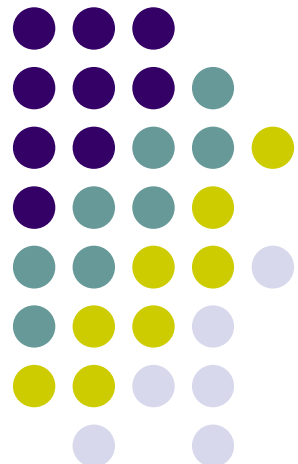
Design by contract

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.

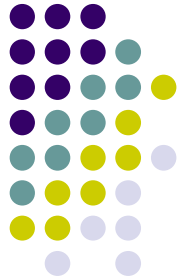
Bibliografia

- 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 …

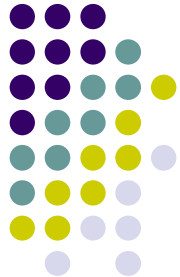


- Oggetto del contratto: il software, in OO, un insieme di classi
- **Client**: chi usa il programma
- **Fornitore**: chi scrive il programma
- Esempio: una libreria

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

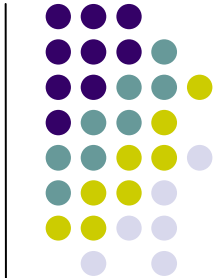
Esempio di un contenitore



- Operazione insert(element) (in un Vector)
  - **no_elements**: numero di elementi memorizzati
  - **size**: dimensione del vettore
- Precondition
  - **no_elements < size**
- Postcondition
  - element is in table
  - **no_elements' = no_elements +1**

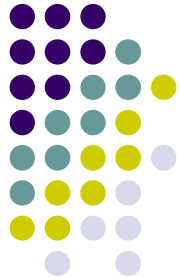
    no_elements' indica il valore dopo l'operazione

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'e' 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

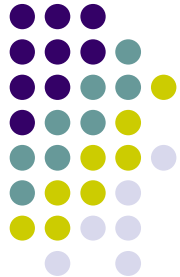


- Precondition
  - Il cliente deve garantire la proprietà. Come fare?
  - se pre è la precondizione per un metodo m, (per un oggetto x)
    ```
    if (x.pre) x.m()
    else //special treatment
    }
    ```
  - Oppure siamo sicuri che `pre` vale prima della chiamata di m, in base al ragionamento sul 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

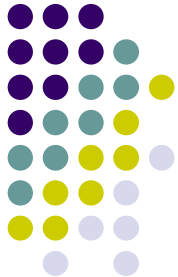
- Possiamo specificare che una proprietà vale per tutte le istanze come *invariant*
- L'invariante è vero dopo la creazione dell'oggetto e dopo ogni operazione
- Esempio: **0 <= no_elements <= size**
- L'invariante definisce un obbligo ulteriore
  - L'implemetazione di una classe deve soddisfarla

