

# Design by contract

Angelo Gargantini

# Design by contract

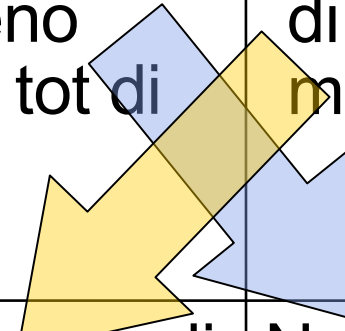
- Segue questa idea, soprattutto valida per l'OO:
  - L'interfaccia di un modulo definisce un contratto
- Cosa è un contratto? Un accordo tra cliente e contrattore (o fornitore)
- Un contratto:
  - Lega le due (o più) parti: fornitore e cliente
  - È esplicito (scritto)
  - Specifica gli obblighi e i benefici di entrambe le 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 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
- **Client**: chi usa il programma
- **Fornitore**: chi scrive il programma
- Esempio: una libreria

# Contratti per un oggetto

## PRE e POST condizioni

- Definire cosa ogni metodo richiede (obbligo per il cliente)
  - **precondition**
- e cosa fornisce il metodo (obbligo per il fornitore)
  - **postcondition**
- Precondizioni e postcondizioni potrebbero essere espressi usando la logica

# Esempio di un contenitore

- Operazione **insert(element)** (in un Array)
  - **no\_elements**: numero di elementi memorizzati
  - **size**: dimensione del vettore
- Precondition
  - **no\_elements < size**
- Postcondition
  - element is in table
  - **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 complicazione 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.



# Preconditions and postconditions

- **Precondition**

- Il cliente deve garantire la proprietà. Come fare?
- se `pre` è la precondizione per un metodo `m`, (per un oggetto `x`)

```
if (x.pre) x.m()  
else {//special treatment  
}
```

- Oppure siamo sicuri che `pre` vale prima della chiamata di `m`, in base al ragionamento sul programma

- **Postcondition**

- Il fornitore deve garantirle nell'implementazione del metodo

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

# Proprietà interne di una classe: INVARIANTS

- Possiamo specificare che una proprietà vale per tutte le istanze come *invariant*
- L'invariante è vero dopo la creazione dell'oggetto e dopo ogni operazione
- Esempio: **0 <= no\_elements <= size**
- L'invariante definisce un obbligo ulteriore
  - L'implementazione di una classe deve soddisfarla

