

# Design By Contract e JML

Angelo Gargantini  
Informatica III 2006/2007

# Design by contract

**Segue questa idea, soprattutto valida per l'OO:**

- l'interfaccia di un modulo definisce un contratto

**Cosa è un contratto?**

**Un accordo tra cliente e fornitore che:**

- lega le due (o più) parti: fornitore e cliente
- è esplicito (scritto)
- specifica gli obblighi e i benefici delle due parti
- normalmente mappa gli obblighi di una parte come benefici dell'altra parte
- non contiene clausole nascoste: gli obblighi sono quelli dichiarati

# Bibliografia

## **Meyer, Bertrand**


- Object-Oriented Software Construction, Prentice Hall, 1988
- <http://archive.eiffel.com/doc/manuals/technology/contract/page.html>

## **Inventò il linguaggio di programmazione EIFFEL**

- la torre Eiffel come esempio:
  - costruita rispettando tempi e budget
  - a partire da "componenti"
  - costruita come cosa temporanea, invece c'è tutt'oggi

# Esempio: contratto di costruzione

	Obblighi	Vantaggi
Cliente	Mettere a disposizione 1 ettaro di terreno edificabile e un tot di soldi	Ricevere un edificio di 3 piani entro tot mesi
Fornitore	Costruire un palazzo di 3 piani entro tot mesi	Non deve fare nulla se non riceve i soldi o se il terreno non è adatto



# Contratti nel software

**Oggetto del contratto: il software, in OO, un insieme di classi**

**Cliente:** chi usa il programma

**Fornitore:** chi scrive il programma

**Esempio: una libreria**

**Il contratto di ogni classe dovrebbe essere:**

- esplicito
- parte dello stesso elemento software

**Il linguaggio Eiffel è stato costruito proprio seguendo i principi del DbC:**

- il linguaggio di programmazione contiene proprio il linguaggio per scrivere contratti

# PRE e POST condizioni

**In un contratto si può:**

- **definire cosa ogni metodo richiede**
  - obbligo per il cliente
  - **precondizioni**
- **cosa fornisce il metodo**
  - obbligo per il fornitore
  - **postcondizioni**

Precondizioni e postcondizioni potrebbero essere espresse usando la logica, meglio per i programmatori usare la sintassi del linguaggio di programmazione o del tool

# Esempio di un Array

## Operazione `insert(element)` in un Array

- `no_elements`: numero di elementi memorizzati
- `size`: dimensione del vettore

## Precondition

- l'array non è pieno:

`no_elements < size`

## Postcondition

- l'elemento `element` è nell'array
- l'array contiene un elemento in più:

`no_elements' = no_elements + 1`

- `no_elements'` indica il valore dopo l'operazione

# Perchè le precondizioni?

## Un metodo deve gestire ogni possibile input?

- In generale **NO**
- Precondizioni **deboli** (TRUE significa nessuna precondizione); tutte le complicazioni sono gestite dalla routine
- Precondizioni **forti** (FALSE significa che non può essere chiamato)

**La scelta di precondizioni è una scelta di progetto; non c'è una regola assoluta: però è meglio scrivere metodi semplici che soddisfino un contratto ben definito che un metodo che "cerca" di gestire tutte le situazioni possibili.**



# Come garantire le precondizioni

## Il cliente deve garantire la precondizioni prima di invocare il metodo. Come fare?

- Se **pre** è la precondizione per un metodo *m* di un oggetto *x*:
  - *pre* potrebbe consistere nel verificare un campo di *x*, o chiamare un metodo, es.  $x.a > 0$  o  $x.c() == 1, \dots$
- il cliente dovrà controllare l'invocazione del metodo *m* di *x* in questo modo:

```
if ( pre ) x.m()  
else { /* special treatment */ }
```
- oppure essere sicuro che **pre** vale prima della chiamata di *m*, in base al ragionamento sul programma

# Come garantire le postcondizioni

## Il fornitore deve garantire le postcondizioni nell'implementazione del metodo

- il cliente potrà assumere che le post condizioni siano vere dopo l'invocazione del metodo

**Cliente:** controlla le precondizioni prima di invocare un metodo ma assume (**non controlla**) che le postcondizioni valgano dopo

**Fornitore:** assume (**non controlla**) che le precondizioni valgano e garantisce le postcondizioni

**Si evita che il controllo venga fatto sia dal cliente che dal fornitore**

# Pre e Post condizioni: test

Considera di essere un laureato in Informatica e sul mercatino cerchi lavoro per fare il programma A (con  $\{P\} A \{Q\}$ , P precondizioni e Q postcondizioni). Sei un po' sfaticato.

