

# Una implementazione del Design By Contract in Java

Leonardo Cecchi

16 novembre 2006

## Sommario

Viene introdotta la metodologia di ‘Progettazione per Contratto’ [dbc-92] e vengono confrontate le più significative implementazioni in Java. Dal confronto nasce la discussione di nuovi requisiti implementativi e viene descritta una nuova implementazione che permette di superare i problemi degli altri framework.

## 1 Introduzione

Durante la fase di progettazione di un sistema software orientato agli oggetti è necessario definire delle specifiche funzionali per ogni modulo del sistema prima che questo venga codificato.

Molto spesso queste specifiche non vengono rese formali attraverso dei meccanismi come la *Z Specification Notation* [zsn-ref] a causa dell’amplificazione dei costi di progettazione dell’applicativo.

La specifica rimane quindi a documentazione del progetto ma non si trasforma in nessun codice eseguibile.

La Progettazione per Contratto [dbc-92] è una forma di specifica funzionale del software che permette un controllo continuo sul software (riguardante le specifiche funzionali) e che ha dato vita a linguaggi appositi [eiffel].

I fondamenti della progettazione per contratto sono due:

**Tipi di Dati Astratti** Tipi di dati che possono essere manipolati esclusivamente basandosi sulla loro semantica e non dalla loro implementazione

**Metafora di un Contratto Legale** Ogni modulo del software è dotato di un ‘contratto’ che viene specificato prima della fase di codifica. Questo contratto enuncia quale può essere lo stato dell’elaboratore prima della elaborazione del modulo ed in quali stati l’elaboratore si può trovare successivamente all’esecuzione.

Nel caso della progettazione ad oggetti è possibile identificare due tipi di moduli:

**Metodi:** devono essere progettati in modo da enunciare in maniera formale lo stato del programma prima e dopo la loro esecuzione

**Classi:** possono essere progettate in modo da enunciare quali sono le condizioni necessarie alla consistenza dei dati contenuti nelle loro istanze.

Nei contratti vengono individuate tre tipi di clausole primarie:

**Pre-condizioni** ovvero le condizioni che devono essere rispettate prima dell'esecuzione di un metodo. Nella progettazione per contratto classica espressa in [dbc-92] comprendono anche i check di consistenza dei parametri formali del metodo

**Post-condizioni** ovvero le condizioni che devono essere rispettate dopo l'esecuzione di un metodo.

**Invarianti di classe** ovvero la condizione che garantisce che i dati contenuti in una istanza della classe siano consistenti.

Dalla precedente descrizione è possibile notare che le clausole sono facilmente descrivibili con delle condizioni booleane che devono essere periodicamente controllate per assicurare la correttezza del software.

Il precedente approccio è implementato anche in [eiffel].

L'uso della progettazione per contratto permette anche un raffinamento del concetto di ereditarietà come specificato in [dbc-intro].

Una classe *BClass* che eredita da una classe *AClass* può provvedere una nuova definizione per un certo metodo di *AClass*: in questo modo è libera di cambiarne la semantica. Questa conseguenza è particolarmente problematica nel caso del polimorfismo perchè la chiamata:

```
AClass a;  
...  
a.method();
```

potrebbe causare l'invocazione dell'implementazione di *method* nella classe *AClass* oppure nella classe *BClass* in base al *dynamic binding*.

Questo caso, per la progettazione per contratto, è un *subappalto*, ovvero la classe *BClass*, ridefinendo il comportamento di *method*, deve rispettarne anche le precondizioni oppure alleggerirle.

Per quanto riguarda le postcondizione deve essere verificata la procedura simmetrica ovvero l'implementazione nella classe derivata deve rispettare le postcondizioni dichiarate da quella nella superclasse oppure restringerle.

## 2 Progettazione per Contratto in Java

In Java non è presente il supporto alla progettazione per contratto. Nella documentazione è semplicemente consigliato un controllo di validità dei parametri dei metodi che può essere interpretato come una forma di preconditione [java-assert]:

```
Do not use assertions to check the parameters of a public method.  
An assert is inappropriate because the method guarantees that it  
will always enforce the argument checks. It must check its arguments  
whether or not assertions are enabled. Further, the assert construct  
does not throw an exception of the specified type. It can throw only  
an AssertionError.
```

Sarà quindi necessario, come requisito implementativo, distinguere l'evento di una pre-condizione non rispettata dai controlli dei parametri.

Nella versione 1.5 di Java non è presente il supporto per la progettazione per contratto ma è disponibile una semplice funzionalità di gestione delle asserzioni.

Questo permette di implementare le varie clausole dei contratti come metodi separati e di garantirne l'invocazione quando serve.

