

# Design By Contract in C++

*Nicola Di Nisio*

Il Design By Contract è una tecnica di design del software Object Oriented, introdotta con Eiffel, il cui scopo è quello di assicurare una maggiore qualità del software, una maggiore affidabilità dello stesso ed un suo maggior riuso.

Il concetto chiave è quello di disegnare una classe partendo dalla sua interfaccia e dando condizioni che debbono essere soddisfatte prima e dopo che ogni azione su un oggetto di quella classe venga compiuto (precondizioni e postcondizioni). Oltre a ciò si possono dare proprietà invarianti per la classe, cioè sempre vere durante la vita di ogni istanza di quella classe.

Queste condizioni, insieme ad una interfaccia ben fatta, dovrebbero garantire che ogni istanza della classe si comporti sempre come noi ci aspettiamo.

In questo articolo vedremo il supporto che Eiffel dà al Design By Contract e cercheremo di replicarlo in C++, per quanto possibile.

## DESIGN BY CONTRACT IN EIFFEL

Il Design By Contract (d'ora in poi DBC) è stato per la prima volta implementato in Eiffel, un linguaggio OO fortemente tipato (vedi [www.eiffel.com](http://www.eiffel.com)). Questo linguaggio offre costrutti per realizzare

- precondizioni
- postcondizioni
- invarianti di classe

Una precondizione è una espressione booleana valutata prima del codice di un metodo. Questa deve sempre essere vera, altrimenti viene emessa una segnalazione di errore all'utente.

Una postcondizione è una espressione booleana valutata al termine di un metodo. Questa deve sempre essere vera, altrimenti viene emessa una segnalazione di errore all'utente.

Un invariante di classe è una condizione che, ogni volta che viene valutata, deve valere *true*, altrimenti viene emessa una segnalazione di errore all'utente. L'invariante di classe viene valutato dopo la creazione di ogni istanza della classe e dopo ogni chiamata di ogni metodo pubblico su un oggetto della classe.

In Eiffel una precondizione è introdotta a mezzo della parola chiave **require** e può comporsi di una o più espressioni booleane separate da ‘;’. Una postcondizione è introdotta a mezzo della parola chiave **ensure** ed un invariante di classe a mezzo della parola chiave **invariant**.

Ecco una semplice classe Eiffel che mostra questi costrutti all'opera:

```
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

```

Notate la parola chiave **old** nella postcondizione: essa indica che stiamo referenziando il valore dell'attributo *value* come esso era prima di valutare la preconditione. Quella linea di codice ci dice che la radice quadrata di un numero è sempre minore od uguale del numero stesso se il numero è maggiore od uguale ad 1 e viceversa se è minore. Se ciò non dovesse essere vero, allora sono due i casi: o abbiamo beccato un baco, oppure abbiamo dimostrato un nuovo teorema :))

L'invariante di classe è una proprietà che esprime la condizione necessaria e sufficiente per un numero affinché sia elevabile a radice quadrata senza andare nel campo complesso, cioè quella di essere maggiore od uguale a zero.

Con una apposita utility è possibile estrarre la **short form** di quella classe, cioè una sorta di interfaccia che omette l'implementazione dei metodi e non riporta per niente le feature private (in terminologia C++ diremmo dati e funzioni membro private), ma include preconditioni, postcondizioni ed invariante di classe. La short form della classe che abbiamo appena visto è la seguente:

```

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

```

Ora è evidente il fatto che la maggior parte delle informazioni da sapere per usare un oggetto SquareRootable sono contenute nel nome dei metodi (e nei loro commenti pubblici) e nelle pre e postcondizioni, oltre che nell'invariante di classe. Queste garanzie pubbliche, rendono a noi più facile ed affidabile il compito di usare gli oggetti della classe SquareRootable, perché essendo tutto esplicito, possiamo far sì che le richieste di tali oggetti siano sempre soddisfatte. Tutto andrà bene finché noi rispetteremo questo *contratto*. Questo è il motivo per cui tale tecnica si chiama Design By Contract.

