



---

# PROGRAMMING BY CONTRACT

---

PR2 2018-19



## Cosa vogliamo ottenere

---

- Implementare il sistema “giusto”
  - *validation*
- Implementare il sistema correttamente
  - *verification*
- PR2 affronta il secondo aspetto: come creare del software che funziona correttamente
  - Compito difficile: specificare, progettare, implementare, testare e “debug” anche per semplici programmi

PR2 2018-19



## Un piccolo SE primer

---

- **Specifica:** Cosa si deve realizzare
- **Design:** Come si decompone il sistema in moduli in modo da ridurre la complessità. Quali sono le migliore strategie di progetto?
- **Implementazione:** scrivere il codice che soddisfa le specifiche
- **Testing:** Determinare sistematicamente eventuali errori
- **Debugging:** Risolvere in modo sistematici gli errori
- **Manutenzione:** Come far evolvere il sistema nel tempo
- **Documentazione:** Descrivere tutto le scelte fatte e le loro motivazioni.

PR2 2018-19



## La scalabilità del software

---

- I programmi di piccole dimensioni sono facilmente maneggiabili e modificabili
  - Coding (facile)
  - Modifiche (facile)
- Programmi di grosse dimensione sono maggiormente complessi da gestire sia in fase di coding che di manutenzione.
- Il motivo principale è dato dalla difficoltà nella gestione delle interazioni tra i diversi moduli che costituiscono il sistema

PR2 2018-19

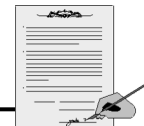


## Specifiche

---

- Il nostro approccio: due modi distinti di osservare il comportamento di un programma:
  - La visione di chi **implementa** (come costruire il sistema)
  - La visione di chi lo **usa**
- Motivazioni:
  - Quando si implementa un modulo software conviene anche avere la visione dei clienti che utilizzano quel modulo
  - La visione di chi implementa potrebbe essere troppo restrittiva
  - Permette di identificare quali sono i moduli che costituiscono il sistema e le loro dipendenze
- A PR2 questi aspetti sono analizzati mediante la nozione di *specifica*

PR2 2018-19



## Specifica = contratto di uso

---

- L'insieme dei **vincoli** che definiscono le mutue aspettative tra le parti
- **Strumento di decomposizione**
  - Permette di identificare i vincoli del cliente senza considerare il dettaglio di implementazione
  - Separa chiaramente la parte di implementazione dalle caratteristiche di uso.
  - Evita l'utilizzo di vincoli impliciti nascosti
- Facilita la manutenzione e le modifiche
  - Permette di ragionare in termini di vincoli di interazione non di strutture di implementazione

PR2 2018-19



## Soluzione limitata: Java Interface

In Java (ma anche negli altri linguaggi OO) una *interface* definisce il confine tra implementazione e cliente:

```
public interface List<E> {  
    public E get(int index);  
    public void set(int index, E elem);  
    public void add(E elem);  
    public void add(int index, E elem);  
    ...  
}
```

Interface = *syntax & types*

Ma non permette di osservare i comportamenti e il loro effetto

*Interface: informazione limitata verso I clienti*

PR2 2018-19



## Usiamo il codice?

```
Static <T> boolean sub(List<T> src, List<T> part) {  
    int part_index = 0;  
    for (T elt : src) {  
        if (elt.equals(part.get(part_index))) {  
            part_index++;  
            if (part_index == part.size()) {  
                return true;  
            }  
        } else {  
            part_index = 0;  
        }  
    }  
    return false;  
}
```

Siamo sicuri che sia chiaro?

PR2 2018-19



## Il codice è complicato!!!

---

- Un numero elevato di dettagli non utili al cliente
- Sforzo non banale di comprensione:
  - Per usare una API si deve leggere la sua implementazione in dettaglio? Meglio evitare
- “Client cares only about *what* the code does, not *how* it does it”

PR2 2018-19



## Commenti del codice

---

Commenti: descrizione del comportamento

```
// This method checks if "part" appears as a
// sub-sequence in "src"
static <T> boolean sub(List<T> src, List<T> part) {
    ...
}
```

Problema: informazione incompleta

- Cosa succede nel caso in cui **src - part** sono liste vuote?
- In quali casi il metodo restituisce come risultato il valore **true**?

PR2 2018-19



## Il nostro esempio

---

```
static <T> boolean sub(List<T> src, List<T> part) {
    int part_index = 0;
    for (T elt : src) {
        if (elt.equals(part.get(part_index))) {
            part_index++;
            if (part_index == part.size()) {
                return true;
            }
        } else {
            part_index = 0;
        }
    }
    return false;
}
```

PR2 2018-19



## Una descrizione più accurata

---

```
// controlla se "part" è una sottosequenza in "src"

// * src e part non possono essere null
// * se src è una lista vuota allora si deve
restituire true come risultato
```

**Match parziali:**  
list (1,2,1,3) deve essere identificata come  
sottolista di (1,2,1,2,1,3).

```
// Il metodo scorre la lista "src"
// per determinare l'occorrenza "part", e
// scorre nuovamente la lista in caso di ...
```

PR2 2018-19

## Non va bene: troppo complicato

---



- *Semplificare la descrizione!*
  - Una descrizione complessa porta a un pessimo design della soluzione.

