

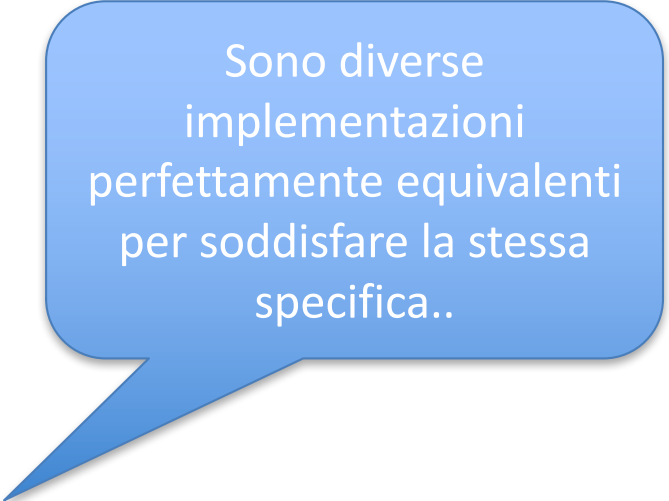
Astrazioni procedurali

Astrazione attraverso la specifica

- Non importa come il codice implementa un certo metodo
 - Basta che si comporti come ci si aspetta
- La specifica è la descrizione di “**cosa**” esegue il metodo, l’implementazione del “**come**”
 - La specifica di un metodo consente di ignorare l'algoritmo "incapsulato" nel metodo stesso

Esempio

- Specifica
 - /* valore assoluto di p */
- Parametrizzazione
`public static int abs(int p)`
- Implementazione
`if (p<0) return -p; return p;`
`oppure... return (p<0) ? -p : p;`
`oppure... return p *sgn(p);`
`oppure... return sqrt(p*p);`



Sono diverse
implementazioni
perfettamente equivalenti
per soddisfare la stessa
specifica..

Vantaggi della specifica

- **Località**
 - L'implementazione può essere letta o scritta senza la necessità di esaminare le implementazioni di altre astrazioni
 - Indipendenza dei programmatori: basta mettersi d'accordo sulla specifica
- **Modificabilità**
 - L'astrazione può essere re-implementata senza effetti sulle astrazioni che la usano ("non sorprendere gli utilizzatori")
 - Manutenzione semplice ed economica quando la specifica non cambia
- Quindi, la specifica deve essere definita con molta cura

Astrazione procedurale

- Definisce tramite una specifica un'operazione complessa su dati generici (o parametri)
- Può in generale avere diverse implementazioni, che ne rispettino la specifica

Specifica delle astrazioni procedurali

- Specifica data usando il linguaggio naturale o una semplice notazione matematica
 - La specifica deve essere chiara e precisa, in modo da non lasciare spazio ad ambiguità
- **requires** o preconditione (pre)
 - ...condizioni sui parametri sotto le quali la specifica è definita e valida
- **ensures** o postcondizione (post)
 - ...effetto garantito al termine dell'esecuzione dell'astrazione, sotto l'ipotesi che la preconditione sia verificata

Esempio Pre- e Post-condizioni

```
/*requires valore  $\geq 0$  del parametro x  
  *ensures restituisce la radice quadrata di x */  
public static float sqrt(float x)
```

- Cosa succede se $x < 0$?
 - Il comportamento del metodo sqrt non è definito
 - Qualunque comportamento è accettabile perché non deve mai succedere che la preconditione non è verificata

Pre- e Post-condizioni

- La **precondizione** di un metodo ci dice che cosa deve essere vero per poterlo chiamare
- La **postcondizione normale** ci dice che cosa deve essere vero quando il metodo ritorna normalmente (senza sollevare eccezioni)
- La **postcondizione eccezionale** ci dice che cosa è vero quando il metodo ritorna sollevando un'eccezione

Programmazione per contratto

- Termine usato spesso per descrivere astrazioni procedurali

**If you (caller) provide XYZ,
then I (method) promise to give you ABC**

- Pre-conditions (“requires”)
- Post-conditions (“ensures”)

Contratti per il software

```
/*@ requires x >= 0.0;  
  @ ensures (x - \result * \result) < eps;  
  @*/  
public static double sqrt(double x) { ... }
```

Il risultato al
quadrato dovrà
essere circa
uguale ad x...