Il punto saliente di tale tecnica non è che becca i casi in cui gli oggetti vengono usati fuori dal loro campo di applicabilità, basterebbero le asserzioni per questo, bensì che mette bene in chiaro *a priori* quali sono i patti che devono intercorrere tra l'oggetto ed il suo client. Se il designer della classe ha esplicitato tutti i vincoli e lo sviluppatore che usa gli oggetti di tale classe li ha rispettati tutti ed in ogni caso, allora non usciranno fuori messaggi a run-time che ci dicono: "attenzione, questo non me lo dovevi fare". Questi messaggi servono solo al programmatore in fase di sviluppo e debug per aiutarlo a trovare i problemi e a dargli un suggerimento per la loro soluzione (ciò presuppone che pre e post ed invarianti di classe siano autoesplicative). È un po' come documentare molto bene ogni metodo della classe, ma a differenza di semplice documentazione cartacea essa ci aiuta anche a trovare i bachi, indicandoci i punti dove il contratto non è stato rispettato.

## USARE IL DESIGN BY CONTRACT IN C++

I programmatori C/C++ ben conoscono la macro *assert()*, la quale pretende che l'espressione passata risulti vera ogni volta che viene valutata. Si potrebbe pensare che un uso estensivo ed intelligente di *assert()* possa aiutare lo sviluppatore C++ a progettare classi secondo i canoni del DBC. Sfortunatamente questo è falso e per diverse ragioni:

- Ogni programmatore del vostro team potrebbe adottare il suo proprio stile per piazzare asserzioni all'interno del suo codice, quindi molte asserzioni potrebbero non essere immediatamente identificabili come precondizioni o postcondizioni di un metodo: esse sono semplicemente cose che devono essere vere ogni volta che la CPU ci passa sopra. La conseguenza è l'impossibilità di ottenere la short form della classe in maniera automatizzata.
- In generale è complesso simulare il servizio offerto dalla parola chiave **old** di Eiffel, nel senso che noi dovremmo immagazzinare una copia, all'inizio del metodo, di ogni cosa di cui vogliamo poter poi poter confrontare il valore finale con quello iniziale.
- La valutazione dell'invariante di classe è una cosa che dovrebbe essere implementata a mano dal designer della classe e questo porta a scrivere più linee di codice per ogni classe... Inoltre è complesso in un tale contesto "libero" assicurare che un processo automatico per la generazione della short form possa estrarre tali condizioni ed invarianti di classe dal codice: dovremmo obbligare ogni programmatore del nostro team a seguire convenzioni molto rigide (spesso già non seguono quelle elementari, figuriamoci le altre...)

Ora è chiaro che il solo uso di *assert()* non può portare agli stessi risultati cui possono portare i costrutti offerti da Eiffel nell'ambito del DBC.

Per ottenere risultati paragonabili dobbiamo fornire alcune convenzioni di codifica ed un oggetto che automatizzi molte cose, come la valutazione dell'invariante di classe ogni volta che è necessario ed il recupero degli old-values di ogni attributo dell'oggetto.

## TECNICHE PER IMPLEMENTARE IL DBC IN C++

La prima ovvia strada per dare ai programmatori C++ la possibilità di seguire in modo semplice lo stile di progettazione e codifica dettato dal DBC è quello di estendere il linguaggio con quei costrutti che Eiffel ha introdotto a tale scopo.

Questo si potrebbe fare attraverso un preprocessore che mappi da questo C++ "arricchito" al C++ standard. Seguendo questa via si hanno degli svantaggi, come l'eccessivo tempo necessario per implementare un tale preprocessore (sostanzialmente un compilatore per un nuovo linguaggio), il fatto che questo C++ drogato non sarebbe riconosciuto dai nostri IDE intelligenti (quindi cose come l'evidenziazione della sintassi andrebbero a pallino), e per ultimo la perdita di portabilità (dovremmo "portare" anche il preprocessore, altrimenti si sviluppa e precompila su una piattaforma e si compila, esegue e debugga su un'altra, il sogno di ogni sviluppatore C++... qualcuno già si lamenta dei lunghi tempi di compilazione, figuriamoci cosa direbbe di questo schema di lavoro)

Un'altra strada è quella di mettere in una classe il codice necessario per automatizzare la definizione e la valutazione di precondizioni, postcondizioni ed invarianti di classe, con le caratteristiche e la semantica che abbiamo già visto. Questo secondo approccio è più semplice, alla portata di un singolo programmatore, ed è quello che andremo a vedere nel prosieguo dell'articolo. Tra l'altro ha l'indubbio vantaggio di essere "compatibile" con gli ambienti di sviluppo integrati cui noi tutti (io per primo) non vogliamo rinunciare. Quest'ultima osservazione vale sia per quanto riguarda gli aspetti estetici che funzionali, ad esempio, nel caso del precompilatore, come la mettiamo con il debugging del codice? Noi vorremmo debuggare a livello di C++ drogato, mentre ci ritroveremmo a debuggare il codice precompilato...