- Alternativa:

```
// restituisce true iff esistono due sottosequenze
A, B tali che
// src = A : part : B
// dove ":" è l'operazione di concatenazione
static <T> boolean sub(List<T> src, List<T> part) {
```

- Rigore matematico elimina le ambiguità
- Stile dichiarativo: si descrive **cosa** deve fare non **come** lo fa!
  - **what** vs **how**

PR2 2018-19

## Spec: valore aggiunto #1

---



- Un modo disciplinato di scrittura delle specifiche ha una forte influenza sulla struttura del codice:
  - Guida alla scrittura di programmi strutturati in cui sono chiaramente identificati i punti di scelta del progetto
- Se la specifica è troppo complicata forse è il caso di ripensare alle scelte di progetto.

PR2 2018-19



## Javadoc

---

- Javadoc: strumento nativo di Java per la scrittura delle specifiche
  - Segnatura del metodi
  - Descrizione testuale del comportamento
  - `@param`: *description of what gets passed in*
  - `@return`: *description of what gets returned*
  - `@throws`: *exceptions that may occur*

PR2 2018-19



## Esempio: Javadoc `String.contains`

---

```
public boolean contains(CharSequence s)
```

```
Returns true if and only if this string contains  
the specified sequence of char values.
```

```
Parameters:
```

```
s- the sequence to search for
```

```
Returns:
```

```
true if this string contains s, false otherwise
```

```
Throws:
```

```
NullPointerException
```

```
Since:
```

```
1.5
```

PR2 2018-19





## Noi ... è simile ma ....

---

- *precondition*: vincolo sulle proprietà che devono valere prima dell'invocazione del metodo (se non valgono allora non esistono le condizioni per eseguire il codice del metodo)
  - **@requires**: descrive le condizioni di validità che il cliente deve garantire
- *postcondition*: vincolo sulle proprietà che devono valere al momento della terminazione del metodo (nel caso in cui la preconditione sia soddisfatta!!!)
  - **@modifies**: elenca gli oggetti che sono modificati durante l'esecuzione del metodo, gli oggetti non presenti nell'elenco non sono modificati.
  - **@throws**: elenca le possibili eccezioni (come Javadoc)
  - **@effects**: descrive le proprietà dello stato finale
  - **@return**: descrive i valori restituiti dal metodo (come Javadoc)

PR2 2018-19



## Descrizione alternativa equivalente

---

- *precondition*: vincolo sulle proprietà che devono valere prima dell'invocazione del metodo (se non valgono allora non esistono le condizioni per eseguire il codice del metodo)
  - **@requires**: descrive le condizioni di validità che il cliente deve garantire
- *postcondition*: vincolo sulle proprietà che devono valere al momento della terminazione del metodo (nel caso in cui la preconditione sia soddisfatta!!!)
  - **@effects**:
    - elenca gli oggetti che sono modificati durante l'esecuzione del metodo, gli oggetti non presenti nell'elenco non sono modificati.
    - elenca le possibili eccezioni (come Javadoc)
    - descrive le proprietà dello stato finale
    - descrive i valori restituiti dal metodo (come Javadoc)

PR2 2018-19



## Esempio

`static <T> int change(List<T> lst, T oldelt, T newelt)`  
requires lst, oldelt, enewelt sono non-null.  
oldelt occorre in lst.  
modifies lst  
effects rimpiazza la prima occorrenza di oldelt in lst con newelt & non effettua altre modifiche su lst  
returns la posizione in lst che conteneva il valore oldelt e ora contiene newelt

```
static <T> int change(List<T> lst,
                    T oldelt, T newelt) {
    int i = 0;
    for (T curr : lst) {
        if (curr == oldelt) {
            lst.set(newelt, i);
            return i;
        }
        i = i + 1;
    }
    return -1;
}
```

PR2 2018-19



## Esempio 2

`static List<Integer> zipSum(List<Integer> lst1, List<Integer> lst2)`  
requires lst1 lst2 sono non-null.  
lst1 e lst2 hanno la stessa dimensione.  
modifies none  
effects none  
returns una lista della stessa dimensione dove l'elemento di posizione i è la somma dei valori degli elementi in posizione i delle due liste lst1 e lst2

```
static List<Integer> zipSum(List<Integer> lst1
                          List<Integer> lst2) {
    List<Integer> res = new ArrayList<Integer>();
    for(int i = 0; i < lst1.size(); i++) {
        res.add(lst1.get(i) + lst2.get(i));
    }
    return res;
}
```

PR2 2018-19



## Esempio 3

---

```
static void listAdd(List<Integer> lst1, List<Integer> lst2)
```

```
  requires  lst1 & lst2 != null.  
            lst1.size() = lst2.size().  
  modifies  lst1  
  effects   lst.get(i) = old(lst1.get(i)) + lst2.get(i)  
  returns   none
```

