Marco Faella

Programmazione tramite contratti

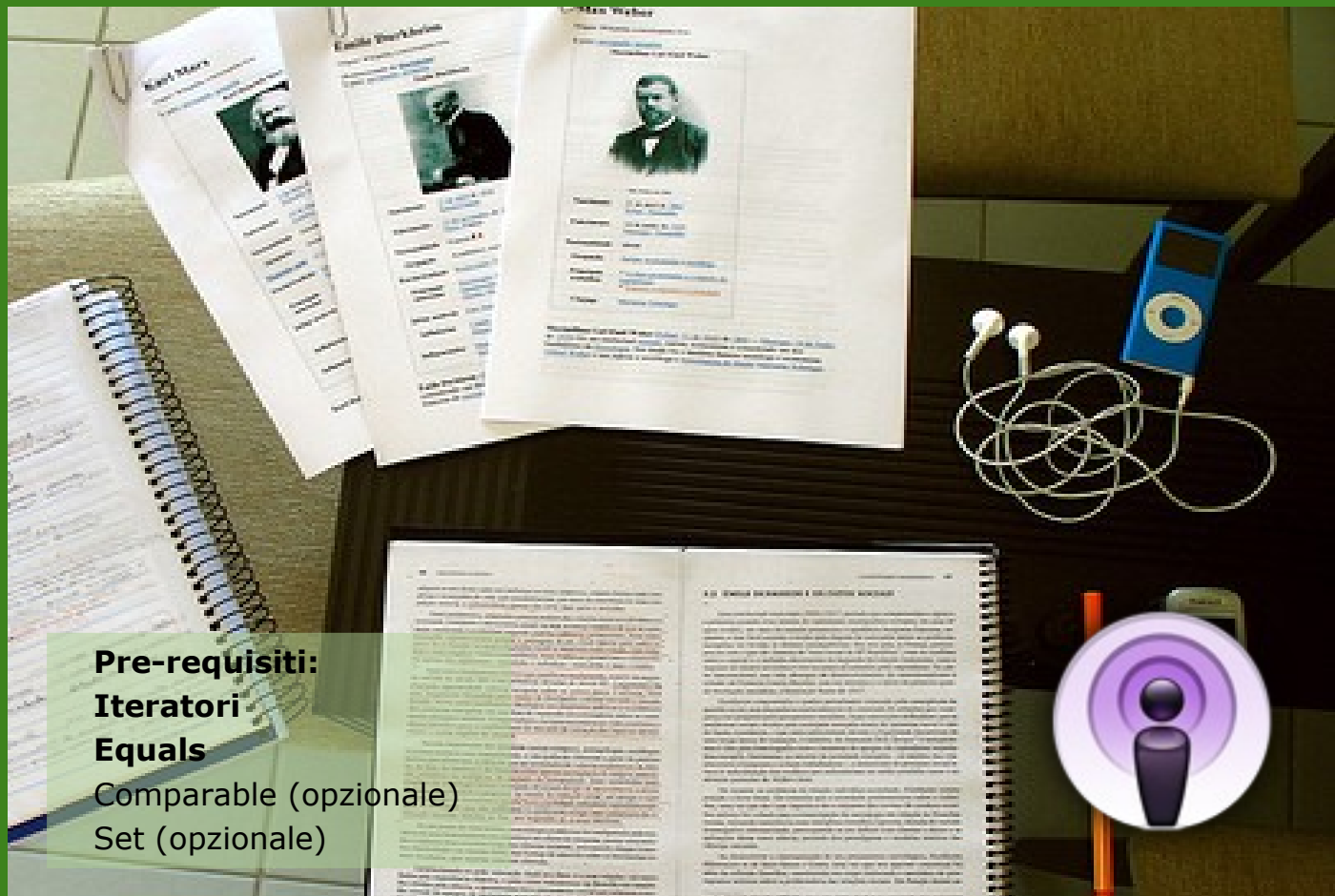
Lezione n. 23

Parole chiave:
Java

Corso di Laurea:
Informatica

Insegnamento:
Linguaggi di Programmazione II

Email Docente:
faella.didattica@gmail.com



Pre-requisiti:
Iteratori
Equals
Comparable (opzionale)
Set (opzionale)

Programmazione tramite contratti

- L'idea della programmazione guidata da contratti (in inglese, *Design by contract*) è stata proposta alla fine degli anni '80 da Bertrand Meyer, studioso di linguaggi di programmazione e creatore del linguaggio *Eiffel*