Consideriamo la seguente classe annotata con i contratti:

```
// Invariante di classe: "message!=null && user!=null && pChannel!=null"
public class ChatMessage {
    private String message, user;
    private Channel channel;

    public ChatMessage(String pMessage, String pUser, Channel pChannel);

    // Precondizione: "channel.isOpened()"
    // Postcondizione: "channel.isFlushed()"
    public void trasmit() throws IOException;
}
```

Nella precondizione del costruttore non è stato introdotto, in concordanza con [java-assert], il controllo dei parametri. E' possibile controllare l'invariante di classe con la seguente trasformazione:

```
public class ChatMessage {
    private String message, user;
    private Channel channel;

    public ChatMessage(String pMessage, String pUser, Channel pChannel) {
        ...
        check_invariant();
    }

    // Precondizione: "channel.isOpened() && channel.isReady()"
    // Postcondizione: "channel.isFlushed()"
    public void trasmit() throws IOException {
        ...
        check_invariant();
    }

    /**
     * Gestione invariante di classe
     */
    public void check_invariant() {
        if(!(message!=null && user!=null && pChannel!=null)) {
            throw new ContractError("Invalid Class Invariant");
        }
    }
}
```

E' possibile trattare in maniera simile la precondizione e la postcondizione.

```

public class ChatMessage {
    private String message, user;
    private Channel channel;

    public ChatMessage(String pMessage, String pUser, Channel pChannel) {
        ...
        check_invariant();
    }

    public void trasmit() throws IOException {
        pre_trasmit();
        ...
        post_trasmit();
        check_invariant();
    }

    /**
     * Precondizione del metodo trasmit
     */
    public void pre_trasmit() {
        if(!(channel.isOpened() && channel.isReady())) {
            throw new ContractError("Precondizione non rispettata");
        }
    }

    /**
     * Postcondizione del metodo trasmit
     */
    public void post_trasmit() {
        if(!(channel.isFlushed())) {
            throw new ContractError("Postcondizione non rispettata");
        }
    }

    /**
     * Gestione invariante di classe
     */
    public void check_invariant() {
        if(!(message!=null && user!=null && pChannel!=null)) {
            throw new ContractError("Invariante di classe non rispettata");
        }
    }
}

```

Consideriamo adesso la classe *EnhancedChatMessage* che fornisce le stesse funzionalità della precedente ma in modo ottimizzato:

```

public class EnhancedChatMessage extends ChatMessage {
    private String message, user;
    private Channel channel;

```

```

public EnhancedChatMessage(String pMessage, String pUser, Channel pChannel);

        // Precondizione: "channel.isOpened()"
public void trasmit() throws IOException;
}

```

Questa potrà quindi essere trasformata come segue:

```

public class EnhancedChatMessage {
    private String message, user;
    private Channel channel;

    public EnhancedChatMessage(String pMessage, String pUser, Channel pChannel) {
        check_invariant();
    }

    public void trasmit() throws IOException {
        pre_trasmit();
        ...
        post_trasmit();
        check_invariant();
    }

    /**
     * Precondizione del metodo trasmit
     */
    public void pre_trasmit() {
        super.pre_trasmit();
        if(!(channel.isOpened())) {
            throw new ContractError("Precondizione non valida");
        }
    }

    /**
     * Postcondizione del metodo trasmit
     */
    public void post_trasmit() {
        super.post_trasmit();
    }

    /**
     * Gestione invariante di classe
     */
    public void check_invariant() {
        super.check_invariant();
    }
}

```

Le trasformazioni effettuate per l'implementazione della progettazione per contratto sono facili però aumentano molto il tempo necessario per la codifica quindi non sono già proponibili per progetti di media dimensione.

Si rivela quindi la necessità di un prodotto in grado di implementare la progettazione per contratto nel linguaggio Java che realizzi in modo automatico queste trasformazioni.

### 3 Requisiti Implementativi

Per l'implementazione del framework qui esposto sono stati selezionati i seguenti requisiti:

**Facilità d'uso per l'analizzatore** Un framework semplice da usare per l'analizzatore incoraggia la diffusione della tecnica di progettazione per contratto e quindi porta a analisi più efficienti.

**Facilità d'uso per lo sviluppatore** Il ciclo di sviluppo deve cambiare il meno possibile.

**Esposizione dei contratti nella documentazione** I contratti espressi durante la fase di progettazione devono essere esposti anche nella documentazione implementativa del progetto. La presenza dei contratti migliora la comprensione del software sviluppato e quindi rende maggiormente probabile un riuso.

**Indipendenza** Il framework non deve dipendere da altre piattaforme perchè questo rende difficile la creazione degli ambienti di sviluppo.

**Contratto e implementazione nello stesso file** Avere l'esposizione del contratto e l'implementazione corrente nello stesso file incoraggia il programmatore al rispetto del contratto stesso.

**Facilità di disabilitazione** I contratti possono essere pesanti da verificare. E' quindi opportuno disabilitarli in fase di deploy.