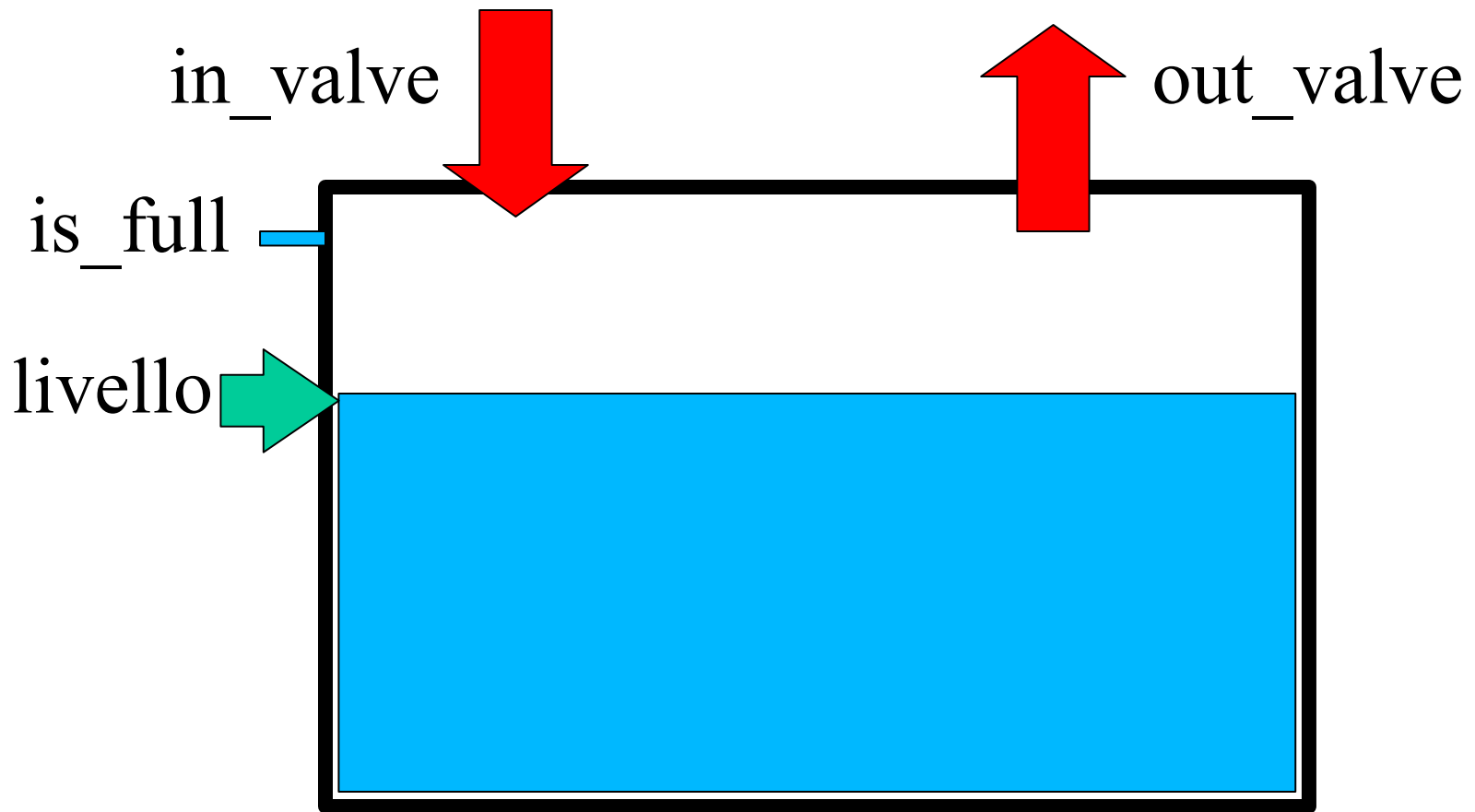
# Esempio concreto: Design by Contract in Eiffel

- Il contratto di ogni elemento software dovrebbe essere:
  - Esplicito
  - Parte dello stesso elemento software
- Il linguaggio Eiffel è stato costruito proprio seguendo i principi del DbC
- PRE e POST, INV nel linguaggio

# Esempio di classe Eiffel

```
deferred class VASCA inherit TANK
feature in_valve, out_valve: VALVE
fill is -- Fill the vat.
require
    in_valve.open; out_valve.closed
deferred (i.e. specified only not implemented)
ensure
    in_valve.closed; out_valve.closed; is_full
end
empty, is_full, is_empty, gauge, maximum,
...[Other features] ...
invariant
    is_full = ((gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum))
end
```

# VASCA



# Precondition

```
deferred class VASCA inherit TANK  
feature in_valve, out_valve: VALVE  
fill is -- Fill the vat.  
require  
    in_valve.open; out_valve.closed
```

# Postcondition /Invariant

**ensure**

```
in_valve.closed; out_valve.closed;  
is_full
```

**invariant**

```
is_full = ((livello >= 0.97 * maximum) and  
(livello <= 1.03 * maximum))
```

# Obblighi

Fill	<b><i>OBBLIGHI</i></b>	VANTAGGI
Cliente	<p>(dalle precondizioni:)</p> <p>Assicurarsi che la valvola d'ingresso sia aperta e quella di uscita chiusa</p>	<p>(dalle postcondizioni:)</p> <p>Ottiene il serbatoio pieno con le valvole chiuse</p>
Fornitore	<p>(dalle postcondizioni:)</p> <p>Riempie il serbatoio e chiude entrambe le valvole</p>	<p>(dalle precondizioni:)</p> <p>Può assumere che le valvole sia già nella posizione corretta</p>



