Progettazione

“By Contract”

- **Documentazione di progetto**
  - Contratti per i metodi
- **Correttezza dell'implementazione**
  - invarianti per le classi
- **Verifica**
  - asserzioni
- **Eccezioni**

Progettazione *By Contract*

- **Contratti per i metodi**
  - formalizzazione del comportamento di ciascun metodo per controllare le interazioni con i clienti
- **Invarianti di classe**
  - Vincoli di buona formazione degli oggetti della classe, necessari per garantire la correttezza dei metodi
- **Asserzioni**
  - Controllo dinamico di contratti ed invarianti

Contratti per i metodi

- **La dichiarazione di un metodo specifica solo il tipo**
- **Un contratto specifica il comportamento;**
  - quale servizio il metodo implementa, sotto quali condizioni
- **Specifiche informali e/o incomplete = problemi:**
  - Ambiguità di interpretazione
  - Contraddizione con altri contratti
- **Specifiche formali garantiscono/permettono**
  - Precisione e univocità di interpretazione
  - Controlli a run-time su violazioni del contratto e/o errori di uso
  - Facilità di ragionamento su correttezza e uso dei metodi

Continua…

Contratti per i metodi

- **Esistono linguaggi di specifica che permettono la specifica completa del comportamento di componenti**
- **Nostro obiettivo**
  - Specifica formale di parte del comportamento
  - Scopo: controllare gli aspetti rilevanti del comportamento e dell'implementaizione
  - Verifica della specifica mediante asserzioni

Continua…

Contratti per i metodi

Precondizioni:

- condizioni che devono essere soddifatte al momento della chiamata
- vincoli per il *chiamante*, che deve assicurare che le condizioni siano vere

Postcondizioni:

- condizioni che devono essere soddifatte al momento del ritorno dalla chiamata
- vincoli per il *chiamato*, che deve dare garanzie al chiamante

Continua…

pre/post condizioni

- **Documentate con commenti in stile Javadoc**
  - due tag speciali

```
/**
   @pre precondizione
   @post postcondizione
*/
 public void metodo( ... ) { ... }
```

- **Non supportate da javadoc**
  - altri strumenti: JML, XjavaDoc, iContract, …

Continua…

pre/post condizioni

- **Espressioni booleane**
  - specificate in sintassi Java
- **Estensioni sintattiche utili alla specifica**
  - **`@result:`** il valore restituito da un metodo
  - ***`expr`*`@pre:`** il valore dell'espressione prima dell'esecuzione del metodo
  - **`@nochange:`** il metodo non modifica this
  - **`@forall:`*`dom`* `@`*`expr`**, **`@exists:`*`dom`* `@`*`expr`** espressioni quantificate (su un certo dominio)
  - **`==>, <=>:`** implicazione ed equivalenza logica

Continua…

pre/post condizioni

- **Specificate da una o più clausole `@pre` e `@post`**
  - in presenza di più clausole la condizione è definita dalla congiunzione di tutte le clausole
- **Specifiche non sempre esaustive**
  - la specifica delle pre/post condizioni integrata dalla documentazione standard
  - quando il commento iniziale esprime in modo preciso la semantica evitiamo di ripeterci nella specifica
- **Condizioni espresse sempre in termini dell'interfaccia pubblica**

Continua…

Esempio: MySequence<T>

```
interface MySequence<T> {

    // metodi accessors
    int size();
    boolean isEmpty();
    T element(int i);
    T head();
    . . .
    // metodi mutators
    void insert(T item, int i)
    void insertHead(T item);
    T remove(int i);
    T removeHead();
    . . .
}
```

size()

```
/**
 * Restituisce la dimensione della sequenza
 * @result = numero di elementi nella sequenza
 * @pre true
 * @post @nochange
 */
 public int size();
```