Alla luce di questi requisiti vengono esaminati i framework con licenze open-source compatibili presenti.

#### 3.1 JContractor

JContractor [jcontractor] è un framework rilasciato sotto licenza Apache che implementa la progettazione per contratto in Java eseguendo una modifica del bytecode precedentemente generato dal compilatore.

I contratti possono venire espressi come metodi a parte nella classe di appartenenza oppure come classi a parte identificate dal suffisso *\_CONTRACT*.

Ad ogni ciclo di sviluppo il codice generato dal compilatore deve essere nuovamente modificato dal Framework.

Di seguito viene citato un esempio di codice da [jcontractor-doc].

```
class Stack implements Cloneable {
    private Stack OLD;
    private Vector implementation;
```

```

public Stack () { ... }
public Stack (Object [] initialContents) { ... }
public void push (Object o) { ... }
public Object pop () { ... }
public Object peek () { ... }
public void clear () { ... }
public int size () { ... }
public Object clone () { ... }
private int searchStack (Object o) { ... }
}

class Stack_CONTRACT extends Stack {
    private Stack OLD;
    private Vector implementation;

    protected boolean Stack_Postcondition (Object [] initialContents, Void RESULT) {
        return size() == initialContents.length;
    }

    protected boolean Stack_Precondition (Object [] initialContents) {
        return (initialContents != null) && (initialContents.length > 0);
    }

    protected boolean push_Precondition (Object o) {
        return o != null;
    }

    protected boolean push_Postcondition (Object o, Void RESULT) {
        return implementation.contains(o) && (size() == OLD.size() + 1);
    }

    private int searchStack (Object o) {
        return 0;
    }

    private boolean searchStack_Precondition (Object o) {
        return o != null;
    }

    protected boolean _Invariant () {
        return size() >= 0;
    }
}

```

Rispetto ai requisiti implementativi abbiamo che:

**Facilità d'uso per l'analizzatore** L'analizzatore può scrivere direttamente i contratti nella classe di appartenenza.

**Facilità d'uso per lo sviluppatore** Il ciclo di sviluppo deve essere variato per includere la fase di integrazione dei contratti

**Esposizione dei contratti nella documentazione** I contratti vengono esposti nella documentazione ma in maniera indipendente dai metodi di appartenenza.

**Indipendenza da questo framework** Il framework non è dipendente dalle altre piattaforme.

**Contratto e implementazione nello stesso file** E' possibile.

**Facilità di disabilitazione** E' possibile evitando l'inclusione dei contratti. E' anche possibile rimuovere i contratti dalle classi con una funzione di 'strip'.

## 3.2 iContract2

iContract2 [icontract2] è una nuova versione del progetto iContract di Rileable Systems, ormai abbandonato.

I contratti vengono espressi come dei tag javadoc-like nei commenti dei metodi:

```
/**
 * @inv !isEmpty() implies top() != null // no null objects are allowed
 */
public interface Stack
{
    /**
     * @pre o != null
     * @post !isEmpty()
     * @post top() == o
     */
    void push(Object o);

    /**
     * @pre !isEmpty()
     * @post @return == top()@pre
     */
    Object pop();

    /**
     * @pre !isEmpty()
     */
    Object top();

    boolean isEmpty();
}
```

I contratti vengono inseriti nelle classi di appartenenza generando dal sorgente delle classi un altro codice Java che viene compilato e successivamente riletto dalla virtual machine.

Rispetto ai requisiti implementativi sopra espressi abbiamo che:

**Facilità d'uso per l'analizzatore** L'analizzatore può scrivere direttamente i contratti nella classe di appartenenza.

**Facilità d'uso per lo sviluppatore** Il ciclo di sviluppo deve essere variato per includere la fase di integrazione dei contratti

**Esposizione dei contratti nella documentazione** I contratti vengono esposti nella documentazione nei metodi di appartenenza.

**Indipendenza da questo framework** Il framework non è dipendente da altre piattaforme.

**Contratto e implementazione nello stesso file** E' possibile.

**Facilità di disabilitazione** E' possibile evitando l'inclusione dei contratti.

### 3.3 Contract4J

Contract4J [contract4j] è un framework per l'implementazione della progettazione per contratto che permette di inserire le clausole come annotazioni di Java 1.5.

Il problema dell'inclusione dei contratti nel bytecode finale viene risolto con l'uso di tecnologie di programmazione orientate agli aspetti (AOP).

Il framework AOP usato è AspectJ [aspectj].