## LA CLASSE VERIFY

Tutto ciò che serve per sviluppare con il DBC in C++ è una classe template, che qui chiameremo *Verify<T>*. Un'istanza di questa classe, che per convenzione chiameremo *verify*

(nell'articolo i nomi di istanza iniziano per lettera minuscola, quelli di classe per maiuscola), viene creata nella sezione privata di ogni oggetto di classe *T* ed inizializzata nel costruttore di *T*, come nel seguente esempio di definizione di classe

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

Dentro al costruttore della classe *A* la prima cosa che dobbiamo fare è inizializzare l'oggetto *verify*, passandogli un puntatore all'oggetto ospite (quello che lo contiene) ed il nome della classe dell'oggetto ospite: il primo è usato per richiamare il metodo che calcola l'invariante di classe (vedi dopo) e per fare copie per il recupero degli old-values (vedi dopo), il secondo è usato per rendere ogni segnalazione di errore di *verify* più leggibile.

Una nota a margine per quanto riguarda la dizione "oggetto ospite". Qui l'attributo ospite è usato nel senso di "colui che ospita", cioè lo stesso significato che ha la sua controparte anglosassone "host". In Italiano, tuttavia, lo stesso termine è usato anche nel senso opposto, ad indicare ciò che è ospitato, ed anzi questo è l'uso più comune del termine. In ogni caso entrambe le accezioni sono valide, ma nel prosieguo dell'articolo con "oggetto ospite" si indicherà l'oggetto che contiene, non l'oggetto contenuto, così come indica l'analoga dizione inglese "host object".

Questo è tutto quello che abbiamo bisogno di aggiungere alle nostre classi per ottenere i servizi necessari alla definizione ed alla valutazione di precondizioni, postcondizioni ed invarianti di classe. Più tardi vedremo come si ottengono tali servizi e come *Verify<T>* li implementa.

Notiamo subito che si sarebbe potuto fornire a *Verify<T>* un costruttore che prendeva gli stessi due parametri di *init()*, in modo da creare ed inizializzare l'oggetto con un solo colpo. Lo svantaggio di tale opzione è che in tale modo l'oggetto *verify* non lo si sarebbe potuto creare in modo statico all'interno della classe *A* (è necessario un costruttore senza parametri per fare ciò), lo si sarebbe dovuto allocare nell'heap e quindi deallocare a mano, nel distruttore di *A*. Oltre all'aumento di lavoro da parte nostra ed a costituire una possibile nuova fonte di problemi, ciò avrebbe anche rallentato il processo di generazione di istanze di *A*, perché allocare nell'heap è più costoso, in termini di tempo. Tra l'altro avremmo costretto il sistema operativo ad allocare due aree di memoria distinte, una per l'istanza di *A* ed una per la sua istanza di *Verify<A>*, invece nella soluzione scelta il sistema operativo compie una sola allocazione di memoria, che alla bisogna (se vogliamo la massima velocità) può essere interamente riservata nello stack (quando l'istanza di *A* è allocata nello stack), mentre allocando l'istanza di *Verify<A>* nell'heap i vantaggi di allocare le istanze di *A* nello stack sarebbero stati in parte vanificati.

## LA DEFINIZIONE DI UN INVARIANTE DI CLASSE

Un invariante di classe è un pezzo di codice che deve avere accesso a tutti i membri dell'oggetto, sia pubblici che privati, e che deve calcolare una certa funzione booleana degli attributi. Tale funzione deve sempre ritornare il valore logico true.

Un ottimo posto per tale codice è un metodo pubblico di istanza, che per convenzione potremo chiamare *classInvariant()*. La firma di tale metodo sarà la seguente

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

Esso deve ritornare il valore logico risultato della funzione booleana dei membri dell'oggetto che è l'invariante di classe. Si noti subito che da questa definizione discende che l'invariante è

una funzione booleana, non un'espressione booleana, quindi trascende i limiti della logica del prim'ordine, questo ci servirà per delle considerazioni al termine dell'articolo.

Il qualificatore **const** serve a sottolineare e ad assicurare il fatto che la computazione dell'invariante di classe non ha effetti collaterali sull'istanza che la subisce.