- **La precondizione `true` è sempre soddisfatta**
  - un metodo con questa precondizione può sempre essere invocato
- **La postcondizione indica solo l'assenza di side effects (il commento completa la specifica)**

isEmpty()

- **pre/post condizioni sono (devono essere) espresse in termini dell'interfaccia pubblica**
  - non avrebbe senso esporre nel contratto elementi dell'implementazione

```
/**
 * Restituisce true sse la sequenza è vuota
 *
 * @pre true
 * @result <=> size() == 0
 * @post @nochange
 */
 public boolean isEmpty();
```

element()

- **Precondizioni specificano vincoli**
  - sul valore degli argomenti di chiamata
  - sullo stato dell'oggetto al momento della chiamata

```
/**
 * Restituisce l'elemento in posizione i
 *
 * @pre 0 <= i && i < size()
 * @post @nochange
 */
 public T element(int i);
```

head()

- **Precondizioni specificano vincoli**
  - sul valore degli argomenti di chiamata
  - sullo stato dell'oggetto al momento della chiamata

```
/**
 * Restituisce il primo elemento
 *
 * @pre !isEmpty()
 * @result == element(0)
 * @post @nochange
 */
 public T head();
```

insert()

- **Postcondizioni specificano vincoli**
  - sul valore calcolato dal metodo
  - sullo stato dell'oggetto al termine della chiamata

```
/**
 * Inserisce un nuovo elemento alla posizione i
 *
 * @pre item != null && i >= 0 && i < size()
 * @post size() = size()@pre + 1
 * @post @forall k : [0..size()-1] @
 *       (k < i ==> element(k)@pre == element(k)) &&
 *       (k == i ==> item == element(k)) &&
 *       (k > i ==> element(k-1)@pre == element(k)
 */
 public void insert(T item, int i);
```

insertHead()

```
/**
 * Inserisce un elemento sulla testa della sequenza
 *
 * @pre item != null
 * @post size() = size()@pre + 1
 * @post item == element(0))
 * @post @forall k : [1..size()-1]
 *                  @ element(k-1)@pre == element(k)
 */
 public void insertHead(T item);
```

remove()

```
/**
 * Rimuove l'elemento alla posizione i
 *
 * @pre size() > 0
 * @pre i >= 0 && i < size()
 * @result == element(i)@pre
 * @post size() = size()@pre - 1
 * @post @forall k : [0..size()-1] @
 *       (k < i ==> element(k)@pre == element(k)) &&
 *       (k >= i ==> element(k+1)@pre == element(k))
 */
 public T remove(int i);
```

removeHead()

```
/**
 * Rimuove l'elemento in testa e lo restituisce
 *
 * @pre size() > 0
 * @post @result == element(0)@pre
 * @post size() = size()@pre - 1
 * @post @forall k : [0..size()-1]
 *                  @ element(k+1)@pre == element(k)
 */
 public T removeHead();
```

Domanda

- Come completereste la specifica del metodo `deposit()` nella gerarchia `BankAccount`?

```
/**
 * Deposita un importo sul conto
 *
 * @pre
 * @post
 */

 public void deposit(double amount) { . . . }
```

Risposta

```
/**
 * Deposita un importo sul conto
 *
 * @pre amount > 0
 * @post getBalance() = getBalance()@pre + amount
 */

 public void deposit(double amount) { . . . }
```

Domanda

- **Vi sembra corretta la seguente specifica del metodo `withdraw()`?**

```
/**
 * Preleva un importo sul conto
 *
 * @pre amount > 0
 * @post balance = balance@pre - amount
 */

 public void withdraw(double amount) { . . . }
```

Risposta

- No, le pre/post condizioni devono essere espresse sempre in termini dell'interfaccia pubblica mentre `balance` è una variabile privata della classe
- la versione corretta è espressa utilizzando il metodo `getBalance()`così come visto per il metodo `deposit()`.