- **Preferisci P forte o debole?**
- **Q forte o debole?**

P forte e Q debole rendono A più facile da scrivere

- **Due offerte speciali:**
  - $\{\text{False}\} A \{\dots\}$
  - $\{\dots\} A \{\text{True}\}$

**Quale scegli?**

nel primo caso non devi scrivere A

nel secondo caso puoi implementare A come vuoi

# Proprietà interne di una classe

- Possiamo specificare nel contratto anche una proprietà che vale sempre per tutte le istanze

## INVARIANTE

- L'invariante è vero dopo la creazione dell'oggetto e dopo ogni operazione

### **Esempio per l'Array:**

il numero di elementi è sempre compreso tra 0 e la dimensione massima:

```
0 <= no_elements <= size
```

### **L'invariante definisce un obbligo ulteriore**

- l'implementazione di ogni metodo non deve violare l'invariante
- e una preconditione ulteriore: chi implementa il metodo sa che l'invariante vale

# Contratti come documenti

I contratti possono essere inseriti anche prima di fornire una vera implementazione del metodo

## **Documentazione delle classi**

- dalle definizioni dei contratti si può estrarre in modo automatico le precondizioni e le post condizioni e gli invarianti, che documentano cosa fa la classe

# Ruolo delle eccezioni

**Le eccezioni si sollevano solo quando il contratto è violato**

- precondition (colpa del client)
- postcondition o invariant

**Le eccezioni sono casi eccezionali e NON casi particolari**

Per casi speciali usa le strutture di controllo (es. "if the sum is negative, report an error...")

**Una violazione dei un contratto è la manifestazione di un DIFETTO ("BUG")**

# Vantaggi TECNICI nell'uso del DbC

## **Dal libro di Meyer:**

- *Development process becomes more focused.*
- *Writing to spec.*
- *Sound basis for writing reusable software.*
- *Exception handling guided by precise definition of "normal" and "abnormal" cases.*
- *Interface documentation always up to date, can be trusted.*
- *Documentation generated automatically.*
- *Faults occur close to their cause. Found faster and more easily.*
- *Guide for black-box test case generation.*

# Alcuni sistemi

**Eiffel** – linguaggio OO simile a Simula

**GNU Nana:** per C e C++

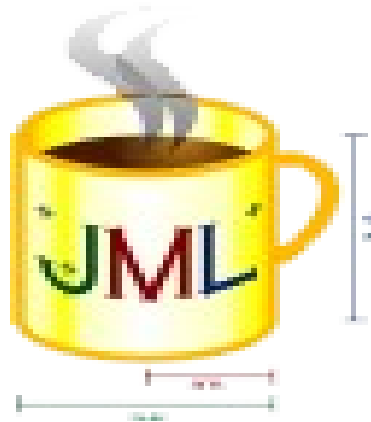
**iContract (Java)**

**(Java assertion in jdk 1.4)**

<http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>

**Jass: Java Assertions:** <http://semantik.informatik.uni-oldenburg.de/~jass/>

**JML: The Java Modeling Language (JML)**





# DbC per Software OO

**Vedremo come specificare le pre, post condizioni e invarianti in programmi **Java** usando **JML****

**JML è un insieme di tool e di tecniche per il design by contract in Java ed altro**

<http://www.cs.iastate.edu/~leavens/JML/>

[http:// www.jmlspecs.org](http://www.jmlspecs.org)

**Sintassi simile per tutti i tool dbc e per tutti i linguaggi OO**

# Sintassi di JML in breve

Per rendere JML facile da usare le annotazioni JML vengono aggiunte come **commenti** nel file .java tra

