE_RANGE_MIN && type <= TERMINAL_TYPE_RANGE_MAX);

        Status status;

        // Check if the terminal is already present in the map. count()
        // returns the total number of entries that are keyed by terminalId
        if(m_terminalMap.count(terminalId) == 0)
        {
            // count() returned zero, so no entries are present in the map
            Terminal *pTerm = new Terminal(terminalId, type);

            // Make sure that the newly created terminal is in consistent state
            IS_VALID(pTerm);

            // Since map overloads the array operator [ ], it gives
```

```

    // the illusion of indexing into an array. The following
    // line makes an entry into the map
    m_terminalMap[termId] = pTerm;

    status = SUCCESS;
}
else
{
    // count() returned a non zero value, so the terminal is already
    // present.
    status = FAILURE;
}

// Checking post conditions:
// 1. TerminalManager should be consistent
// 2. The new terminal should always be found
// 3. The manager should not be controlling more terminals
//    than allowed
// 4. Make sure correct return code is being returned.
IS_VALID(this);
ENSURE(FindTerminal(termId));
ENSURE(m_terminalMap.count() <= MAX_TERMINALS_PER_MANAGER);
ENSURE(status == SUCCESS || status == FAILURE);
return status;
}

Status RemoveTerminal(int terminalId)
{
    // Check pre-conditions
    // Note: Here the REQUIRE macro makes sure that
    // terminal to be deleted is actually present. A similar
    // check will be done in the main body of the code.
    // The duplicate check in the REQUIRE macro allows flagging
    // the error earlier.

    REQUIRE(terminalId < MAX_TERMINAL_ID);
    REQUIRE(FindTerminal(terminalId));

    Status status;
    // Check if the terminal is present
    if (m_terminalMap.count(terminalId) == 1)
    {
        // Save the pointer that is being deleted from the map
        Terminal *pTerm = m_terminalMap[terminalId];

        // Make sure that terminal object being deleted is in a consistent
        // state
        IS_VALID(pTerm);

        // Erase the entry from the map. This just frees up the memory for
        // the pointer. The actual object is freed up using delete
        m_terminalMap.erase(terminalId);
        delete pTerm;

        status = SUCCESS;
    }
    else
    {
        status = FAILURE;
    }

    // Checking Post-conditions:
    // 1. Terminal has been successfully deleted (terminal find

```

```
        //    should return NULL)
        // 2. Only valid status should be returned.
        // 3. Terminal Manager is in a consistent state
        ENSURE(FindTerminal(terminalId) == NULL);
        ENSURE(status == SUCCESS || status == FAILURE);
        IS_VALID(this);

        return status;
    }

// Find the terminal for a given terminal id, return
// NULL if terminal not found
Terminal *FindTerminal(int terminalId)
{
    Terminal *pTerm;
    if (m_terminalMap.count(terminalId) == 1)
    {
        pTerm = m_terminalMap[terminalId];
    }
    else
    {
        pTerm = NULL;
    }

    return pTerm;
}

void HandleMessage(const Message *pMsg)
{
    // Check pre-conditions:
    IS_VALID(pMsg);
    ENSURE(FindTerminal(pMsg->GetTerminal()));

    int terminalId = pMsg->GetTerminalId();

    Terminal *pTerm;

    pTerm = FindTerminal(terminalId);

    if (pTerm)
    {
        pTerm->HandleMessage(pMsg);
    }
}
};
```