Contratti, sottotipi e *method overriding*

- **A proposito di ereditarietà, avevamo detto**
  - la ridefinizione di un metodo della superclasse nella sottoclasse deve rispettare il tipo del metodo nella superclasse
- **Compatibilità di tipi necessaria per la correttezza del principio di sostituibilità:**
  - istanze di sottotipo possono essere assegnate a variabili di un supertipo
- **Necessaria, non sufficiente**
  - il comportamento del metodo nella sottoclasse deve essere compatibile con il metodo della superclasse

Continua…

Contratti, sottotipi e *method overriding*

- **Design by contract**
  - ciascun metodo di una classe deve rispettare il contratto del corrispondente metodo nella sua superclasse e/o nelle interfacce implementate dalla classe

Continua…

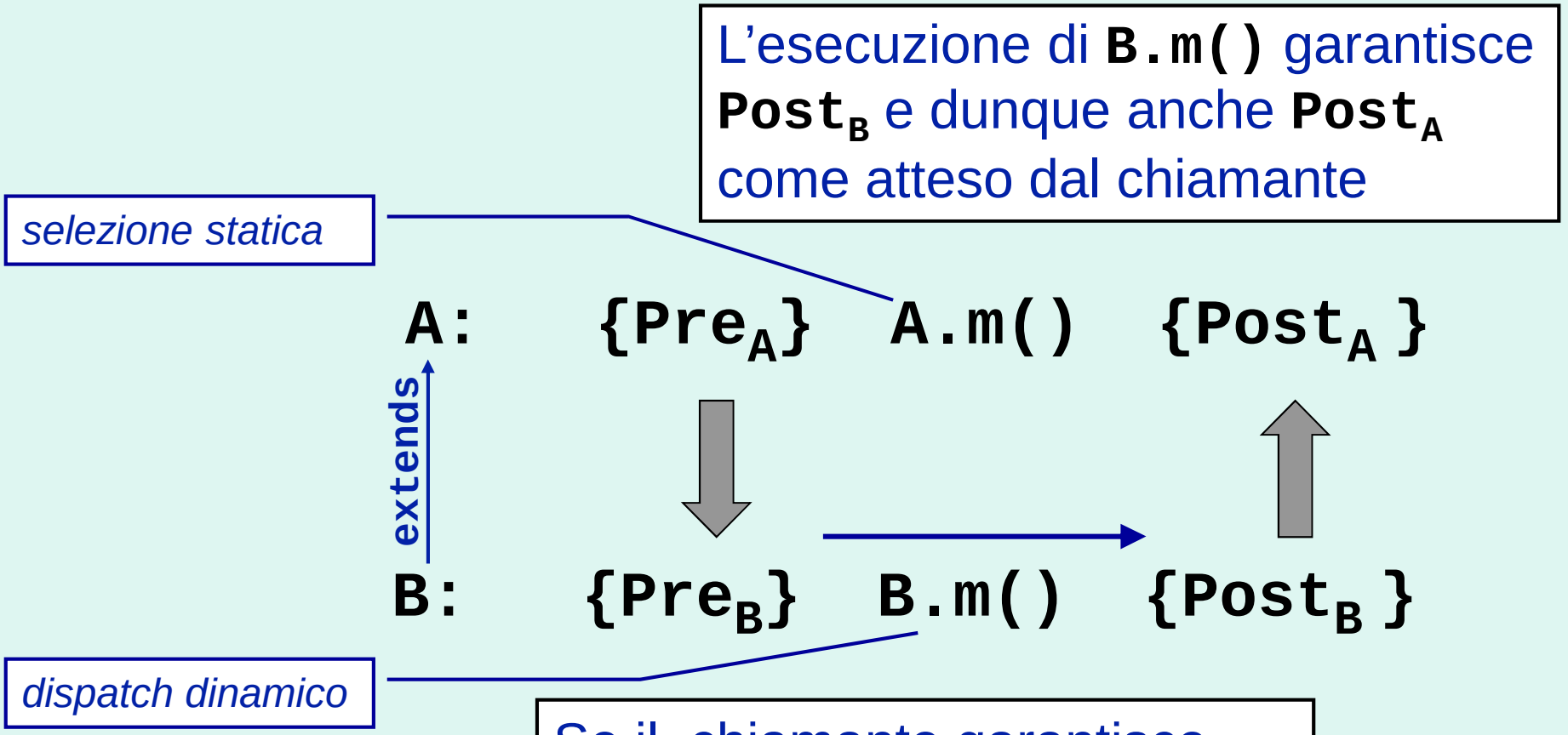
Contratti, sottotipi e *method overriding*

```
class B extends A { . . .   }
```