Per far usare a *verify* questo invariante di classe dobbiamo passargli un puntatore a questa funzione membro e lo facciamo a mezzo del metodo *Verify<T>::useClassInvariant()* usato come segue

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

Tale azione può essere compiuta in ogni momento posteriore all'inizializzazione di *verify*, anche se la cosa più logica è che la si faccia immediatamente dopo l'inizializzazione di *verify*. La necessità di inizializzare prima *verify* nasce dal fatto che per richiamare l'invariante di classe sull'oggetto ospite, *verify* ha bisogno di un puntatore allo stesso, che ottiene solo al momento dell'inizializzazione.

Il calcolo dell'invariante di classe avviene di norma dopo la valutazione di ogni preconditione e postcondizione di un metodo pubblico. È *verify* a preoccuparsene per noi e vedremo poi come comunicargli quando un metodo è pubblico.

La valutazione automatica dell'invariante di classe non è implementata per i metodi dichiarati come protetti o privati, perché si suppone che un'istanza possa temporaneamente violare l'invariante dopo la chiamata di un metodo privato o protetto, l'importante è che non accada per i metodi pubblici. Ad ogni modo, è sempre possibile richiedere manualmente il calcolo dell'invariante di classe a mezzo del metodo *Verify<T>::evaluateClassInvariant()*.

Di norma l'invariante di classe andrebbe valutato anche immediatamente dopo la creazione dell'oggetto ospite e questo lo facciamo richiamando esplicitamente *evaluateClassInvariant()* al termine del processo di creazione.

A questo punto un modello per il costruttore della classe A si presenta così

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

## ORA USIAMO LE PRECONDIZIONI E LE POSTCONDIZIONI

Siamo arrivati nella zona calda dell'articolo dove vedremo come definire preconditioni e postcondizioni nel nostro codice.

Precondizioni e postcondizioni sono implementate attraverso chiamate ad i metodi *preCondition()* e *postCondition()* dell'oggetto *verify*. Questi metodi prendono i seguenti parametri

1. L'espressione booleana da valutare
2. Un messaggio da mostrare a video in caso di valutazione a false dell'espressione
3. Il nome del metodo nel quale si sta richiamando la preconditione
4. Un indicatore del tipo di metodo (*mtPublic*, *mtProtected* od *mtPrivate*)

I parametri 1 e 2 sono sempre obbligatori, il terzo lo è solo per le preconditioni, mentre il quarto si usa solo per le preconditioni ed è opzionale (il valore assunto è *mtPublic*). In sostanza alla postcondizione si possono non comunicare il nome del metodo di cui sta valutando la postcondizione e di che tipo di metodo si tratta (pubblico, protetto o privato), in quanto tali informazioni sono già state passate attraverso la definizione della preconditione immediatamente precedente. Naturalmente si fa l'assunzione che la preconditione sia stata definita e questa è la

situazione normale, anche se, come vedremo in seguito, *Verify*<*T*> dà la possibilità di rilasciare questo ed altri vincoli, a beneficio soprattutto delle prestazioni, possibilità da usarsi con cautela.

Ora vediamo come viene la classe *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
```

Come si può osservare, nel metodo *verify.postCondition()* gli old-values sono recuperati a mezzo del puntatore *old*, che punta ad una copia dell'oggetto ospite così come esso era al momento della chiamata del metodo *verify.enableOld()*. È quest'ultimo metodo che si è occupato di fare una copia dell'oggetto ospite e tale copia viene distrutta al termine della valutazione della postcondizione.

## LA SHORT FORM DELLA CLASSE

Così come avviene in Eiffel, ora possiamo realizzare un programma che scandisce la nostra classe e che automaticamente mantiene solo le firme dei metodi pubblici, le loro precondizioni e postcondizioni, l'invariante di classe e gli attributi pubblici, oltre che i commenti pubblicati, quelli cioè che iniziano, ad esempio, con una sequenza del tipo */\*\**. L'output di un tale programma per la versione C++ di *SquareRootable* dovrebbe essere qualcosa di molto simile a ciò che segue:

```
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

```

## EREDITARE INVARIANTI DI CLASSE, PRECONDIZIONI E POSTCONDIZIONI

L'invariante di classe di un antenato può essere ereditato semplicemente richiamandolo nell'invariante di classe della classe derivata. Anzi, questo dovrebbe sempre essere fatto, in quanto una classe derivata è una specializzazione, quindi deve soddisfare tutte le condizioni della classe antenata, più eventualmente un altro insieme di condizioni aggiuntive. Un esempio di invariante di classe ereditato è nel seguente frammento di codice:

```

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

class Derived: public Base {
  bool classInvariant() const {
    return Base::classInvariant();
  }/*classInvariant

};/*class Derived

```

Per quanto riguarda precondizioni e postcondizioni, i metodi ereditati continuano ad essere coperti dalle precondizioni e dalle postcondizioni definite nell'antenato, ma non è possibile derivare precondizioni e postcondizioni in metodi virtuali: tali condizioni devono essere interamente riscritte nei metodi delle classi derivate, che sovrascrivono i relativi metodi della classe antenata. Alternativamente possiamo fornire funzioni membro protette per valutare precondizioni e postcondizioni di ogni metodo virtuale e richiamare tali funzioni sia nei metodi della classe antenata che in quelli della classe base. Vediamo un esempio di come si possa fare tutto ciò:

```

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

```

## IL RECUPERO DEGLI OLD-VALUES

Sfortunatamente il recupero degli old-values in questo framework non è così efficiente come può esserlo in una implementazione di Eiffel. Infatti per essere in grado di recuperare gli old-values di ogni attributo, dobbiamo fare una copia dell'intera istanza, quando magari abbiamo bisogno solo del valore di un paio di variabili intere... Ciò è chiaramente inefficiente, ma è la sola via semplice e generale per farlo ed è lì disponibile per tutti quei casi in cui le nostre istanze non sono troppo grandi ed il prezzo pagato in termini di occupazione di memoria e CPU-time è accettabile in rapporto al diminuito costo del tracciamento di banchi dovuto al fatto che non abbiamo dovuto scrivere codice aggiuntivo per l'immagazzinamento ed il recupero degli old-values. In tutti gli altri casi dobbiamo implementare immagazzinamento e recupero degli old-values a mano, gestendo anche l'eventuale allocazione e la deallocazione della memoria necessaria per tali oggetti.

Un'altro problema insito in questa "via facile" al recupero degli old-values è l'assunzione che il copy constructor dell'oggetto ospite faccia una deep-copy ricorsiva dell'oggetto, intendendo che ogni puntatore o reference all'interno dell'oggetto ospite sia a sua volta oggetto di una deep-copy e così via. Se ciò non fosse vero, allora potremmo avere puntatori e reference copiati in altri puntatori e reference, ad un certo livello della gerarchia di contenimento, ma gli oggetti cui puntano sono sempre gli stessi e quando noi cambiamo uno di questi, anche il relativo oggetto

nella copia sarà cambiato. La conseguenza è che non possiamo più tener traccia di alcuni degli old-values.

Per assicurare che una classe abbia un **deep-copy constructor ricorsivo**, dobbiamo richiedere che esso faccia una deep-copy dell'oggetto e che a loro volta tutti gli oggetti in esso contenuti abbiano un copy constructor che opera una deep-copy ricorsiva. Questa è una definizione ricorsiva, che quando viene *srotolata* significa semplicemente che ogni oggetto contenuto dentro l'oggetto ospite, ad ogni livello della gerarchia di contenimento, deve avere un copy constructor che opera una deep-copy.

Per quanto possa sembrare strano, l'assunzione poc'anzi fatta è molto forte, nel senso che non sono rare le classi che non la soddisfano, cioè che non hanno un deep-copy constructor ricorsivo. Un esempio è dato dalla classe *TForm* della VCL di Borland, fornita con il C++ Builder 1.0, compilatore su cui il codice fornito insieme all'articolo è stato testato (è stato testato con successo anche sul g++ in versione b18 per Win32 della Cygnus).

## NOTE SULL'EFFICIENZA

Come abbiamo visto, l'automazione del recupero degli old-values può essere molto dispendiosa in termini di risorse spazio-temporali necessarie. A volte tale costo può essere inaccettabile, portandoci a scrivere a mano il codice necessario all'immagazzinamento ed al recupero degli old-values.

Altre inefficienze sono dovute al fatto che questo framework opera a livello di linguaggio, e per questo nessuna ottimizzazione può essere operata per ridurre il costo computazionale del calcolo degli invarianti e delle condizioni. Ad esempio, nella versione C++ della classe *SquareRootable*, l'invariante di classe è un fattore della forma congiuntiva dell'espressione booleana presente sia nella preconditione che nella postcondizione. Dato che l'invariante di classe viene valutato dopo la valutazione della preconditione è ovvio che se la preconditione vale true, allora anche l'invariante varrà true, quindi la valutazione di quest'ultimo è inutile, in questo caso. Lo stesso accade per la postcondizione. Sfortunatamente *verify* non sa tutto questo, perché esso non può analizzare le condizioni e l'invariante di classe, egli riceve semplicemente un true od un false. Senza contare che l'invariante di classe può, in questo framework, essere una funzione booleana dei membri dell'oggetto ospite, molto più che una semplice espressione booleana degli stessi (ad esempio può includere cicli e salti).