Un esempio di codice:

```
import ...PhoneNumber;
import ...Address;
import com.aspectprogramming.contract4j.*;

@Contract
public class SearchEngine {
    ...
    @Pre
    @Post("$return != null && $return.isValid()")
    public PhoneNumber search (String first, String last,
                               Address streetAddress) {
        PhoneNumber result =
            doSearch (first, last, streetAddress);
        return result;
    }
    ...
}
```

Analisi dei requisiti implementativi:

**Facilità d'uso per l'analizzatore** L'analizzatore può scrivere direttamente i contratti nella classe di appartenenza.

**Facilità d'uso per lo sviluppatore** Il ciclo di sviluppo deve essere variato per includere la fase di integrazione dei contratti

**Esposizione dei contratti nella documentazione** I contratti non vengono esposti nella documentazione nei metodi di appartenenza.

**Indipendenza da questo framework** Il framework è dipendente dal framework per l'Aspect Oriented Programming.

**Contratto e implementazione nello stesso file** E' possibile.

**Facilità di disabilitazione** E' possibile escludendo la compilazione degli aspetti oppure manipolando i join-points.

### 3.4 C4J

Usando C4J [c4j] i contratti vengono inseriti in classi separate.

Questa framework usa le nuove funzionalità della VM Java 1.5 includendo il bytecode necessario alla gestione dei contratti direttamente (al caricamento di queste ultime) nella virtual machine.

Un esempio di codice:

```
@ContractReference(contractClassName = "DummyContract")
public class Dummy
{
    public double divide(double x, double y) {
        return x / y;
    }
}

public class DummyContract
{
    public void classInvariant() {
        // Nothing to do here
    }

    public void pre_divide(double x, double y) {
        assert y != 0;
    }
}
```

Analisi:

**Facilità d'uso per l'analizzatore** L'analizzatore non può scrivere direttamente i contratti nella classe di appartenenza.

**Facilità d'uso per lo sviluppatore** Il ciclo di sviluppo non deve essere variato per includere la fase di integrazione dei contratti

**Esposizione dei contratti nella documentazione** I contratti non vengono esposti nella documentazione nei metodi di appartenenza ma in classi separate

**Indipendenza da questo framework** Il framework non è dipendente da alcuna altra piattaforma.

**Contratto e implementazione nello stesso file** Non e' possibile.

**Facilità di disabilitazione** E' possibile escludendo l'agent di manipolazione del bytecode.

### 3.5 Valutazione comparativa

Tabella riassuntiva delle caratteristiche dei framework analizzati:

Framework	Facilità Analizzatore	Facilità Sviluppatore	Contratti e documentazione
JContractor	OK	Variazione	Indipendentemente
iContract2	OK	Variazione	OK
Contract4J	OK	Variazione	Non presenti
C4J	OK	OK	Indipendentemente

Framework	Indipendenza	Coresidenza sorgente/contratti	Disabilitazione
JContractor	OK	OK	OK anche strip
iContract2	OK	OK	OK
Contract4J	Necessita AspectJ	Variazione	Non presenti
C4J	OK	OK	Possibile

L'assenza di un framework che abbia tutti i requisiti identificati ha portato allo sviluppo di un'altra implementazione che sia leggera ma flessibile e semplice da usare.

## 4 Descrizione dell'Implementazione

L'implementazione proposta usa le annotazioni di Java 1.5 per inserire i contratti all'interno del codice sorgente e modifica il bytecode direttamente all'interno della Virtual Machine Java usando la nuova opzione 'javaagent' [man-java].

Questa funzionalità permette di registrare all'avvio della virtual machine un agente in grado di definire una funzione di trasformazione del tipo:

$$b \in \{byte[]\}, b' \in \{byte[]\} \cup \{null\}, \quad transform : b \rightarrow b'$$

Questa funzione è invocata ad ogni lettura di un file di classe (da un file jar, da un file class, da una sorgente web oppure da ogni implementazione definita di *ClassLoader*).

Il risultato della funzione viene usato come segue:

$b' = null$  Nella virtual machine viene usato il bytecode rappresentato da  $b$ .

$b' \neq null$  Nella virtual machine viene usato il bytecode rappresentato da  $b'$ .

E' possibile trovare una documentazione della struttura qui delineata in [api-java-lang-instrument].

A questa struttura è possibile aggiungere la funzionalità delle asserzioni da conservarsi nel bytecode (*RetentionPolicy.RUNTIME*):

```

/**
 * Precondizioni
 * @author Leonardo Cecchi
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface LeoPre
{
    /**
     * Precondizione da applicare ad un certo metodo
     */
    public String value() default "true";
}

```

Questa funzionalità è invece documentata in [java-annotations].

Nel framework proposto il processo di deploy dei contratti si divide in queste fasi:

**Scrittura dei contratti** da effettuarsi in fase di progettazione e attraverso le annotazioni.

**Compilazione** da effettuarsi con qualsiasi compilatore che supporti le specifiche del bytecode di Java 1.5

**Esecuzione** con la VM Java ed il relativo parametro di installazione dell'agent

Durante l'esecuzione l'agent viene invocato per ogni classe letta. E' da notare che la semantica dell'agente di trascodifica dei bytecode non prevede, diversamente da quanto accade per i ClassLoader, che le superclassi e le interfacce implementate siano già caricate al momento del caricamento della classe.