**`/*@ . . . @*/` , o dopo `//@`**

Le condizioni sono scritte come espressioni **boolean** di java con alcuni operatori in più:

**`\result`, `\forall`, `\old`, `==>`, ...**

e alcune parole chiave

**`requires`, `ensures`, `invariant`, ...**

**Il .java contiene il codice e il suo modello formale secondo DbC**

**Il .java è un valido file java**

# Esempio: conto in banca

**Sia Account una classe che modella un conto in banca [onesta]- senza dbc:**

```
public class Account{
    int balance;
    static final int minBalance = 1000
    /** crea un nuovo conto con una
        certa somma iniziale*/
    public Account (int initialAmount) {...}
    /** preleva dei soldi */
    public int withdraw(int amount) {...}
    /** deposita un certo importo */
    public int deposit(int amount) {...}
```

# DBC per nuovo Account

Quando **creo** un account :

**PRE** - chiedo che `initialAmount` sia maggiore di `minBalance` per poter creare un conto:

- **uso requires per le precondizioni**

**POST** - assicuro che sul conto ci sarà l'intera somma inizialmente versata:

- **uso ensures per le postcondizioni**

```
/** crea un nuovo conto */  
//@ requires initialAmount >= minBalance;  
//@ ensures balance = initialAmount;  
public Account (int initialAmount){...}
```

# DBC per deposit

Quando **deposito**

**PRE**: chiedo che la somma depositata sia positiva

**POST**: assicuro che il bilancio verrà aumentato di quanto ho versato

- **uso `old` per riferirmi ad un valore prima dell'invocazione di un metodo**

```
//@ requires amount >= 0;  
//@ ensures balance == \old(balance) + amount;  
public int deposit(int amount) {...}
```

# DBC per withdraw

Quando ritiro un somma **amount**

**PRE:**  $0 \leq \text{amount} \leq \text{balance} - \text{minBalance}$

**POST:** il bilancio viene diminuito di **amount** e il valore restituito dal metodo è uguale al nuovo ammontare sul conto:

- **uso `\result` per riferirmi al valore restituito**

```
/*@requires amount >= 0
    && amount <= balance - minBalance;
    ensures balance == \old(balance) - amount
    && \result == balance;@*/
public int withdraw(int amount) {
    balance -= amount; return balance;}

```

# Contratto per withdraw

	<b>OBBLIGHI</b>	<b>BENEFICI</b>
<i>Cliente</i>	<p><i>(Soddisfare le precondizioni:)</i></p> <p>Assicurarsi che la somma non è ne' troppo grande ne' troppo piccola.</p>	<p><i>(dalle postcondizioni)</i></p> <p>Avere il conto aggiornato con la somma ritirata</p>
<i>Fornitore</i>	<p><i>(Soddisfare le postcondizioni)</i></p> <p>Aggiornare il conto togliendo la somma ritirata</p>	<p><i>(Dalle precondizioni)</i></p> <p>Calcoli più semplici: può assumere che la somma sia corretta</p>

# Operazione vs dichiarazione

istruzione

postcondizione

<code>balance = balance - amount</code>	<code>ensures balance == \old(balance) - amount</code>
<b>OPERAZIONE</b>	<b>DICHIARAZIONE</b>
Come?	Cosa?
<b>IMPLEMENTAZIONE</b>	<b>SPECIFICA</b>
Comando	Query
Istruzione	Espressione
Imperativo	Controllo



# Invariante per Account

In ogni caso

**INV**: un conto non potrà avere un bilancio inferiore al minimo

- **uso *invariant* per specificare un invariante**

```
public class Account {  
...  
/*@ invariant balance >= minBalance; @*/  
...  
}
```

# Uso del contratto

## 1. Documentazione

- può essere controllata sintatticamente, esportata in formato javadoc, ...