	Obblighi	Diritti
Cliente	Passare parametro non negativo	Ottenere radice quadrata approssimata
Fornitore	Calcolare e ritornare la radice quadrata	Ricevere argomenti non negativi

Contratti e astrazioni

- Un contratto può essere soddisfatto in molti modi
 - Per esempio, square root:
 - Linear search
 - Binary search
 - Metodo di Newton
 - (diverse implementazioni per una stessa specifica)
- Ciò che cambia sono le proprietà non funzionali
 - Efficienza
 - Consumo di memoria
- Un contratto pertanto **astrae da tutte le implementazioni** che posso cambiare nel seguito

Contratti come documentazione

- Per ciascun metodo va definito
 - Che cosa richiede
 - Che cosa assicura
- I contratti sono
 - Più astratti del codice
 - Spesso verificabili meccanicamente

Ulteriori vantaggi

- **Attribuzione delle colpe**
 - Chi è il colpevole se:
 - ...la preconditione è violata?
 - ...la postcondizione è violata?
- **Evita inefficienti controlli difensivi**
 - ... delegati a chi deve soddisfare le pre-condizioni, cioè il chiamante

```
//@ requires a != null && (* a is sorted *);  
public static int binarySearch(Thing[] a, Thing x) { ... }
```

Regole di ragionamento

- Codice cliente
 - Deve funzionare per ogni implementazione che soddisfi il contratto
 - Può dunque solo far riferimento al contratto (e **non** al codice!)
 - ...deve assicurare la preconditione
 - ...deve prendersi carico della postcondizione
- Codice dell'implementazione
 - Deve soddisfare il contratto, e cioè
 - ...deve prendersi carico della preconditione
 - ...deve assicurare la postcondizione
 - Ma può fare qualunque cosa purchè permessa dal contratto

I commenti sono sufficienti per descrivere contratti?

- Se le specifiche sono scritte solo con commenti si pone il problema di come essere certi che le specifiche siano esatte e non ambigue, e che l'implementazione sia corretta
- Esempio:
- `*` se $x \geq 0$ restituisce la radice quadrata del parametro x `\`
- `public static float sqrt(float x)`
 - E' soddisfacente?
 - In realtà, la funzione calcola un valore approssimato della radice di x
 - Una specifica dovrebbe anche dire qual è l'approssimazione ammessa
- In generale un commento in linguaggio naturale può essere ambiguo o addirittura errato
- Inoltre, chi garantisce che l'implementazione rispetti la specifica?
 - ...commenti in linguaggio naturale non sono verificabili meccanicamente

Linguaggi per i contratti

- Esistono notazioni matematiche per scrivere le specifiche
 - Le specifiche non sono ambigue
 - Il codice e le specifiche possono essere verificate se la notazione è eseguibile

JML: Java Modeling Language

- Linguaggio per la descrizione formale delle specifiche, specializzato per Java
 - Useremo una versione semplificata e modificata
- Le specifiche JML sono contenute in annotazioni, rappresentati con la sintassi dei commenti:
 - `//@ ...`
 - oppure
 - `/*@ ... @*/`
 - I simboli “at” (@) all’inizio di una linea sono ignorati all’interno delle annotazioni

Asserzioni in JML

- Asserzioni JML sono espressioni booleane Java
 - Non possono avere side-effect
 - ...non si può usare =, ++, --, etc.
 - ...possono solo chiamare metodi senza side effect
 - Sono precedute da opportune keywords (requires, ensures, ...)
 - Possono usare alcune estensioni di Java come \result, \forall, ...