Esempio concreto: Design by Contract in Eiffel



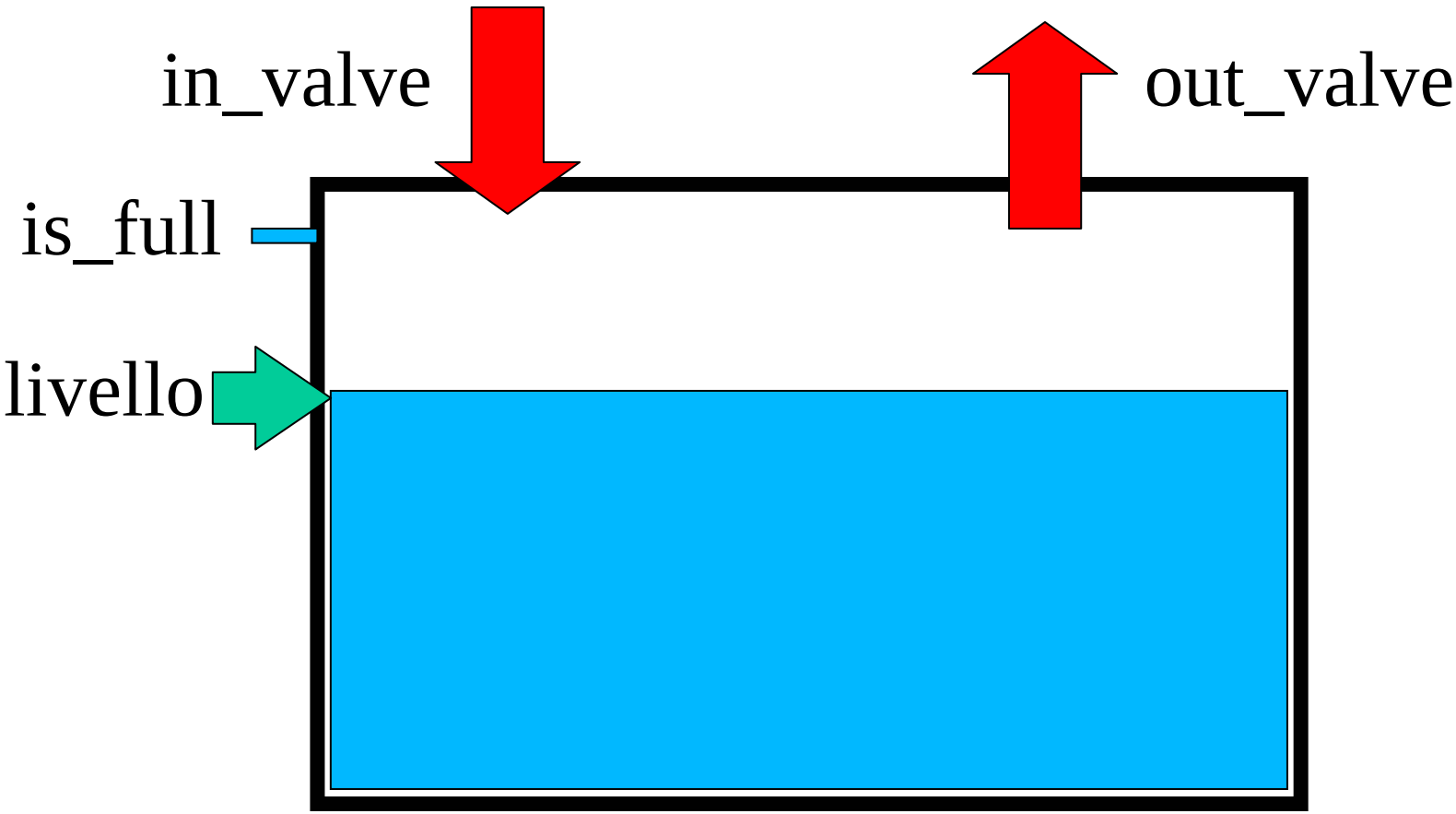
- Il contratto di ogni elemento software dovrebbe essere:
  - Esplicito
  - Parte dello stesso elemento software
- Il linguaggio Eiffel è stato costruito proprio seguendo i principi del DbC
- PRE e POST, INV nel linguaggio

Esempio di classe Eiffel

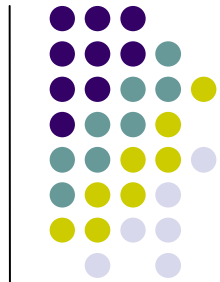


```
deferred class VASCA inherit TANK
feature in_valve, out_valve: VALVE
fill is  -- Fill the vat.
require
    in_valve.open; out_valve.closed
deferred (i.e. specified only not implemented)
ensure
   in_valve.closed; out_valve.closed; is_full
end
empty, is_full, is_empty, gauge, maximum,
...[Other features]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

VASCA



Precondition

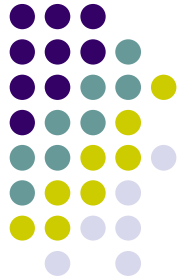