La trasformazione del bytecode Java avviene quindi nelle seguenti fasi:

1. Creazione di una cache di bytecode già caricati. La cache è modellabile come una funzione del tipo:

$$cache : \{ClassLoaders\} \times \{Strings\} \rightarrow \{Bytecodes\} \cup \{Null\}$$

Se la classe  $Cl \in \{ClassLoaders\}$  e  $Name \in \{Strings\}$  è già stata caricata e elaborata allora la funzione  $cache(Cl, Name) = Bytecode$  ritorna il bytecode già elaborato.

Se invece non è ancora stata caricata allora la funzione  $cache(Cl, Name) = Null$ .

2. Lettura e decodifica del bytecode con seguente identificazione della classe caricata. Il bytecode letto potrebbe corrispondere ad una classe oppure ad una interfaccia. Se il bytecode corrisponde ad una interfaccia la funzione di trasformazione ha valore nullo.
3. Supponendo che la classe letta si chiami  $c$ , il classloader usato per il caricamento sia  $Cl$ , che la sua superclasse si chiami  $\bar{c}$  e che ci siano  $n$  interfacce da essa implementate chiamate  $i_1, i_2, \dots, i_n, 0 \leq i < n$ , si controllano i valori

di  $cache(Cl, \bar{c}), cache(Cl, i_1), cache(Cl, i_2), \dots, cache(Cl, i_n)$ . Se qualcuno di questi valori è *Null* si procede al caricamento ed alla elaborazione della classe non caricata prima di continuare.

4. A questo punto è possibile assumere che tutte le classi necessarie all'elaborazione di  $c$  siano state caricate e elaborate. La decodifica del bytecode di  $c$  viene passata attraverso un numero variabile di filtri in base alla configurazione del framework. Ogni filtro è modellabile con una funzione del tipo:

$$byteCodeFilter : \{Bytecode\} \rightarrow \{Bytecode\} \times \{Ok, Error\}$$

La creazione di un errore da parte di un filtro causa la terminazione dell'operazione di trasformazione. L'agente provvede a visualizzare l'errore ed in questo caso la funzione di trasformazione ha valore nullo.

E' da notare che questa ultima clausola implementativa permette al software di essere eseguito anche in caso di errore di compilazione dei contratti.

Questa semantica è quindi compatibile con quella di compilazione di una classe Java.

L'applicazione dei filtri è quindi esprimibile così:

```
Bytecode filteredByteCode=originalByteCode;
for(byteCodeFilter in ByteCodeFilters) {
    (filteredByteCode, status)=byteCodeFilter(filteredByteCode);
    if(status==Error) {
        return(null);
    }
}
return(filteredByteCode);
```

5. Il bytecode filtrato viene inserito nella cache per le successive elaborazioni.
6. Il bytecode filtrato viene passato alla macchina virtuale Java.

## 4.1 Javassist makes Java bytecode manipulation simple

Per manipolare il bytecode Java viene usata la libreria Javassist [chiba-gpce03] rilasciata sotto doppia licenza MPL e LGPL e liberamente scaricabile da [javassist-main].

La libreria Javassist, modellata sullo stile delle Java Reflection API, permette di manipolare il bytecode in modo semplice ed efficace ed include un compilatore in grado di generare da un linguaggio Java-Like del bytecode Java che è inseribile prima e dopo il codice corrispondente ai metodi ed ai costruttori.

Questo la rende perfetta per la realizzazione di un toolkit per la progettazione per contratto.

## 4.2 Struttura dei package

Il toolkit è strutturato in vari package Java:

**leodbc.javaagent** Agente della VM Java

**leodb.c.javaagent.generation** Codice della generazione dei metodi della implementazione dei contratti

**leodb.c.javaagent.instrumentor** Funzione di trasformazione registrata all'interno della VM Java

**leodb.c.javaagent.res** Risorse condivise (costanti)

**leodb.c.javaagent.transformations** Filtri di trasformazione

**leodb.c.support** Classi di supporto necessarie in fase di compilazione anche se non viene utilizzato l'agent

Per le singole API è possibile consultare la documentazione in formato JavaDoc presente all'interno del pacchetto

### 4.3 Annotazioni

Per la scrittura dei contratti sono state create delle annotazioni che sono caratterizzate solamente da un metodo *value*.

Questa caratteristica permette di abbreviare molto la sintassi usata per l'inserimento dei contratti nei codici sorgenti.