L'ottimizzazione della valutazione delle espressioni booleane è invece possibile in Eiffel, perché nel suo caso si opera a livello di compilazione, dove è possibile analizzare le espressioni booleane delle condizioni e dell'invariante di classe (in Eiffel l'invariante di classe è sempre e solo una espressione booleana).

Ci sono altre considerazioni da fare riguardo all'overhead dovuto all'uso di *Verify<T>* e queste vanno tutte investigate, soprattutto perché una delle caratteristiche del C++ è quella di non far perdere al programmatore il controllo delle risorse, ma anzi di consentirgli di controllarle al massimo grado, ai fini del raggiungimento della massima efficienza possibile.

Facendo riferimento al **Listato 1**, che riporta il codice sorgente del file *verify.h*, una prima considerazione riguarda lo swich *Verify\_RELAX\_PRE\_POST\_CLOSURE*, che quando definito rilascia il vincolo di mettere una postcondizione ogni volta che si è definita una preconditione in un metodo ed inoltre impedisce di mettere una postcondizione senza aver prima definito una preconditione per lo stesso metodo. La chiusura pre-post è proprio un vincolo del framework che all'occorrenza può essere rilasciato, magari per motivi di efficienza, naturalmente a scapito della stretta aderenza al paradigma del DBC (ci sono casi in cui può essere desiderabile, anche se non sono così tanti come si potrebbe pensare). Per disabilitare la chiusura pre-post basta definire la costante simbolica prima di includere *verify.h*

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

Si noti che se il vincolo di chiusura pre-post viene rilasciato, allora non essendo obbligatorio il richiamo di `verify.postCondition()`, non è garantita la distruzione automatica della copia fatta da `verify.enableOld()` dell'oggetto ospite ai fini del recupero degli old values. Tale distruzione va operata in modo manuale attraverso il metodo `Verify<T>::freeOld()`, che può essere richiamato senza problemi anche se non è stato precedentemente copiato l'oggetto ospite con `enableOld()`.

Un secondo switch, `Verify_ENSURE_INITIALISATION`, controlla che l'oggetto `verify` sia inizializzato. Tale controllo è operato prima della copia dell'oggetto ospite ai fini del recupero degli old-values (`verify.enableOld()`) e prima o dopo la valutazione delle condizioni e dell'invariante di classe. Tale controllo ha principalmente senso durante lo sviluppo ed il debug del codice, per questo la condizione di default è che non venga operato. Lo si abilita semplicemente mettendo una apposita direttiva `#define` prima di includere `verify.h`

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

Un altro switch utile può essere `Verify_INLINE`, che quando è definito fa sì che i metodi per il calcolo di pre, post ed invariante di classe siano espansi in linea. È vero che questi tre metodi sono lunghi, ma soprattutto se i vincoli sulla chiusura pre-post e sull'inizializzazione sono disabilitati, allora il loro codice effettivo si riduce a due confronti e due chiamate di funzione, addirittura un confronto ed una chiamata di funzione nel caso di `evaluateClassInvariant()`, quindi in alcuni casi può aver senso eliminare il function call overhead per questi metodi, suggerendo al compilatore di metterli inline (sì, perché alla fine è Lui, in compilatore, a stabilire se è veramente il caso di mettere tali metodi inline). Troppe espansioni in linea, però, possono portare al cosiddetto code-bloat, cioè ad una esplosione della quantità di codice effettivamente compilato, frutto dell'espansione in linea di molte chiamate di funzione, ognuna delle quali molto lunga, ma a questo punto interverranno congiuntamente la saggezza del programmatore e del suo fido compilatore, che valuteranno, pregi e difetti di ogni scelta nel loro contesto, l'importante è che abbiano la possibilità di scegliere...

L'ultimo switch che andiamo a vedere è `Verify_ENSURE_POST_WITH_METHODNAME`, che impone di passare il nome del metodo corrente al metodo `Verify<T>::postCondition()`. Questo switch è normalmente inattivo, perché si fa conto sul fatto che la chiusura pre-post sia attiva e quindi che tale informazione sia stata passata a `verify` attraverso una precedente chiamata a `Verify<T>::preCondition()` operata all'inizio dello stesso metodo. In caso di rilassamento del vincolo di chiusura pre-post questa assunzione potrebbe non essere più vera, per cui qualche sadico capo progetto potrebbe dire "ok, non ti costringo ad usare sempre le precondizioni, giusto perché il nostro programma è troppo lento, ma guai a te se non mi dichiari il nome dei metodi nelle postcondizioni, non ti faccio proprio compilare!". Questo è lo scopo di questo switch...