- **precondizione**
  - la precondizione di ciascun metodo di **B** deve essere implicata dalla precondizione del metodo corrispondente di **A**
- **postcondizione**
  - la postcondizione di ciascun metodo di **B** deve implicare la postcondizione del metodo corrispondente di **A**

Continua…

Contratti, sottotipi e *method overriding*



Continua…

Domanda

- **Delle due seguenti versioni di `withdraw()`, quale deve stare nella superclasse e quale nella sottoclasse secondo il principio di compatibilità?**

```
/**
 * @pre amount > 0
 * @post getBalance() = getBalance()@pre - amount
 */
 public void withdraw(double amount) { . . . }
```

```
/**
 * @pre amount > 0 && getBalance() >= amount
 * @post getBalance() = getBalance()@pre - amount
 */
 public void withdraw(double amount) { . . . }
```

Risposta

- **nella sottoclasse**

```
/**
 * @pre amount > 0
 * @post getBalance() = getBalance()@pre - amount
 */
 public void withdraw(double amount) { . . . }
```

- **nella superclasse**

```
/**
 * @pre amount > 0 && getBalance() >= amount
 * @post getBalance() = getBalance()@pre - amount
 */
 public void withdraw(double amount) { . . . }
```

Invarianti di classe

- **Lo stato di un oggetto si dice**
  - *transiente* durante l'esecuzione di un metodo invocato sull'oggetto
  - *stabile* se l'oggetto è stato inizializzato e nessuno dei metodi della classe è in esecuzione sull'oggetto stesso
- **Invarianti di classe**
  - condizione verificata su tutti gli oggetti della classe che si trovano in stato stabile

Continua…

Invariante di classe per `MySequence<T>`

- **Supponiamo che la rappresentazione della sequenza sia mediante una lista doppia con puntatori alla testa e coda della lista**

```
// classe interna
private static class Node<T> {
   T element;
   Node<T> next, prev;
}
// testa e coda della sequenza
private Node<T> head, tail;
// contatore degli elementi in sequenza
private int count;
```

Continua…

Invariante di classe per `MySequence<T>`

- **Vincoli di consistenza sulla rappresentazione della sequenza**
  - se la sequenza è vuota **`tail = head = null`**;
  - se la sequenza è non vuota, **`head`** punta al primo nodo, **`tail`** all'ultimo
  - **`count`** = numero di elementi in sequenza
  - per ogni nodo intermedio il **`next`** del precedente e il **`prev`** del seguente puntano al nodo stesso
  - il **`prev`** del primo nodo e il **`next`** dell'ultimo sono **`null`**

Continua…

Invariante di classe per `MySequence<T>`

```
protected boolean _wellFormed() {
      int n = 0;
      for (Node<T> p = head; p != null; p = p.next) {
            n++;
            if (p.prev != null) {
                  if (p.prev.next != p) return false;
            } else { // p è il primo
              if (head != p) return false;
            }
            if (p.next != null) {
                  if (p.next.prev != p) return false;
            } else { // p è l'ultimo
              if (tail != p) return false;
            }
       }
      if (n == 0 && tail != null) return false;
      return n == count;
}
```