# Contratti in analisi

- Nota che i contratti possono essere inseriti anche prima di fornire una vera implementazione del metodo
- **Documentazione delle classi:**
  - Dalle definizioni si può estrarre in modo automatico le precondizioni e le post condizioni e gli invarianti, che documentano cosa fa la classe
- Ma non solo: correttezza del sw

# Correttezza del Software

- La correttezza è una nozione relativa: consistenza dell'implementazione rispetto la sua specifica. (Questo assume che ci sia una specifica!)
- Notazione base: ( P, Q: asserzioni, cioè proprietà dello stato di calcolo. A: istruzioni o programma).

**{P} A {Q}**

**“Hoare triple”**

- Questo significa: Ogni esecuzione di A che inizi in uno stato che soddisfa P termina in uno stato che soddisfi Q.

# TRIPLE DI HOARE TRIPLES: un esempio

- $\{n > 5\} \ n := n + 9 \ \{n > 13\}$
- È vero?
- $\{n > 0\} \ x := 0 \ \{n > 0 \ \text{AND} \ x = 0\}$

# Specifica di una routine per il calcolo della radice quadrata

- A: ... algoritmo per calcolare  $y$  come radice quadrata di  $x$  (con precisione  $\epsilon$ )
- P:  $\{x \geq 0\}$
- Q:  $\{ \text{abs}(y^2 - x) \leq 2 * \epsilon * y \}$
- $y$  approssima esattamente il quadrato di  $x$  con un errore pari a  $\epsilon$

# Pre e Post condizioni

- Considera di essere un laureato in Informatica e sul mercatino cerchi lavoro per fare A (con  $\{P\}$  A  $\{Q\}$ ). Sei un po' sfaticato. Devi prendere P forte o debole. Q forte o debole?
- Due offerte speciali:
  1.  $\{False\}$  A  $\{...\}$
  2.  $\{...\}$  A  $\{True\}$

# Correttezza di una Classe

- Operazione di creazione
  - $\{\text{pre}_{\text{costr}}\}$  constructor  $\{\text{INV}'\}$
- Ogni altro metodo OP
  - $\{\text{pre}_{\text{op}} \wedge \text{INV}\}$  OP  $\{\text{post}_{\text{op}} \wedge \text{INV}'\}$

# Esempio: conto in banca

## CREATE

```
class ACCOUNT
  balance: INTEGER
  Min_balance: INTEGER is 1000

  make (initial_amount: INTEGER) is
    -- Set up account with initial_amount
    do balance := initial_amount end

  require
  large_enough: initial_amount >= Min_balance

  ensure
  set: balance = initial_amount
```

# Conto in banca 2 ADD

```
feature {NONE}
```

```
  -- PRIVATE Implementation of deposit and  
  withdrawal
```

```
add (sum: INTEGER) is
```

```
  -- Add sum to the balance (secret procedure).
```

```
do
```

```
balance := balance + sum
```

```
end
```



# Deposit

```
deposit (sum: INTEGER) is
  -- Deposit sum into the account
  do add (sum)
  end
require
  not_too_small: sum >= 0
ensure
increased: balance =
  old balance + sum
```

# Withdraw

```
withdraw (sum: INTEGER) is
-- Withdraw sum from the account
do
    add (-sum) -- i.e. balance := balance - sum
end
require
    not_too_small: sum >= 0
    not_too_big: sum <= balance - Min_balance
ensure
    decreased: balance = old balance - sum
```

# Invariant

`invariant`

`balance >= Min_balance`

# Contratto

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

# OPERZIONE vs DICHIARAZIONE

<b>do</b> <b>balance := balance – sum</b>	<b>ensure</b> <b>balance = old balance – sum</b>
OPERAZIONE	DICHIARAZIONE
Come?	Cosa?
IMPLEMENTAZIONE	SPECIFICA
Comando	Query
Istruzione	Espressione
Imperativo	Controllo

# Prova, testing e qualità

L'ideale sarebbe avere uno strumento che prova la correttezza del software

Esempio `{y < 5} for (i= 0;i <10;i++) y++ {y<15}`

Praticamente impossibile: uso delle condizioni durante lo sviluppo. Si può:

1. no assertion checking (under which assertions have no effect at all, serving as a form of standardized comments),
2. preconditions only (the default),
3. preconditions and postconditions,
4. all of the above plus class invariants, all assertions.