- L'idea consiste nell'applicare al software, in particolare a quello orientato agli oggetti, la nozione comune di *contratto*
- Comunemente, un contratto è un accordo in cui le parti si assumono degli *obblighi* in cambio di *benefici*

- Si veda:

  B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988(1st) - 1997(2nd)

- Applicato ad un **metodo** di una classe, un contratto specifica quale **compito** il metodo promette di svolgere e quali sono le **pre-condizioni** richieste

- Dal punto di vista del client:
  - il compito svolto dal metodo è un beneficio
  - le pre-condizioni sono obblighi

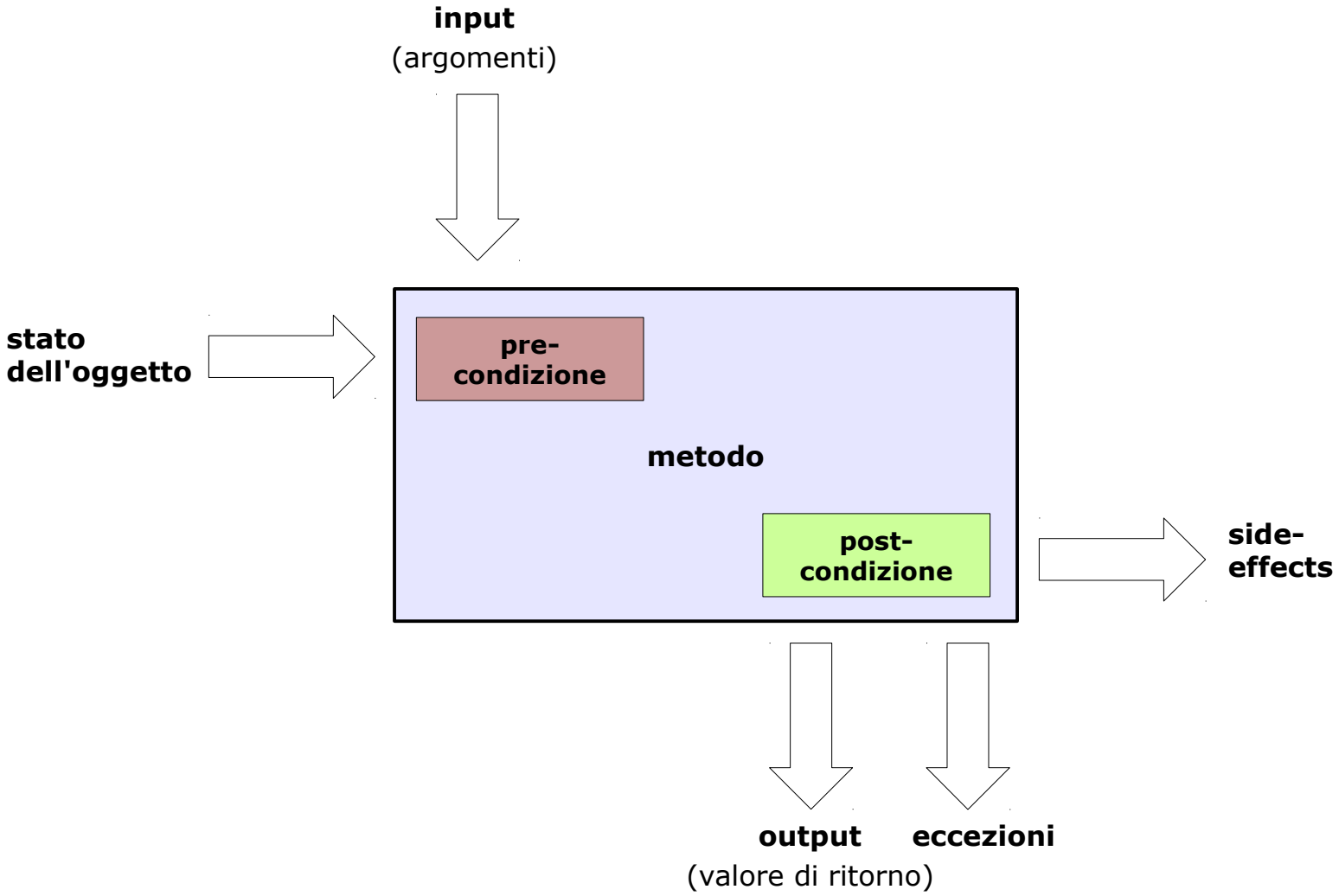
- Dal punto di vista del metodo:
  - il compito da svolgere è un obbligo
  - le pre-condizioni sono un beneficio (agevolano o consentono il compito)

- La **pre-condizione** riguarda:
  - i valori passati al metodo come parametri attuali
  - lo stato dell'oggetto su cui viene invocato il metodo