Continua…

Invariante di classe per `MySequence<T>`

- **L'invariante può (deve) essere documentato nella implementazione della classe**
- **utilizziamo una tag specifica**

```
public class LinkedSequence<T> implements Sequence<T>
{
    /**
     * @invariant _wellFormed()
     */
    protected boolean _wellFormed() { . . . }
   . . .
}
```

Invarianti di classe

- **A differenza delle pre/post condizioni, l'invariante esprime vincoli sulla rappresentazione interna delle classi**
- **Utilizzato per giudicare la correttezza dell'implementazione**

Continua…

Invarianti di classe

- **Devono essere garantiti dai costruttori**
  - l'invariante deve valere dopo l'inizializzazione
  - tutti i costruttori pubblici di una classe devono avere l'invariante di classe come post condizione
- **Devono essere preservati dai metodi pubblici**
  - l'invariante può essere assunto come precondizione del corpo di ciascun metodo pubblico,
  - deve essere trattato come una postcondizione da soddisfare al termine dell'esecuzione

Invarianti e correttezza dei metodi

Specifica astratta data in termini del contratto di m() in A

$$\{Pre_A\} \quad A.m() \quad \{Post_A\}$$

$$\{Pre_A \text{ AND } Inv_A\} \; Body_{A.m()} \{Post_A \text{ AND } Inv_A\}$$

Specifica concreta, verificata nell'implementazione
***Inv_A** è l'invariante di classe*

Domanda

- La seguente implementazione di **BankAccount** è corretta rispetto a specifiche e invarianti?

```
class BankAccount {
   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre - import */
    public void withdraw(double import) { balance =- import;}

   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre + import */
    public void deposit(double import) { balance =+ import; }

   /** @pre true
    *  @post @nochange  */
    public double getBalance() { return balance; }

   /** saldo e fido associati al conto
    *  @invariant balance >= credit  */
    private double balance, credit;
 }
```

Risposta

- **NO:**
  - il costruttore default inizializza a zero entrambi i campi e quindi rende vero l'invariante
  - ma l'invariante non è implicato dalla postcondizione del metodo **`withdraw()`**
  - il problema è che la precondizione di **`withdraw()`** è troppo debole
- **Dobbiamo rinforzare la precondizione per `withdraw()`**
  - insieme ad altri piccoli aggiustamenti per chiudere il ragionamento

Continua…

Risposta

```
class BankAccount {
   /** @pre import > 0 && getBalance() – import >= getCredit()
    *  @post getBalance() = getBalance()@pre - import */
    public void withdraw(double import) { balance =- import;}

   /** @pre import > 0
    *  @post getBalance() = getBalance()@pre + import */
    public void deposit(double import) { balance =+ import; }

   /** @pre true
    *  @post @nochange  */
    public double getBalance() { return balance; }

   /** @pre true
    *  @post @nochange  */
    public double getCredit() { return credit; }

   /** saldo e fido associati al conto
    *  @invariant balance >= credit
         && balance == getBalance() && credit == getCredit() */
    private double balance, credit;
 }
```

Asserzioni

- **Una asserzione è una affermazione che permette di testare le assunzioni riguardo determinati punti del programma**
- **Ogni asserzione contiene una espressione booleana che si assume essere verificata**
- **La verifica delle asserzioni permette di effettuare dinamicamente controlli sulla correttezza del codice**

Continua…

Asserzioni: sintassi

assert *Expression*;

Esempio:

```
assert  i >= 0 && i < size();
```

Scopo:

Verificare se una condizione è soddisfatta. Se le asserzioni sono abilitate e la condizione è falsa lancia un errore di asserzione. Altrimenti non ha effetto.

Continua…

Asserzioni: sintassi

assert *Expression1 : Expression2*;

Esempio:
assert i >= 0 && i < size() : “indice fuori range”

Scopo:
Come nel caso precedente, ma utilizza la seconda espressione per documentare l’errore.
Se le asserzioni sono abilitate e *Expression1* è falsa valuta *Expression2* e passa il risultato insieme all’errore di asserzione.
Altrimenti non ha effetto.