# Vantaggi TECNICI nell'uso del DbC

- *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.*

# 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 (e.g. “if the sum is negative, report an error...”)
- Una violazione dei un contratto è la manifestazione di un DIFETTO (“BUG”)



# Inheritance

- Sottoclassi possono aggiungere attributi e metodi (nuovi pre e post condizioni)
- Possono ridefinire i metodi
- Grazie al polimorfismo quando chiamo un metodo non so se chiamo quello della sopraclasse o di una sottoclasse
- Esempio
  - X: ACCOUNT particolare (con FIDO)
  - X. WITHDRAW(100)

# Inheritance

- Oltre ai vincoli sugli argomenti ho vincoli sui contratti: EREDITARIETA' come SOTTOCONTRATTO
- Precondizioni indebolite
  - preclass -> presubclass
  - In questo modo il client può chiamare la sottoclasse (che richiede di meno)
- Postcondizioni rafforzate
  - postsubclass -> postclass
  - Viene fornito di più

# Esempio : withdraw per conto con fido

```
require
not_too_small: sum >= 0
not_too_big: sum <= balance - Min_balance + FIDO
ensure
  decreased: balance = old balance - sum
```

not\_too\_big superclass => not\_too\_big subclass

# Alcuni sistemi esistenti oltre Eiffel

GNU Nana: (C e C++)

iContract (Java)

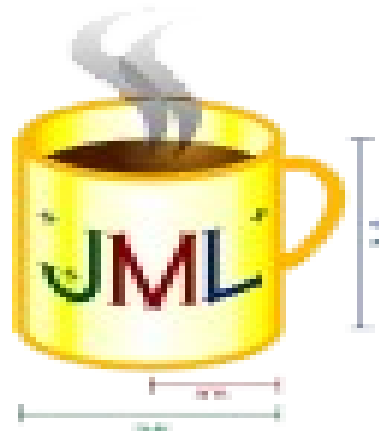
Java assertion in jdk 1.4

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

Jass: Java Assertions:

<http://semantik.informatik.uni-oldenburg.de/~j>

JML; The Java Modeling Language (JML)



# Esempio 1 (nana)

```
int square_root(int x)
/*@
assume x >= 0;
return y where y >= 0;
return y where y*y <= x && x <
    (y+1) * (y+1);
@*/
```

...

assume: preconditione

return: postcondizione

# Con Java assertion jdk 1.4: es. precondition in un metodo privato

```
/** * Sets the refresh interval.  
 * @param interval refresh interval in milliseconds.  
 */  
private void setRefreshInterval(int interval) {  
    //Confirm adherence to precondition in nonpublic method  
    assert(interval > 0 && interval <= MAX_REFRESH_RATE);  
    // Set the refresh interval  
    ...}
```

assertion fallirà se  $interval \leq 0$  o se  $interval > MAX\_REFRESH\_RATE$ . Vorrebbe dire che c'è un baco nella libreria!

Vantaggio rispetto le eccezioni normali: le asserzioni si possono abilitare o disabilitare !

# Problemi di assert

- ASSERT non stabilisce un contratto
  - I clienti non vedono gli assert come parte dell'interfaccia
  - Gli assert non hanno una specifica della semantica (se non quella di Java)
- Non so se assert si riferisce ad una pre o post cond. o ad un invariante
- Niente supporto all'ereditarietà
- No documentazione automatica

# Tipi di asserzioni con Jass

Pre- and postconditions for methods

Invariants for classes

Invariants and variants for loops

Check statements

Forall and exists expressions

E tante altre funzionalità:

- Inheritance of assertions (refinement)

- JavaDoc support

- Comments in assertions

- Trace-Assertions

- Rescue and retry statements