Sintassi	Semantica
$a \implies b$	a implica b
$a \Leftarrow b$	b implica a
$a \iff b$	a iff b
$a \Leftarrow \neq \Rightarrow b$	$\neg(a \iff b)$

Clausole requires e ensures

```
//@ requires in >= 0;
```

```
//@ ensures Math.abs(\result*\result-in)<0.0001;
```

```
public static float sqrt (float in);
```

- **requires** stabilisce la preconditione
- **ensures** la postcondizione
- **\result** nella clausola ensures indica il valore restituito al termine dell'esecuzione del metodo

Omettere requires o ensures

- Omettere pre- o post-condizione
 - Equivale ad assumere che siano true
- Se la clausola requires è omessa, il metodo non ha nessuna preconditione
- Se la clausola ensures è omessa, il metodo non ha nessuna postcondizione
 - ...quindi qualunque cosa faccia va bene
 - ...non è molto utile

Commenti

- Spesso è troppo complesso scrivere specifiche complete utilizzando JML
- Si possono inserire invece commenti
- I commenti possono essere strutturati o inseriti in formule JML: (* ... *) e valgono sempre true

//@ ensures (* a è una permutazione del valore originale di a*)

//@ && (* a è ordinato per valori crescenti *)

```
public static void sort (int[] a)
```

- Si cerca comunque di mantenere un struttura simil-JML

Descrizioni miste

- Si può spesso scrivere una postcondizione JML più debole di quella corretta, ma ancora significativa, piuttosto di scrivere solo un commento

```
public class IMath {  
    /*@ requires x>=0;  
        @ ensures \result >= 0 &&  
        @ (* \result is an int approximation to square root of x *)  
    @*/  
    public static int isqrt(int x) { ... }  
}
```

Assertzioni in Java

- Assertzione: espressione booleana da verificare a run-time
 - Se l'espressione è vera allora si continua
 - Se l'espressione è falsa il sistema genera un errore `AssertionError`, e il programma termina
- Abilitazione assertzioni
 - Compilazione: `javac -source 1.4`
 - Esecuzione: `java -ea`
- Sintassi: `assert <espressione booleana>;`

Descrizioni miste e asserzioni

- Se `a` è un array ed `x` è un elemento di `a`, allora `\result` è un indice di `a` in cui si trova `x`, altrimenti `\result == -1`

```
//@ ensures (*x e' in a*) && x==a[\result]  
//@ || (* x non e' in a *) && \result == -1;
```

- Così è possibile inserire dei check durante il test
`assert (i == -1 || a[i] == x);`
`return i;`
- Se l'implementazione restituisse un valore che non corrisponde all'indice a cui si trova `x` oppure a `-1`, si otterrebbe un `AssertionError`
- Però se l'implementazione restituisse sempre `-1`, non causerebbe errori!

Oggetti modificabili

- Per denotare lo stato delle variabili prima e dopo l'operazione, si usa `\old(espressione)`
 - Restituisce il valore che ha espressione al momento della chiamata
- Esempio: Metodo che inverte stato di accensione di un'Automobile

```
//@ ensures p.accesa() <==> !\old(p.accesa())
public static int invertiStato(Automobile p) {}
```
- A valle dell'esecuzione del metodo:
`p.accesa() == true` sse `\old(p.accesa) == false`
...e viceversa

Assignable

- Per segnalare che un parametro può essere modificato si usa assignable

```
//@ assignable a;  
//@ ensures (* a è una permutazione del valore originale di a*)  
//@ && (* a è ordinato per valori crescenti *)  
public static void sort (int[] a)
```

- Se un metodo non ha side-effect si può scrivere assignable \nothing.

```
//@ assignable \nothing;  
//@ ensures (*x e' in a*) && x==a[\result]  
//@ || (* x non e' in a *) && \result ==-1;  
public static int search (int[ ] a, int x)
```

Come progettare astrazioni procedurali?