**@LeoPre** Annotazione che esprime le precondizioni (da applicare a metodi oppure a costruttori)

**@LeoPost** Annotazione che esprime le postcondizioni (da applicare a metodi oppure a costruttori)

**@LeoInvariant** Annotazione che esprime una invariante (da applicare a classi oppure a interfacce)

**@LeoArgCheck** Annotazione da applicare ad un parametro formale di un metodo oppure di un costruttore. di una certa espressione prima dell'invocazione del metodo. Questo permette una migliore efficacia delle postcondizioni (vedere esempi)

### 4.4 Il filtro per le precondizioni

Il filtro per le precondizioni genera un metodo per ogni contratto di precondizione da applicare. Una chiamata al metodo creato, che ha la stessa segnatura del metodo del contratto eccetto il nome, viene inclusa nel metodo del contratto.

Questo è lo schema del codice generato:

```
public void pre_<mname>(<margs>) {
    leodb.c.support.VariableStore _store=leodb.c.support.VariableStoreFactory.get();
    _store.addMethodMarker();
    <chiamata precondizioni superclasse se applicabile>
    if(!(<asserzione>)) {
        throw new leodb.c.support.ContractError("Wrong precondition");
    }
}
```

La riga relativa alla definizione della variabile `_store` serve per poter memorizzare i valori al momento della definizione della preconditione. Questi valori possono poi essere riutati nel check della postcondizione. Negli esempi viene trattata una situazione dove la variabile `_store` viene usata.

E' da notare, visto che l'elaborazione del contratto avviene in fase di lettura delle classi, che non sono piu' presenti le informazioni relative ai nomi dei parametri formali.

Questo comporta l'uso di una sintassi posizionale nel riferirsi ai parametri del metodo durante una pre-condizione. La sintassi usata è del tipo: `$0`, `$i`, `...`, `$n` per riferirsi all'i-esimo parametro.

Durante la valutazione della pre-condizione l'uso dei parametri formali è poco frequente quindi l'uso della sintassi posizionale degli argomenti non mette a disagio il programmatore.

Per riferirsi ai parametri formali è appositamente presente lo strumento delle annotazioni `@LeoArgCheck`.

## 4.5 Il filtro per le postcondizioni

Il filtro per le postcondizioni genera un metodo per ogni contratto di postcondizione da applicare. Viene successivamente inserita una chiamata al nuovo metodo in fondo al metodo del contratto.

Questo è lo schema del codice generato:

```
public void post_<mname>_<msig>(<marg>) {
    leodbc.support.VariableStore _store=leodbc.support.VariableStoreFactory.get();
    try {
        <chiamata postcondizioni superclasse se applicabile>
        if(!<asserzione>)) {
            throw new leodbc.support.ContractError("Wrong postcondition");
        }
    } finally {
        _store.removeMethodAssociations();
    }
}
```

Il metodo creato ha:

**Un solo argomento** del tipo uguale al tipo di ritorno del metodo del contratto se questo è diverso da `void`

**Nessuno argomento** se il tipo di ritorno del metodo del contratto è `void`

E' possibile riferirsi al valore ritornato dal metodo durante la scrittura delle asserzioni usando il marcatore `$1`.

Anche qua la riga relativa alla definizione della variabile `_store` serve per poter memorizzare i valori al momento della definizione della preconditione.

## 4.6 Il filtro per le invarianti di classe

Per ogni annotazione `@LeoInvariant` viene creato un metodo che non ha alcun parametro formale e ritorna `void`. Una chiamata a questo metodo viene inserita in tutti i metodi della classe.

Questo è lo schema del metodo generato

```
public void check_invariant() {
    <chiamata alla invariante della superclasse se applicabile>
    if(!(<asserzione>)) {
        throw new leodbc.support.ContractError("Wrong class invariant");
    }
}
```

## 4.7 Il filtro per il controllo dei parametri dei metodi

Questo filtro permette di inserire un controllo sul valore attuale dei parametri formali. Il controllo viene aggiunto in testa al metodo dichiarante ed ha questa forma:

```
if(!(<asserzione>)) {
    throw new IllegalArgumentException("Illegal argument");
}
```

Al posto della eccezione `ContractError` qui viene sollevata una eccezione del tipo `IllegalArgumentException` per essere coerenti con il contratto generico dei metodi in Java (vedi [java-assert]).

## 5 Esempi d'uso

Per dimostrare la validità e la semplicità d'uso del framework progettato propongo un esempio i cui contratti non sono ancora espressi in modo formale.

Attraverso il processo di affinamento verranno trasformati in contratti formalizzati con le annotazioni e questo aiuterà lo sviluppatore a creare una implementazione corretta.