```
deferred class VASCA inherit TANK

feature in_valve, out_valve: VALVE

fill is  -- Fill the vat.
require
    in_valve.open; out_valve.closed
```

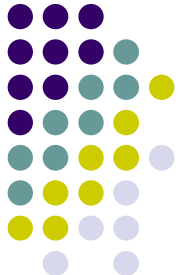
Postcondition /Invariant



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

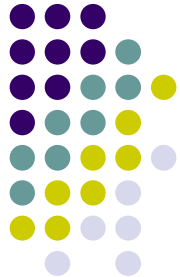
```
invariant
is_full = ((livello >= 0.97 * maximum) and
  (livello <= 1.03 * maximum))
```

Obblighi



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

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

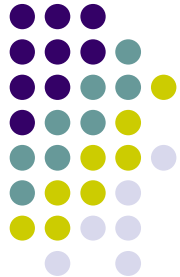
- La correttezza è una nozione relativa: consistenza dell'implementazione rispetto la sua specifica. (Questo assume che ci sia una specifica!)
- Notazione base: (P, Q: asserzioni, cioè proprietà dello stato di calcolo. A: istruzioni o programma).

{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.

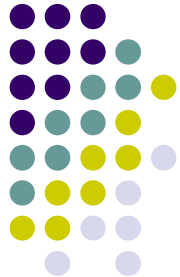
TRIPLE DI HOARE TRIPLES: un esempio



- {n > 5} n := n + 9 {n > 13}

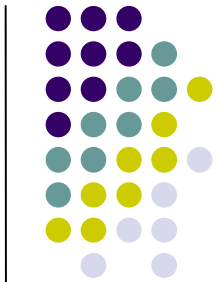
- È vero?
- {n>0} x:= 0 {n>0 AND x =0}

Specifica di una routine per il calcolo della radice quadrata



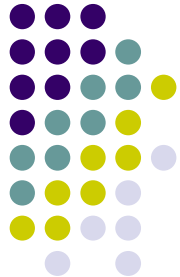
- A: ... algoritmo per calcolare y come radice quadrata di x (con precisione epsilon)
- P: {x >= 0}
- Q: {abs (y ^ 2 – x) <= 2 * epsilon * y}
- y approssima esattamente il quadrato di x con un errore pari a epsilon

Pre e Post condizioni



- Considera di essere un laureato in Ing. 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
  - $\{prec_{costr}\}$ constructor {INV'}
- Ogni altro metodo OP
  - $\{pre_{op} \wedge INV\}$ OP $\{post_{op} \wedge INV'\}$

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
```

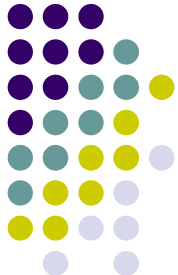
Withdraw

```
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

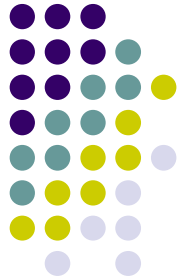
```
invariant
balance >= Min_balance
```



Contratto

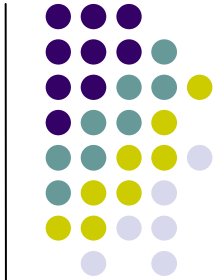
withdraw	**OBBLIGHI**	**BENEFICI**
Client	(***Soddisfare le precondizioni***:) **Assicurarsi che la somma non è ne' troppo grande ne' troppo piccola.**	(***dalle postcondizioni***) **Avere il conto aggiornato con la somma ritirata**
Supplier	(***Soddisfare le postcondizioni)*** **Aggiornare il contro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

OPERZIONE vs DICHIARAZIONE



do **balance := balance – sum**	**ensure** **balance = old balance – sum**
OPERAZIONE	DICHIARAZIONE
Come?	Cosa?
IMPLEMETAZIONE	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.

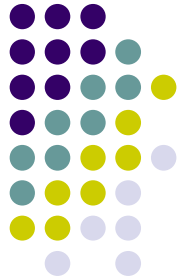
Vantaggi TECNICI nell'uso del DbC

- *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
  - postsublass -> postclass
  - Viene fornito di più

Esempio : withdraw per conto con fido

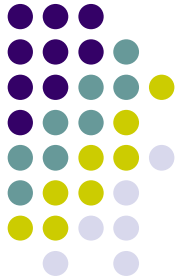
```
require
not_too_small: sum >= 0
not_too_big: sum <= balance – Min_balance + FIDO
  ensure
  decreased: balance = old balance – sum
```

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/~jass/

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);
@*/
...
```