---

```
static void listAdd(List<Integer> lst1,  
                   List<Integer> lst2) {  
  for(int i = 0; i < lst1.size(); i++) {  
    lst1.set(i, lst1.get(i) + lst2.get(i));  
  }  
}
```

PR2 2018-19



## Esempio 4 (da fare!)

---

```
static void uniquify(List<Integer> lst)
```

```
  requires  ???  
            ???  
  modifies  ???  
  effects   ???  
  returns   ???
```

---

```
static void uniquify(List<Integer> lst) {  
  for (int i=0; i < lst.size()-1; i++)  
    if (lst.get(i) == lst.get(i+1))  
      lst.remove(i);  
}
```

PR2 2018-19



## Sulla clausola *requires*

---

- Assumiamo che il cliente invochi un metodo e che la clausola *requires* non sia soddisfatta: allora il codice del metodo non è vincolato ad alcun comportamento
  - Potrebbe restituire valori corrotti
  - È preferibile una soluzione di tipo *fail fast*: sollevare un errore piuttosto che avere comportamenti non corretti
- Precondizioni sono un elemento comune di tutte le librerie
  - Descrivono i valori corretti in ingresso

PR2 2018-19



## Una visione logica (nel senso di LPP)

---

Sia  $M$  una implementazione e  $S$  una specifica

$M$  soddisfa  $S$  se e solo se

- Tutti i comportamenti di  $M$  sono permessi da  $S$
- “I comportamenti di  $M$  sono un sottoinsieme di quelli di  $S$ ”

Se  $M$  non soddisfa  $S$  conviene modificare il codice

PR2 2018-19



## Benefici delle specifiche #2

---

- **Astrazione sulle strutture di implementazione:** il cliente deve solo comprendere le modalità di interazione senza dover comprendere i dettagli del codice
  - Dal punto di vista del cliente il codice potrebbe anche non esistere!
- Metodologia: scrivere prima di tutto le specifiche permette di utilizzarle come guida all'implementazione. Inoltre, permette di comprendere le interazioni con i clienti e la decomposizioni in moduli del sistema , favorendo:
  - *Teamwork*
  - *Definizione delle batterie di test*

PR2 2018-19



## Comparere specifiche

---

- Comprendere se una specifica è più restrittiva di altre
- Una specifica debole permette una grossa libertà di implementazione
  - Una specifica  $S_1$  è più debole di  $S_2$  se per ogni implementazione  $M$ ,
    - $M$  soddisfa  $S_2 \Rightarrow M$  soddisfa  $S_1$
- Due specifiche potrebbero essere inconfrontabile: una implementazione potrebbe soddisfare entrambe

PR2 2018-19



## E' utile?

---

E' importante saper mettere in relazione l'implementazione con la specifica in modo da avere gli strumenti per rispondere alle domande

- L'implementazione rispetta la specifica?
- L'implementazione soddisfa tutti i vincoli di interazione?

E' importante essere in grado di comparare specifiche differenti

- Quale specifica e' maggiormente restrittiva?
- Una implementazione che soddisfa una specifica molto restrittiva può essere utilizzata anche per una specifica più liberale
  - **Substitutability principle**

PR2 2018-19



## Esempio

---

```
int find(int[] a, int value) {
    for (int i=0; i<a.length; i++) {
        if (a[i]==value)
            return i;
    }
    return -1;
}
```

- Specification A
  - requires: value occorre in a
  - returns: i tale che a[i] = value
- Specification B
  - requires: value occorre in a
  - returns: *il piu piccolo indice* i tale che a[i] = value

PR2 2018-19



## Esempio

---

```
int find(int[] a, int value) {
    for (int i=0; i<a.length; i++) {
        if (a[i]==value)
            return i;
    }
    return -1;
}
```

- Specification A
  - requires: value occorre in a
  - returns: i tale che a[i] = value
- Specification C
  - returns: i tale che a[i]=value, or -1 if value non occorre in a

PR2 2018-19



## Rafforzare una specifica

---

- Rafforzare una specifica:
  - Promettere molto di più
    - Effects
    - Returns
    - Minor numero di oggetti modificati
    - Minor numero di eccezioni
  - Chiedere di meno al cliente
    - Rilassare i vincoli in ingresso
- Indebolire una specifica:
  - (opposto a quello scritto sopra)

PR2 2018-19



## Cosa e' meglio, per noi?

---

- Stronger specs: non sono la panacea!
- Weaker specs: non implicano un codice semplice
- Specs Trade-off:
  - Utili per il cliente
  - Favorire l'implementazione
  - Favorire il riuso del codice e la modularità

PR2 2018-19



## Una visione formale

---

- Una specifica è una formula logica
  - $S1$  più restrittiva di  $S2$  se  $S1$  implies  $S2$
- A PLP avete visto come affrontare le definizioni logiche e dimostrare le implicazioni logiche.
- *A PR2 dovete utilizzare quello che avete imparato a PLP*

PR2 2018-19