Ecco i servizi necessari alla implementazione:

```
/**
 * Semplice implementazione di un vettore generico al solo
 * scopo di mostrare l'uso del framework LeoDBC
 */
public interface SimpleVector<T> {
    /**
     * Deve inserire un elemento nel vettore
     */
    public void add(T toInsert);

    /**
     * Deve ritornare un valore booleano vero se l'elemento
     * e' presente nel vettore
     */
    public boolean search(T toSearch);

    /**
     * Deve ritornare il numero di elementi presenti nel vettore
     */
}
```

```

    */
    public int length();
}

```

Ecco una prima implementazione (troppo superficiale) dei servizi coinvolti in SimpleVector<T>:

```

import java.util.Collection;
import java.util.HashSet;

public class VectorOne<T> implements SimpleVector<T> {
    private Collection<T> dataStore=new HashSet<T>();

    public void add(T toInsert) {
        dataStore.add(toInsert);
    }

    public boolean search(T toSearch) {
        return dataStore.contains(toSearch);
    }

    public int length() {
        return -1; // Mi sono scordato di implementare
    }
}

```

E questa è un prototipo della classe che usa l'implementazione del vettore e che verrà usata per testare il software:

Per rendere efficace il test propongo anche una minima classe che usa la classe VectorOne.

```

public class Main {
    public static void main(String args[])
    {
        VectorOne<Integer> k=new VectorOne<Integer>();
        k.add(3);
        k.add(2);
        k.add(1);
        // <altre cose qua>
        if(k.search(2)) {
            System.out.println("Ho trovato l'elemento 2.");
        }
    }
}

```

Ecco l'output della classe Main.

```

E>java -cp .;..\..\out\leodbc.jar Main
Ho trovato l'elemento 2.

```

Dal test sembra funzionare tutto correttamente. Proviamo a formalizzare i contratti a livello di interfaccia:

```

/**
 * Semplice implementazione di un vettore generico al solo
 * scopo di mostrare l'uso del framework LeoDBC
 */
public interface SimpleVector<T> {
    /**
     * Deve inserire un elemento nel vettore
     */
    @LeoPre("_store.putValue(\"oldlen\", length())")
    @LeoPost("_store.getInt(\"oldlen\")==(length()-1)")
    public void add(T toInsert);

    /**
     * Deve ritornare un valore booleano vero se l'elemento
     * e' presente nel vettore
     */
    public boolean search(T toSearch);

    /**
     * Deve ritornare il numero di elementi presenti nel vettore
     */
    @LeoPost("$1>=0")
    public int length();
}

```

Questi sono i contratti che vengono espressi dalle condizioni:

`add(T)` La lunghezza del vettore deve risultare incrementata dopo la chiamata del metodo

`length()` Deve ritornare un numero uguale o più grande di 0.

Adesso eseguiamo il codice sopra:

```

E>java -cp ;..\..\out\leodbc.jar -javaagent:..\..\out\leodbc.jar Main
LeoDBC v0.1alpha start...

```

```

Exception in thread "main" leodbc.support.ContractError: Wrong postcondition
    at leodbc.tutortest.VectorOne.post_length_I_leodbc_tutortest_SimpleVector(VectorOne.java:18)
    at leodbc.tutortest.VectorOne.post_length_I(VectorOne.java:18)
    at leodbc.tutortest.VectorOne.length(VectorOne.java:18)
    at leodbc.tutortest.VectorOne.pre_add_leodbc_tutortest_SimpleVector(VectorOne.java:18)
    at leodbc.tutortest.VectorOne.pre_add(VectorOne.java:18)
    at leodbc.tutortest.VectorOne.add(VectorOne.java:18)
    at leodbc.tutortest.Main.main(Main.java:7)

```

Questo già ci dice che la postcondizione sulla lunghezza non funziona. Ci accorgiamo dell'errore, lo correggiamo e otteniamo la seguente implementazione per il metodo `length`:

```

public int length() {
    return dataStore.size();
}

```

Riproviamo l'esecuzione:

```
E>java -cp .;..\..\out\leodbc.jar -javaagent:..\..\out\leodbc.jar Main
LeoDBC v0.1alpha start...
Exception in thread "main" leodbc.support.ContractError: Wrong postcondition
    at leodbc.tutortest.VectorOne.post_add_Ljava_lang_ObjectV_leodbc_tutortest_SimpleVect
    at leodbc.tutortest.VectorOne.post_add_Ljava_lang_ObjectV(VectorOne.java)
    at leodbc.tutortest.VectorOne.add(VectorOne.java:11)
    at leodbc.tutortest.Main.main(Main.java:10)
```

Adesso la postcondizione sul metodo add fallisce perchè abbiamo usato una struttura di tipo insieme per implementare il vettore. Il secondo di due elementi uguali non viene quindi inserito. Cambiamo l'implementazione in questo modo:

```
private Collection<T> dataStore=new LinkedList<T>();
```

Ecco il risultato:

```
E>java -cp .;..\..\out\leodbc.jar -javaagent:..\..\out\leodbc.jar Main
Ho trovato l'elemento 2.
```

Adesso il nostro programma rispetta le pre/post condizioni che abbiamo espresso nella interfaccia. Possiamo migliorare il grado di sicurezza del nostro programma aggiungendo una invariante di classe che ci assicura che l'oggetto che usiamo per memorizzare i dati non diventi mai nullo.