- **Generalità:** le specifiche devono essere il più generali possibile
 - Esempio: usare parametrizzazione (una ricerca in array deve essere definita per array di qualunque lunghezza)
- **Minimalità:** imporre il minimo vincolo sul comportamento
 - La specifica deve lasciare la massima libertà implementativa
 - Se troppo dettagliata, vincola l'implementatore
 - Se troppo generica, ne limita l'utilità
 - in particolare **non** deve specificare **come** risolvere il problema
 - Le specifiche possono essere indeterminate o volutamente incomplete
 - Esempio: in una ricerca, quale indice viene ritornato nel caso di occorrenze multiple?
 - Non essendo rilevante non viene specificato...

Specificare eccezioni

```
//@ assignable \nothing;  
//@ ensures x == a[\result];  
//@ signals (NotFoundException e) (* x non e' presente in a *);  
public static int cerca(int x, int [] a) throws NotFoundException
```

- Significato: al termine si può essere in uno dei due casi
 - La postcondizione è vera e nessuna eccezione è lanciata
 - Viene lanciata un'eccezione ed è vera la condizione della signals corrispondente
- Quindi, al termine di cerca, può valere $x == a[\text{\result}]$; senza lanciare eccezione, oppure x non è presente in a e viene lanciata eccezione
- Occorre prevedere una clausola signal per ogni eccezione che il metodo può lanciare (sia checked che unchecked)

Semantica della \signals

//@ ensures A;

//@ signals (E e) B

...Equivale ad come avere postcondizione:

A && (*nessuna eccezione*)

|| B && (*lancia eccezione e*)

- Può essere lanciata eccezione se, oltre a B, vale anche la postcondizione A
 - È compito nostro distinguere i due casi, ad esempio dicendo che B nega A ...
- Il predicato B della signal può far riferimento sia all'oggetto eccezione e che all'interfaccia del metodo

Schema completo

```
visibility class c_name {  
    // commento generale sulla classe  
    //@ requires preconditione  
    //@ ensures postcondizione  
    //@ signals eccezioni  
    //@ assignable .....  
    visibility static ret_type p1(...) {
```

Operazioni parziali

- Molti metodi sono parziali, cioè hanno un comportamento specificato solo per un sottoinsieme del dominio degli argomenti
- Per esempio
 - `//@ requires n >= 0;`
 - `//@ ensures (* \result è il fattoriale di n *);`
 - `public static int fact (int n)`
- Cosa succede se i parametri non rispettano il vincolo?
 - La procedura per $n < 0$ calcola un valore scorretto che poi è usato da altre parti del programma
 - L'errore si propaga a tutto il programma, rovinando i risultati, i dati memorizzati, ecc.
 - Sarebbe meglio se la procedura “segnalasse” al chiamante il problema

Operazioni “parziali” e “robustezza”

- I metodi parziali compromettono la “robustezza” dei programmi
 - Un programma è “robusto” se, anche in presenza di errori o situazioni impreviste, ha un comportamento ragionevole (o, per lo meno, ben definito)
 - Per le procedure parziali il comportamento al di fuori delle precondizioni è semplicemente non definito dalla specifica
 - Se un metodo non è definito per alcuni valori (in quanto “inattesi”), si ottengono errori runtime o, peggio, comportamenti imprevedibili quando tali valori sono passati come parametri
- Per ottenere programmi robusti, le procedure devono essere “totali”!!!

Eccezioni e precondizioni

- In Java per convenzione la violazione di una precondizione di un metodo pubblico deve comportare il lancio di un'eccezione, e quindi avere funzioni totali

```
public static int fact (int n) throws NegativeException {
```

```
...
```

```
    if (n<0) throw new NegativeException();....
```

- È buona norma quindi eliminare la clausola `requires` e lanciare eccezioni quando la `requires` è violata
 - Le eccezioni si includono nella clausola `signals`
- Per i metodi `friendly`, `protected`, e `private`, la decisione se prevedere un'eccezione è lasciata al programmatore...