- Il compito che il metodo assolve, detto anche **post-condizione**, riguarda:
  - il valore restituito
  - il nuovo stato dell'oggetto
  - qualsiasi altro *effetto collaterale (side effect)*

- Esempi di effetti collaterali:
  - stampare un messaggio a video
  - scrivere in un file
  - terminare

- Nota: tutto quello che fa un metodo e che è visibile all’esterno, eccetto restituire un valore, è considerato un effetto collaterale


input
(argomenti)
stato
dell'oggetto
pre-
condizione
metodo
post-
condizione
side-
effects
output
(valore di ritorno)
eccezioni

- Inoltre, il contratto può specificare come reagisce il metodo nel caso in cui le pre-condizioni non siano soddisfatte dal chiamante, analogamente ad una **penale** in un comune contratto

- In Java, la penale tipica consiste nel lancio di un'eccezione non verificata

La **pre-condizione** deve essere ***verificabile*** da parte del client

Ad esempio, un metodo che si interfaccia con una stampante *non* può prevedere come *pre-condizione* che la stampante abbia carta a sufficienza, perché questa non è una condizione che il client può verificare

In un caso come questo, il metodo lancerà un'eccezione verificata, e ciò sarà parte della sua *post-condizione*

- La **pre-condizione** descrive l'uso corretto del metodo
  - La sua violazione costituisce **errore di programmazione da parte del client**

- In modo simmetrico, la **post-condizione** descrive l'effetto atteso del metodo
  - La sua violazione indica un **errore di programmazione nel metodo stesso**

Documentare un contratto

Esprimere un contratto

- In qualsiasi linguaggio di programmazione, **solo una parte del contratto può essere espressa nel linguaggio stesso,** mentre il resto sarà indicato in linguaggio naturale, ad esempio nei commenti relativi al metodo

- Ad esempio, il numero e tipo dei **parametri formali** di un metodo fanno parte delle **pre-condizioni**
- Il **tipo di ritorno** fa parte *della* **post-condizione**

- Alcuni linguaggi, come *Eiffel*, o strumenti di analisi statica, come *ESC/Java*, permettono di specificare formalmente una parte considerevole del contratto di un metodo

Contratto e clausola “throws” in Java

- A prima vista, potrebbe sembrare che l'eventuale dichiarazione “throws ...” indichi il tipo di eccezione che viene sollevata in caso di violazione della **pre-condizione**
- In realtà, la dichiarazione throws andrebbe usata solo per le eccezioni **verificate**, mentre la violazione di una **pre-condizione** richiede un'eccezione *non verificata*, perché si tratta di un errore di programmazione
- La clausola throws, usata correttamente con eccezioni verificate, esprime invece parte della **post-condizione**
- Infatti, è responsabilità della **post-condizione** descrivere le eventuali condizioni anomale che possono dare luogo ad eccezioni verificate

Contratti ben definiti

- In un contratto **ben definito**, la pre-condizione contiene tutte le proprietà che servono al metodo di svolgere il suo compito (cioè, per realizzare la sua post-condizione)

- Ad esempio, si considerino i seguenti contratti:

  1) Il metodo **max** accetta due oggetti e restituisce il maggiore tra i due

  - Il contratto *non è ben definito* perché non è chiaro quale relazione d'ordine utilizzare
  - Una versione corretta: il metodo **max** accetta due oggetti *di una classe dotata di ordinamento naturale* e restituisce il maggiore tra i due

Contratti ben definiti

- In un contratto **ben definito**, la pre-condizione contiene tutte le proprietà che servono al metodo di svolgere il suo compito (cioè, per realizzare la sua post-condizione)

- Ad esempio, si considerino i seguenti contratti:

  1) Il metodo **max** accetta due oggetti e restituisce il maggiore tra i due

  - Il contratto *non è ben definito* perché non è chiaro quale relazione d'ordine utilizzare
  - Una versione corretta: il metodo **max** accetta due oggetti *di una classe dotata di ordinamento naturale* e restituisce il maggiore tra i due

  2) Il metodo **reverse** accetta una collezione e inverte l'ordine dei suoi elementi

  - Il contratto *non è ben definito* perché non tutte le collezioni sono ordinate; ad esempio, l'operazione non ha senso su un HashSet
  - Una versione corretta: il metodo **reverse** accetta una *lista* e inverte l'ordine dei suoi elementi

Contratto di Iterator

- Presentiamo il contratto dei metodi dell'interfaccia **`Iterator<T>`**

- Per un'interfaccia, il contratto è particolarmente importante, perché rappresenta la sua vera *raison d'etre*

