Pre- e Post- condizioni
Il ciclo while
Invarianti

Programmazione – Canale M-Z

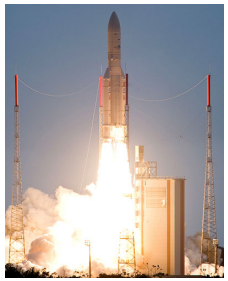
LT in Informatica
13 Dicembre 2016

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

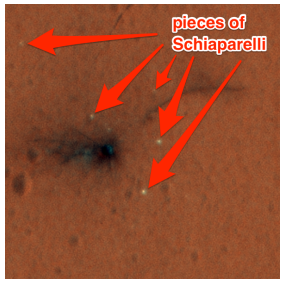
Gli errori di programmazione . . .

. . . possono costare molto cari:

Therac 25
(1985-87, sei incidenti gravi o mortali)



Esplosione dell'Ariane 5
(1996, 500 milioni $)



Schianto del Lander Schiaparelli ? (2016)

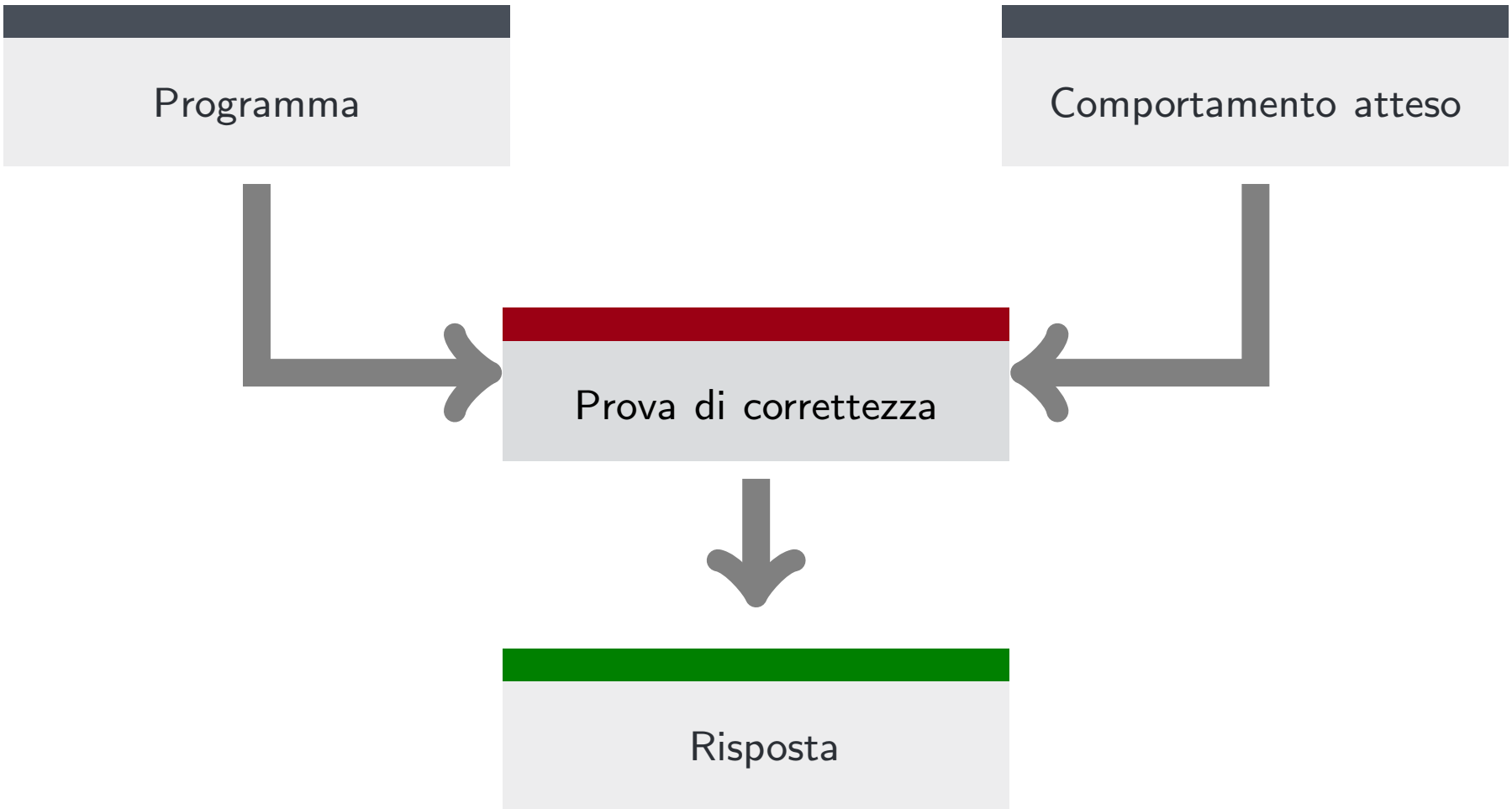
I limiti del testing

- Un modo per trovare errori nei programmi è fare dei test
- I test coprono solo un sottoinsieme dei possibili input
- Riescono a dimostrare l'esistenza di errori, ma non la loro assenza
- Un programma non opera in isolamento: il suo comportamento dipende da quello che succede nell'ambiente esterno
  - gli errori causati dall'interazione con l'ambiente o gli utenti sono molto difficili da individuare!