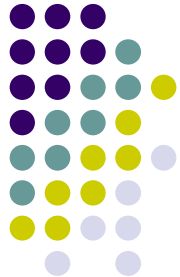
- assume: precondizione
- return: postcondizione

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 e' <=0 o se e' > di MAX_REFRESH_RATE. Vorrebbe dire che c'e' un baco nella libreria!
- Vantaggio rispetto le eccezioni normali: le asserzioni si possono abilitare o disabilitare !

Problemi di assert

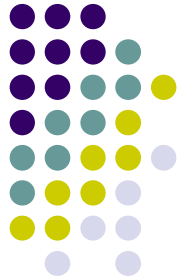


- ASSERT non stabilisce un contratto
  - I clienti non vedono gli assert come parte dell'interfaccia
  - Gli assert non hanno una specifica della semantica (se non quella di Java)
- Non so se assert si riferisce ad una pre o post cond. o ad un invariante
- Niente supporto all'ereditarietà
- No documentazione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

Tipi di asserzioni con Jass

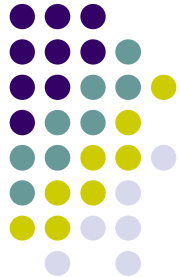
- Pre- and postconditions for methods
- Invariants for classes
- Invariants and variants for loops
- Check statements
- Forall and exists expressions
- E tante altre funzionalità:
  - Inheritance of assertions (refinement)
  - JavaDoc support
  - Comments in assertions
  - Trace-Assertions
  - Rescue and retry statements

Example for a precondition:



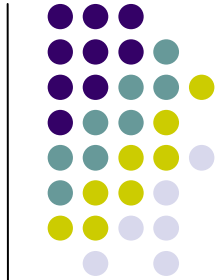
```
public void addElement (Object o) {
/** require !isFull(); o != null;
  **/
...
}
```

Example for a postcondition:



```
public void addElement (Object o) {
...
/** ensure !isEmpty() &&
  contains(o); **/
}
```

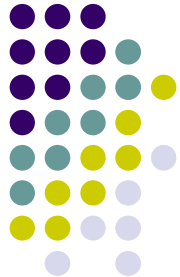
Uso di OLD



- Le post condition possono contenere tre costrutti: **Old**, **changeonly** and **Result**
- Lo stato dell'oggetto viene memorizzato in **Old** all'entrata di un metodo

```
public void addElement (Object o) {
...
/** ensure !isEmpty() && contains(o);
  Old.count == count-1; **/
}
```

Esempi complessi con Jass ...

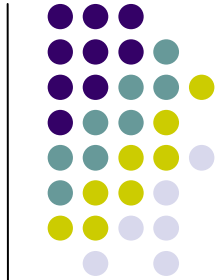


```
/** Each employee must be in the employment list
 * of all his employers
 *
 * @invariant employees_ != null
 * implies
 * forall Employee e in employees_.elements() |
 * exists Employer c in e.getEmployers() |
 * c == this
 */
class Employer {
protected Vector employess_; // of Employee
...
}
```

Come si usa jass

1. Scrivi i tuoi file *.jass
2. Precompila i tuoi sorgenti con jass (questo crea diversi file sorgenti *.java)
3. Compila i sorgenti java con un compilatore java (es. Javac)
4. Esegui normalmente il tuo bytecode (la libreria jass.jar deve essere nel classpath)

Come usare jass



- per eseguire il punto 2 puoi:
  - se X è una tua sottodirectory
  - metti i tuoi file jass in X (supponi che il tuo file si chiami Prova.jass)
  - metti il file jass.jar nella sottodirectory `jass_src` di X
  - apri una shell cygwin/dos e fai cd fino a X
  - esegui java -classpath "jass_src/jass.jar;." jass.Jass Prova.jass
  - se java non è nel tuo path metti il nome completo

EIFFEL

- …. todo