Metodo hasNext `boolean hasNext()`

- *Pre-condizione:* nessuna, tutte le invocazioni sono lecite
- *Post-condizione:*
  - restituisce true se ci sono ancora elementi su cui iterare (cioè, se è lecito invocare `next`) e false altrimenti
  - non modifica lo stato dell'iteratore

Contratto di Iterator

- Presentiamo il contratto dei metodi dell'interfaccia **`Iterator<T>`**

- Per un'interfaccia, il contratto è particolarmente importante, perché rappresenta la sua vera *raison d'etre*

Metodo next `T next()`

- *Pre-condizione:* ci sono ancora elementi su cui iterare (cioè, un'invocazione a `hasNext` restituirebbe true)
- *Post-condizione:*
  - restituisce il prossimo oggetto della collezione
  - fa avanzare l'iteratore all'oggetto successivo, se esiste
- Trattamento degli errori (penale):
  - solleva `NoSuchElementException` (non verificata) se la pre-condizione è violata

Metodo remove `void remove()`

- *Pre-condizione:*
  - prima di questa invocazione a `remove`, è stato invocato `next`, rispettando la sua pre-condizione
  - dall'ultima invocazione a `next`, non è stato già chiamato `remove`
- *Post-condizione:*
  - rimuove dalla collezione l'oggetto restituito dall'ultima chiamata a `next`
  - non modifica lo stato dell'iteratore (cioè, non ha influenza su quale sarà il prossimo oggetto ad essere restituito da `next`)
- Trattamento degli errori (penale):
  - solleva `IllegalStateException` (non verificata) se la precondizione è violata

- Inoltre, il metodo remove viene indicato come "opzionale" dalla documentazione
  - ciò significa che le implementazioni di Iterator non sono obbligate a supportare questa funzionalità
  - se un'implementazione non vuole supportarla, deve far lanciare al metodo remove l'eccezione `UnsupportedOperationException` (non verificata)

Esercizio (esame 21/4/2008, #4)

- Individuare l'output del seguente programma.
- Dire se la classe CrazyIterator rispetta il **contratto** dell'interfaccia Iterator
  - in caso negativo, giustificare la risposta

```
public class CrazyIterator implements Iterator {
   private int n = 0;

   public Object next() {
     int j;
     while (true) {
       for (j=2; j<=n/2 ;j++)
         if (n % j == 0) break;
       if (j > n/2) break;
       n++;
     }
     return new Integer(n);
   }
   public boolean hasNext() {
     n++;
     return true;
   }
   public void remove()  {
      throw new RuntimeException();
   }
```

```
   public static void main(String[] args) {
         Iterator i = new CrazyIterator();

         while (i.hasNext() &&
                (Integer)i.next()<10) {
            System.out.println(i.next());
         }
      }
   }
```

Contratti ed overriding

- Il contratto di un metodo andrebbe considerato vincolante anche per le eventuali ridefinizioni del metodo (overriding)

- Il contratto di un metodo ridefinito in una sottoclasse deve assicurare il **principio di sostituibilità**:

Le chiamate fatte al metodo originario rispettando il suo contratto
devono continuare ad essere corrette
anche rispetto al contratto ridefinito in una sottoclasse.

- Questo è un caso particolare del noto **principio di sostituzione di Liskov** (enunciato da Barbara Liskov nel 1987)

Se un client usa una classe A,
deve poter usare allo stesso modo le sottoclassi di A,
senza neppure conoscerle.

La regola contro-variante

- Quindi, ogni ridefinizione del contratto dovrebbe **offrire** al client almeno **gli stessi benefici**, richiedendo **al più gli stessi obblighi**

- In altre parole, un overriding può **rafforzare la post-condizione** (garantire di più) e **indebolire la pre-condizione** (richiedere di meno)

- Tale condizione prende il nome di ***regola contro-variante***, perché la pre-condizione può variare in modo opposto alla post-condizione

Regole dell'overriding in Java

- Dato un metodo con la seguente intestazione:

```
Vis T foo(U1, …, Un) throws E1, …, Em
```

- Cosa può cambiare in un overriding e come?
- E soprattutto, perché?