Asserzioni *Howto's*

- **Compilazione**

  ```
  javac -source 1.4 <prog>.java
  ```

- **Esecuzione**
  - abilitazione/disabilitazione selettiva di eccezioni

    ```
    java –ea [:<package>|:<classe>] <prog>
         –da [:<package>|:<classe>] <prog>
    ```

  - abilitazione/disabilitazione di asserzioni di sistema

    ```
    java –esa
    java –dsa
    ```

Continua…

Asserzioni e *Unit Testing*

- **Le asserzioni sono molto efficaci per la verifica della corretta implementazione di una classe**
- **Derivate da:**
  - postcondizioni di metodi (pubblici e privati)
  - invarianti di classe
  - precondizioni di metodi privati

Asserzione di postcondizioni

```
/**
 * Restituisce il primo elemento della sequenza
 *
 * @pre !isEmpty()
 * @post @result == element(0)
 */
  public T head() {
       T result = (head != null ? head.item : null);
       assert result == element(0);
       return result;
 }
```

Asserzione di invarianti di classe

```
/**
 * Inserisce un nuovo elemento alla posizione i
 *
 * @pre . . .
 * @post size() = size()@pre + 1
 * @post . . .
 */
 public void insert(T item, int i) {
      assert _wellFormed();
      int size_pre = size();
       // ... codice di inserimento
      int size_post = size();
      assert size_post == size_pre + 1;
      assert _wellFormed();
}
```

Altri tipi di asserzioni

- **Per invarianti interni**
  - Dove tradizionalmente utilizzeremmo commenti …

```
if (i % 3 == 0) { . . . }
else if (i % 3 == 1) { . . . }
else { // a questo punto  i % 3 == 2
  . . .
}
```

  - … è più efficace utilizzare asserzioni

```
if (i % 3 == 0) { . . . }
else if (i % 3 == 1) { . . . }
else { assert i % 3 == 2
  . . .
}
```

Continua…

Altri tipi di asserzioni

- **Invarianti del flusso di controllo**
  - nei casi in cui vogliamo segnalare che un certo punto del programma non dovrebbe mai essere raggiunto;
  - possiamo asserire una costante sempre falsa

```
void m() {
  for (. . . ) {
     if (...)
     return;
  }
  assert false; // qui non ci dovremmo mai arrivare
}
```

Quando *non* usare asserzioni

- **Per verificare precondizioni di metodi pubblici**
  - essendo il metodo pubblico, non abbiamo controllo sul codice che invoca il metodo

```
/**
 * @pre !isEmpty()
 * @post @result == element(0)
 */
  public T head() {
       assert !isEmpty(); // brutta idea
       T result = (head != null ? head.item : null);
       assert result == element(0);
       return result;
 }
```

- **Meglio usare eccezioni in questi casi**

Continua…

Quando *non* usare asserzioni

- **Con espressioni che coinvolgono *side effects***
  - Esempio: vogliamo rimuovere tutti gli elementi **`null`** di una lista **`els`** e verificare che effettivamente la lista conteneva almeno un elemento **`null`**

```
// questa asserzione è scorretta _non_ usare
assert els.remove(null);
```

```
// questo è il modo giusto di costruire l'asserzione
boolean nullRemoved = els.remove(null);
assert nullRemoved;
```

Continua…

Defensive Programming

- **Pratica che mira ad evitare un utilizzo scorretto dei metodi di una classe**
- **Verifica delle precondizioni. Come?**
- **Dipende dalla situazione …**
- **Due situazioni:**
  - Abbiamo controllo sul chiamante (metodo privato)
    - asseriamo la precondizione
  - Evento fuori dal nostro controllo
    - Lanciamo una *eccezione*