Gli switch che abbiamo visto sono attivabili tutti indipendentemente l'uno dall'altro, quindi potremo avere anche una situazione come la seguente

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

## UN'OCCHIATA AL CODICE

Come avrete notato nel corso dell'articolo mi sono limitato a mostrare l'uso del template `Verify<T>`, ma non ho mai parlato di come le cose vengono implementate, questo per porre l'accento sull'obiettivo più che sull'implementazione di `Verify<T>`, che di per sé ha poco di interessante. Quello che conta è l'idea esposta nell'articolo, più che la classe template od i suoi dettagli implementativi. Ad ogni buon modo, il codice sorgente di `Verify<T>` è esposto nei riquadri **Listato 1** (`verify.h`), **Listato 2** (`verify.i`) e **Listato 3** (`verify.cpp`).

Il **Listato 1** (*verify.h*) mostra l'interfaccia della classe *template* con tutti i suoi metodi, che ormai abbiamo già visto all'opera. Al termine del listato avviene l'inclusione di *verify.i*, che è esposto nel **Listato 2** e che altro non ha che l'implementazione dei metodi di *Verify<T>*, oltre ad una semplice classe smart pointer *VerifySmartPointer<T>*, usata solo per esportare il puntatore *old* in modo **const** e per far sì che eventuali accessi ad *old* quando non è stato inizializzato (con una chiamata ad *enableOld()*) possa dare un secco "access violation", ma un più aggraziato ed esplicativo

**class:** A

**method:** aMethod

old value evaluation without having previously called the method *enableOld()*

In fondo tutto questo apparato è stato messo sù per facilitare il tracciamento dei banchi, sarebbe stato assurdo che avesse potuto potenzialmente introdurre altri di poco esplicativi come un "access violation".

Nell'ultimo file, *verify.cpp*, che è il **Listato 3** c'è l'implementazione della funzione *Verify\_throwMessage()*, il cui unico scopo è quello di stampare messaggi di errore il più esplicativi possibile ed in maniera portabile e consona alla piattaforma di compilazione. A tal pro il codice sfrutta la compilazione condizionale per sparare a video l'errore con una dialog box sotto Windows o su *cerr* in ogni altro sistema operativo od in Windows stesso quando l'applicazione è di tipo console. Questa è una funzione e non un metodo privato del template, perché ho trovato dannoso dover costringere ad includere *iostream.h* oppure *windows.h* insieme a *verify.h* anche in codice che in realtà non ha alcun bisogno di includere tali file.

Una particolarità di tutto il codice sorgente è l'estensivo uso delle stringhe c-style, ossia i **char\***, al posto di una ben più elegante classe stringa. Il motivo di tale scelta è da ricercare nel fatto che in realtà non esiste una classe stringa standard disponibile in ogni implementazione di compilatore C++ (AT&T ed ANSI). Considerato l'obiettivo di *Verify<T>* e l'uso che fa dei **char\*** non mi è sembrato opportuno introdurre problemi di portabilità con l'adozione di una classe stringa.

## COSTRUZIONE PER COPIA ED ASSEGNAMENTO

Già abbiamo avuto modo di parlare dei costruttori dei copia per l'oggetto ospite nel corso dell'esposizione del meccanismo per l'ottenimento degli old-values. Ci sono altri problemi che possono sorgere quando la classe ospite non implementa un proprio costruttore di copia, ma si affida a quello generato dal compilatore. Consideriamo il caso di una classe ospite *A* ed il seguente frammento di codice C++

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

Bene, il costruttore di copia generato dal compilatore per *A* ha costruito *b* facendo una copia binaria dell'oggetto *a*, quindi il famoso puntatore alla classe ospite che gli oggetti *a.verify* e *b.verify* hanno sono lo stesso puntatore e per la precisione punta ad *a*. La conseguenza è che gli old values di *b* saranno in realtà quelli di *a*. Lo stesso accade se non si definisce in modo opportuno un operatore di assegnamento per la classe ospite *A*, in quanto ne genera una il compilatore, che opera la copia binaria e nel frammento di codice che segue

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

le conseguenze sono esattamente le stesse. Per ovviare ad un tale problema è opportuno disporre di un costruttore di copia e di un operatore di assegnamento specializzati, che inizializzino l'oggetto *verify* durante la costruzione e l'assegnamento