- Le regole derivano in buona parte dal principio di sostituibilità

Regole dell'overriding in Java

- Dato un metodo con la seguente intestazione:

```
Vis T foo(U1, …, Un) throws E1, …, Em
```

- Cosa può cambiare in un overriding e come?

```
Vis' T' foo(U1, …, Un) throws F1, …, Fk
```

- Vis' può essere più ampia di Vis
- T' può essere sottotipo di T
- Per ogni i=1...k, Fi è sottotipo di qualche Ej

Java e la regola contro-variante

- Le regole dell'overriding in Java rispecchiano solo *una metà* della regola contro-variante:
  - Infatti, **il tipo di ritorno** di un metodo può diventare **più specifico** nell'overriding, rafforzando quindi la post-condizione
  - Invece, **il tipo dei parametri non può cambiare**, mentre la regola contro-variante prevederebbe che potessero diventare più generali

Ad esempio, consideriamo il seguente contratto per il metodo con intestazione

```
public double avg(String str, char ch)
```

Pre-condizione: la stringa str non è null e non è vuota; il carattere ch è presente in str
Post-condizione: restituisce il numero di occorrenze di ch in str, diviso la lunghezza di str

Quali dei seguenti contratti sono overriding validi?

1) Pre-condizione: la stringa str non è null
   Post-condizione: se la stringa è vuota, restituisce 0, altrimenti restituisce il numero di occorrenze di ch in str, diviso la lunghezza di str

2) Pre-condizione: nessuna
   Post-condizione: se la stringa è null oppure vuota, restituisce 0, altrimenti restituisce il numero di occorrenze di ch in str, diviso la lunghezza di str

3) Pre-condizione: la stringa str non è null e non è vuota
   Post-condizione: restituisce il numero di occorrenze di ch in str

Contratti in Javadoc

- *Javadoc* è un tool che estrae documentazione dai sorgenti Java e la rende disponibile in vari formati, tra cui l'HTML
- Javadoc estrae informazioni dalle dichiarazioni e da alcuni commenti speciali, racchiusi tra /** e */
- All'interno di questi commenti, si possono utilizzare dei marcatori (tag) per strutturare la documentazione
- In particolare, il contratto di un metodo andrebbe indicato in un commento che precede il metodo, utilizzando almeno i tag *@param*, *@return* e *@throws*

- Ad esempio, consideriamo un metodo che calcola la **radice quadrata** di un numero dato
  - La **pre-condizione** consiste nel fatto che l'argomento deve essere un numero non negativo
  - La **post-condizione** assicura che il valore restituito è la radice quadrata dell'argomento
  - La reazione all'errore consiste nel lancio dell'eccezione IllegalArgumentException

- Utilizzando Javadoc, il suo contratto può essere sinteticamente indicato come segue

```
/**
 * Calcola la radice quadrata.
 *
 * @param  x numero non-negativo di cui si vuole calcolare la radice quadrata
 * @return   la radice quadrata di x
 * @throws   IllegalArgumentException se x è negativo
 */
public double sqrt(double x)
{
   if (x<0) throw new IllegalArgumentException();
   ...
}
```

```
class
    ACCOUNT

feature
    balance: INTEGER
            -- Current balance

    deposit_count: INTEGER

feature
    deposit (sum: INTEGER)
            -- Add `sum' to account.
        require
            non_negative: sum >= 0
        do
            ... method body ...
        ensure
            one_more_deposit: deposit_count = old deposit_count + 1
            updated: balance = old balance + sum
        end
```

Named precondition

Named postcondition

Queste asserzioni vengono controllate a runtime, se attivate a tempo di compilazione

- In alcuni casi, è conveniente distinguere due parti del contratto:
  - la parte ***generale***, che si applica anche a tutte le possibili ridefinizioni del metodo
  - e la parte ***locale***, che si applica solo alla versione originale del metodo, ma non costituisce un obbligo per le ridefinizioni

- Esempio: metodo equals (a seguire)

Contratto di equals

- Presentiamo il contratto del metodo *equals* di Object, parafrasando la documentazione ufficiale (Java Platform Standard Edition 7)

- Poiché il metodo equals è destinato ad essere ridefinito nelle sottoclassi, il contratto è diviso in parte *generale* e parte *locale*

- La parte locale descrive il comportamento della versione originale presente in Object, ma *non è vincolante* per le sottoclassi di Object

Parte generale

- **Pre-condizione:** nessuna, tutte le invocazioni sono lecite
- **Post-condizione:**
  - il metodo rappresenta una relazione di equivalenza tra istanze non nulle
  - l'invocazione x.equals(y) restituisce vero se y è diverso da null ed è considerato "equivalente" a x
  - invocazioni ripetute di equals su oggetti il cui stato non è cambiato devono avere lo stesso risultato (*coerenza temporale*)

Parte locale

- **Pre-condizione:** nessuna
- **Post-condizione:**
  - l'invocazione x.equals(y) è equivalente a x==y (quando x non sia `null`)