Requires ed eccezioni

```
//@ requires x != null;  
//@ ensures a[\result].equals(x);  
//@ signals (NotFoundException e) (*x non e' in a *);  
public static int cerca(String x, String[] a) throws NotFoundException
```

- Versione “alternativa”: lancia eccezione anche se x è null

```
//@ ensures x != null && a[\result].equals(x);  
//@ signals (NotFoundException e) (*x non e' in a *)  
//@ signals (NullPointerException e) x == null;  
public static int cerca(String x, String[] a) throws NullPointerException,  
NotFoundException
```

- Se in ensures mancasse `x != null`, allora per esempio se `x == null` e `a[\result] == null` la postcondizione sarebbe vera... la funzione potrebbe quindi non lanciare eccezione e terminare regolarmente!

Quando non lanciare eccezioni?

- Esempio con ricerca binaria

```
//@ requires (\forall int i; 0 <= i && i < a.length-1; a[i] < a[i+1]);  
//@ ensures a[\result] == x;  
//@ signals (NotFoundException e)  
//@      (\forall int i; 0 <= i && i < a.length; x != a[i]);
```
- La verifica della condizione requires richiede più tempo che l'esecuzione della ricerca!
- In genere, non si lancia un'eccezione quando il chiamante può fare il controllo della preconditione molto meglio del chiamato o quando il controllo è molto difficile/inefficiente
 - Questo indebolisce la sicurezza del programma, ma può essere un utile **compromesso** con l'efficienza

Elementi delle collezioni

- Per parlare degli elementi di una collezione, si possono usare i metodi pubblici che non hanno side effect
- equals, contains, containsAll, get, sublist,...

//@ ensures (* a è una permutazione di \old(a)*)

//@ && (* a ordinato per valori crescenti*);

public static void sort (ArrayList<Integer> a)

...equivale a:

//@ ensures a.containsAll(\old(a)) && \old(a).containsAll(a)

//@ && (* a ordinato per valori crescenti*);

public static void sort (ArrayList<Integer> a)

- Usiamo convenzione (non JML): se x è di tipo riferimento, \old(x) è un riferimento all'oggetto nello stato al momento della chiamata del metodo

Quantificatori

- I metodi `contains()` e simili delle collezioni spesso non bastano
 - Ad esempio: come scrivere “a ordinato per valori crescenti”?
- JML supporta diversi quantificatori
 - Universale and esistenziale (`\forall` and `\exists`)
 - Funzioni quantificatrici (`\sum`, `\product`, `\min`, `\max`)
 - Quantificatore numerico (`\num_of`)

`(\forall Student s; juniors.contains(s); s.getAdvisor() != null)`

\forall

(\forall variable; range; condizione)

- ...per tutti i possibili valori della variabile che soddisfano range, condizione deve essere true
- Esempio: a ordinato per valori crescenti
(\forall int i; 0 <= i && i < a.length-1; a[i] <= a[i+1])
- Equivalente a
(\forall int i; ; 0 <= i && i < a.length-1 ==> a[i] <= a[i+1])

\exists

//@ ensures

//@ (\exists int i; 0<=i && i<a.length; a[i] == x)

//@ ? x == a[\result]

//@ : \result == -1;

//@ assignable \nothing;

public static int cerca(int x, int [] a)

- Usa “operatore logico” di Java (? :) che funziona come un “if then else “

num_of

(\num_of int i; P(i); Q(i))

- Il numero totale (cardinalità) di i per cui vale P(i) && Q(i)
- Esempio: numero di elementi positivi in array a
(\numof int i; 0<=i && i<a.length; a[i]>0)
- Esempio: nessun elemento di a compare più di due volte in a
(\forall int i; 0<=i && i<a.length;
(\numof int j; i<j && j<a.length; a[i]==a[j]) <=1));

Sommatorie, produttorie...

$(\text{\textbackslash sum int } i; 0 \leq i \ \&\& \ i < 5; i) == 0 + 1 + 2 + 3 + 4$

$(\text{\textbackslash product int } i; 0 < i \ \&\& \ i < 5; i) == 1 * 2 * 3 * 4$

$(\text{\textbackslash max int } i; 0 \leq i \ \&\& \ i < 5; i) == 4$

$(\text{\textbackslash min int } i; 0 \leq i \ \&\& \ i < 5; i-1) == -1$

Serve tanto esercizio!