## 2. Controllo run-time

- le condizioni possono essere attivate o disattivate run-time (rallentano l'esecuzione)
  - attivate durante lo sviluppo e durante il testing
  - disattivate sul codice distribuito

## 3. Prova di correttezza del software

- dimostrare che il contratto è rispettato dall'implementazione

# Cenni di Correttezza del Software

**La correttezza è una nozione relativa:  
consistenza dell'implementazione rispetto la  
sua specifica.**

- nel nostro caso il contratto è la specifica

## **Notazione base:**

- A: istruzioni o programma
- P: precondizioni di A
- Q: postcondizioni di A
- correttezza :  $\{P\} A \{Q\}$  "Hoare triple"

ogni esecuzione di A che inizi in uno stato che  
soddisfa P termina in uno stato che soddisfi Q

# Contratto di SQRT

**Sia SQRT una routine per il calcolo della radice quadrata**

**La correttezza di questa routine richiede la scrittura del contratto:**

- A: ... algoritmo per calcolare  $y$  come radice quadrata di  $x$  (con precisione epsilon)
- P: l'input è positivo

$$\{x \geq 0\}$$

- Q:  $y$  approssima esattamente il quadrato di  $x$  con un errore pari a epsilon

$$\{\text{abs}(y^2 - x) \leq 2 * \text{epsilon} * y\}$$

# Esempio di correttezza

A: singola istruzione  $n := n + 9$

P:  $n > 5$

Q:  $n > 13$

Correttezza:  $\{n > 5\} n := n + 9 \{n > 14\}$

la dimostrazione di correttezza è complessa e difficilmente effettuabile con strumenti automatici

# Correttezza di una Classe

**Nel DBC una classe è corretta se sono corretti:**

**- operazione di creazione**

per ogni nuova istanza INV vale

- $\{\text{PRE}_{\text{costr}}\}$  constructor  $\{\text{INV} \wedge \text{POST}_{\text{costr}}\}$

**- ogni altro metodo OP**

se chiamo un metodo OP con PRE vero (e INV)  
allora dopo l'esecuzione del metodo vale POST e  
continua a valere INV

- $\{\text{PRE}_{\text{op}} \wedge \text{INV}\}$  OP  $\{\text{POST}_{\text{op}} \wedge \text{INV}\}$

- Abbiamo visto:

## In sintesi

- **JML** è un insieme di tool per l'uso del dbc in Java
  - in JML il contratto si mette nel commento precedente un metodo utilizzando `//@` o `/*@ @*/`;
  - le precondizioni sono espressioni java precedute da **requires** (es. `: requires x>0;`)
  - le postcondizioni sono precedute da **ensures**
- **\old** si usa per riferirsi ad un valore prima dell'esecuzione del metodo
- **\result** per riferirsi al valore restituito da un metodo
  - Cenni di prova di correttezza del software

- Nella prossima lezione vedremo:

- altri elementi di sintassi di JML



# Notazione base di JML

## JML permette di aggiungere come commenti particolari ad un classe Java:

- precondizioni ai metodi  
`//@ requires <espressione booleana>;`
- postcondizioni ai metodi  
`//@ ensures <espressione booleana>;`
- invarianti di classe  
`//@ invariant <espressione booleana>;`
- espressione booleana è una espressione java che può essere vera o falsa
- può contenere `\old`, `\result`



# Contratti incompleti

**Nota che un metodo può avere un contratto “debole” non ancora definitivo**

Ad esempio il metodo debit di account:

- richiede che l'ammontare sia positivo
- non garantisce nulla

```
/*@ requires amount >= 0; ensures true;@*/  
public int debit(int amount){...}
```

**Questa è la postcondizione è di default**

## Esempio con $\langle == \rangle$

**Nelle espressioni booleane di JML si può usare l'operatore  $\langle == \rangle$  che vuol dire "se e solo se"**

$A \langle == \rangle B$  è  $( A \ \&\& \ B ) \ || \ ( \ !A \ \&\& \ !B )$

Ad esempio se ho un metodo `minore` che prende in ingresso `j` e `n` e restituisce `true` se e solo se `j < n`:

```
//@ ensures \result <==> j < n;  
boolean minore(int j, int n) {return j < n;}
```

# Quantificatori

**Nelle espressioni di JML posso inserire anche quantificatori:**

- universali `\forall` - per ogni elemento
- esistenziali `\exists` - esiste un elemento
- generali come `\sum`, `\product`, `\min`, `\max`
- numerici `\num_of`