```

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

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

```

Siccome non è improbabile che qualcuno si dimentichi di questo particolare, per evitare spiacevoli conseguenze, la classe *Verify*<*T*> ha dichiarato il proprio costruttore di copia ed il proprio operatore di assegnamento come membri privati, impedendo di fatto al compilatore di generare costruttori di copia ed operatori di assegnamento per le classi ospiti: il compilatore a questo punto pretenderà l'implementazione esplicita di tali metodi, se queste feature sono richieste dal vostro codice.

## UNA PICCOLA NOTA FILOSOFICA SUL DBC

Già nel corso dell'articolo ho evidenziato che lo scopo del DBC non è quello inzeppare il codice di asserzioni pronte a scattare ad ogni piè sospinto (la principale critica che taluni muovono a tale tecnica di design sviluppo), bensì quello di dare un mezzo per fornire quante più informazioni possibili agli utenti di una istanza di una classe, in modo da permettere loro di scrivere applicazioni che in generale saranno più affidabili. Tra l'altro un segnale del fatto che spesso lo sviluppatore utente di una classe ha bisogno di tali informazioni sta nella diffusa credenza che sia necessario disporre dei sorgenti di una classe per usarla in modo tranquillo ed affidabile... questo significa che gli sviluppatori molto spesso vogliono vederci più chiaro, la sola interfaccia di una classe ed un frettoloso manuale d'uso non sono spesso sufficienti a prevenire qualche problema. Non è detto che l'aggiunta di pre e postcondizioni a tali informazioni sia sufficiente, tutto dipende dal buon senso di chi le scrive, ma è un fatto che qualora esse risultino violate, ci portano subito nella zona di codice dove risiede il problema, quindi il problema del tracciamento dei banchi è già mezzo risolto, basta eseguire l'applicazione in quanti più contesti sia possibile, attendendo l'uscita di una dialog di errore. L'alternativa, senza un tale framework, è quella di fermarsi ogni volta a valutare la correttezza dei numeri scritti a video prima di passare ad un altro test e questo è molto più lento da farsi che non dire al programma tester "run" e vedere se, quando e perché esce una dialog di errore. Inoltre quando una tale dialog di errore esce fuori essa non dice "access violation all'indirizzo XXXXYYYY", oppure "abnormal program termination", bensì chi ha causato il problema, dove e perché e questo già fa una bella differenza per quel povero cristo che dovrà debuggare il codice.

## CONCLUSIONI

Questo piccolo framework aiuta lo sviluppatore C++ a seguire i dettami del DBC usando il suo linguaggio di programmazione ed in una maniera pulita ed ordinata. Ciò, insieme all'adozione di alcune convenzioni, come quella di piazzare sempre una preconditione come prima linea di codice di ogni metodo pubblico ed una postcondizione come ultima istruzione di un metodo, porta all'ottenimento di codice più chiaro, affidabile ed soprattutto autodocumentantesi in termini di pre e postcondizioni, attraverso le short form delle classi. Le short form sono ottenibili attraverso un semplice programmino, anche perché il framework standardizza la definizione di preconditioni, postcondizioni ed invarianti di classe e tutto ciò facilita l'adozione del DBC in grandi team di sviluppo in C++ dove non è facile imporre troppe convenzioni nella codifica delle classi...

## BIBLIOGRAFIA

[1] Bertrand Meyer. *Introduction to the Eiffel language and methods*,

[www.eiffel.com/doc/manuals/language/intro/index.html](http://www.eiffel.com/doc/manuals/language/intro/index.html), 1985-1998