Otteniamo così una implementazione migliore:

```
@LeoInvariant("dataStore!=null")
public class VectorOne<T> implements SimpleVector<T> {
    private Collection<T> dataStore=new LinkedList<T>();

    public void add(T toInsert) {
        dataStore.add(toInsert);
    }

    public boolean search(T toSearch) {
        return dataStore.contains(toSearch);
    }

    public int length() {
        return dataStore.size();
    }
}
```

E' da notare come i contratti siano stati espressi direttamente nella interfaccia. Questo garantisce che tutte le classi che la implementino supportino i contratti.

E' possibile anche impedire l'inserimento e la ricerca di valori nulli come segue:

```

public void add(@LeoArgCheck("$1!=null") T toInsert) {
    datastore.add(toInsert);
}

public boolean search(@LeoArgCheck("$1!=null") T toSearch) {
    return datastore.contains(toSearch);
}

```

## 6 Sviluppi futuri

Il framework è ancora in fase di testing. Una volta superata questa fase si pensa di aggiungere le seguenti funzionalità:

**Logging dei contratti validati** Sarà necessario tenere traccia in un file di log dei contratti che vengono validati. Il file dovrà avere una struttura XML e dovranno esistere dei tool grafici per l'analisi dei dati.

Sapere quali contratti sono stati validati (e per quante volte) fornisce una stima di qualità del testing dell'applicazione.

**Estensione sintassi per i valori -old-** La sintassi correntemente usata per tenere traccia dei valori al tempo della pre-condizione è abbastanza complessa.

Si pensa di costruire un parser che riconosca una sintassi del tipo (`old <expr>`) e di usare questa informazione per costruire automaticamente le istruzioni per la memorizzazione e per il recupero dei dati corrispondenti.

**Abilitazione selettiva dei filtri** Sarebbe interessante, e poco difficile da implementare, abilitare e disabilitare dinamicamente i filtri di realizzazione dei contratti.

Questo permetterebbe di tenere abilitati anche in fase di produzione i check sui parametri formali.

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Progettazione per Contratto in Java</b>	<b>2</b>
<b>3</b>	<b>Requisiti Implementativi</b>	<b>6</b>
3.1	JContractor . . . . .	6
3.2	iContract2 . . . . .	8
3.3	Contract4J . . . . .	9
3.4	C4J . . . . .	10
3.5	Valutazione comparativa . . . . .	11
<b>4</b>	<b>Descrizione dell'Implementazione</b>	<b>11</b>
4.1	Javassist makes Java bytecode manipulation simple . . . . .	13
4.2	Struttura dei package . . . . .	13
4.3	Annotazioni . . . . .	14
4.4	Il filtro per le precondizioni . . . . .	14
4.5	Il filtro per le postcondizioni . . . . .	15
4.6	Il filtro per le invarianti di classe . . . . .	15
4.7	Il filtro per il controllo dei parametri dei metodi . . . . .	16
<b>5</b>	<b>Esempi d'uso</b>	<b>16</b>
<b>6</b>	<b>Sviluppi futuri</b>	<b>20</b>

## Riferimenti bibliografici

- [dbc-intro] *Building bug-free O-O software: An introduction to Design by Contract(TM)* - <http://archive.eiffel.com/doc/manuals/technology/contract/>
- [dbc-92] *Bertrand Meyer: Applying Design by Contract, in Computer g(IEEE), vol. 25, no. 10, October 1992, pages 40-51.*
- [zsn-ref] <http://www.zuser.org/z/>
- [eiffel] *Eiffel Software* - <http://www.eiffel.com>
- [java-assert] *Java Documentation - Programming With Assertions* - <http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html>
- [jcontractor] *JContractor: Design by Contract for Java* - <http://jcontractor.sourceforge.net/>
- [jcontractor-doc] *JContractor: Crash Course* - <http://jcontractor.sourceforge.net/doc/crashcourse.html>
- [icontract2] *iContract2 - Design by Contract for Java* - <http://www.icontract2.org/>
- [contract4j] *Contract4J* - <http://www.contract4j.org>
- [aspectj] *AspectJ* - <http://www.eclipse.org/aspectj>
- [c4j] *C4J - Design by Contract for Java made easy* - <http://c4j.sourceforge.net/>
- [man-java] *Java - The application launcher* - <http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/java.html>
- [api-java-lang-instrument] *java.lang.instrument Package API*
- [java-annotations] *Java 1.5 New Features - Annotations* - <http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>
- [chiba-gpce03] *An Easy-to-Use Toolkit for Efficient Java Bytecode Translators* - <http://www.csg.is.titech.ac.jp/paper/chiba-gpce03.pdf>
- [javassist-main] *Javassist makes Java bytecode manipulation simple* - <http://www.csg.is.titech.ac.jp/chiba/javassist/>