**La sintassi è**

`<quant.> <tipo> <var>; <range predicate>; <espressione>`

**Esempio**

Tutti gli studenti contenuti nel Collection `juniors` hanno advisor (dato dal metodo `getAdvisor()`):

```
(\forall Student s; juniors.contains(s);  
s.getAdvisor() != null)
```

# Esempio quantificatori (1)

**Scriviamo il contratto per un metodo che restituisce il minimo di un array**

```
public static int find_min (int a[])
```

**PRE:** l'array a non è null e ha almeno un elemento

**requires** `a != null && a.length >= 1;`

- se a è vuoto non saprei cosa restituire

**POST:** l'elemento restituito è effettivamente minore di tutti gli elementi dell'array:

**ensures** `(\forall int i; 0 <= i && i < a.length; \result <= a[i]);`

- questo non basta: find\_min potrebbe restituire MIN\_INT

## Esempio quantificatori (2)

**La post condizione completa è:**

**POST:** l'elemento restituito è effettivamente minore di tutti gli elementi dell'array e **appartiene all'array a**

- **esiste un elemento in a che è uguale a result**

**ensures**

```
(\forall int i; 0 <= i && i < a.length;  
    \result <= a[i])  
&& (\exists int i; 0 <= i && i < a.length;  
    \result == a[i]);
```

# non\_null

**Molti invarianti e pre e post condizioni richiedono che un certo riferimento non sia mai **null**.**

L'operatore **non\_null** sia usa per esprimere questi requisiti

## **Esempio**

```
public class Directory {  
private /*@ non_null @*/ File[] files;  
void createSubdir  
    (/*@ non_null @*/ String name) {...}  
Directory /*@ non_null @*/ getParent() {...}
```

# assert (1)

Per richiedere che una certa condizione sia verificata ad un certo punto all'interno del codice si può usare `assert`

## Esempio

```
if (i <= 0 || j < 0) { ...
} else if (j < 5) {
    //@ assert i > 0 && 0 <= j && j < 5;
    ...
} else {
    //@ assert i > 0 && j > 5;
    ...
}
```

## assert (2)

**Nota che anche Java (da 1.4) ha la parola chiave `assert` (con uguale scopo)**

in JML, `assert` è più espressivo.

Ad esempio se voglio richiedere che un vettore `a` non contenga `null`, in Java con `assert`:

```
for (n = 0; n < a.length; n++)  
    assert(a[n] != null);
```

In JML:

```
/*@ assert  
(\forall int i; 0 <= i && i < n ; a[i] != null);@*/
```

in Java però viene riconosciuto e controllato dal compilatore e per attivare il controllo run-time basta usare l'opzione `-ea` nel comando `java`



# Benefici di JML

- Le specifiche JML forniscono una documentazione **esplicita** del contratto
- Scrivendo le specifiche in JML rendi chiaro le **assunzioni** fatte a livello di progetto che andranno considerate quando si scrivono le implementazioni
- Rendono più facile la comprensione del codice e la sua manutenzione
- Possono essere analizzate da tool automatici

# Tools per JML

**JML compiler (**jmlc**): compila i file java per avere .class instrumentati**

**JML/Java interpreter (**jmlrac**):**

- esegue il controllo di tutte le asserzioni di JML: se una condizione non è verificata solleva una eccezione particolare

**Controllo statico del codice per la prova di correttezza con **escjava2****

**JML/JUnit unit test tool (**jmlunit**)**

**HTML generator (**jmldoc**)**

**Altri tool ....**

# In sintesi

- Abbiamo visto che in JML si può usare:
  - `<==>` per dire se e solo se;
  - `\forall` `\exists` come quantificatori
  - `non_null` per richiedere che un riferimento non sia mai null
  - `assert` per generiche condizioni
- Ricordate che in JML si usano:
  - `jmlc`: per compilare i file java contenenti i contratti di JML
  - `jmlrac`: per eseguire i file class con i controlli attivati
  - `jmldoc`: per generare la documentazione