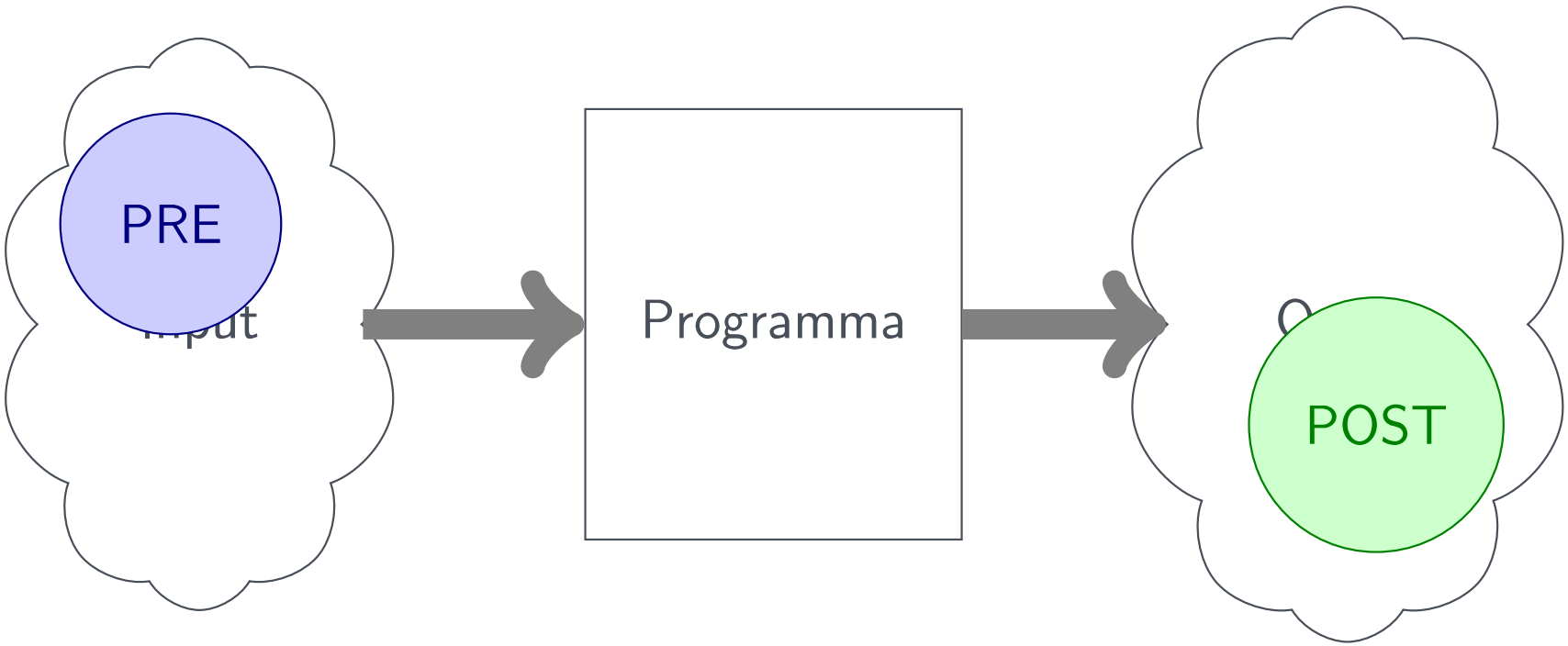
Un modo diverso per trovare gli errori

- Metodi Formali:
  - usare un linguaggio matematico ...
  - ... per descrivere il comportamento corretto ...
  - ... e aiutare il programmatore a scrivere programmi
- Permettono di stabilire la correttezza del programma per tutti i possibili input
- Sono pratica comune nello sviluppo di HW e SW:
  - progettazione di microprocessori (Intel), software critico (NASA), sistemi operativi (Microsoft), ...

Schema di un metodo formale



Pre- e Post- condizioni



- PRE-condizione: descrive gli input ammessi
- POST-condizione: descrive gli output attesi

Esempio: il valore assoluto


```
// PRE = (cin contiene il valore intero A)
int main() {
    int x, abs;
    cin >> x;
    if(x > 0) {
        abs = x;
    } else {
        abs = -x;
    }
}
// POST = (x = A, inoltre, se A > 0 allora abs =
    A, altrimenti abs = -A)
```

La POST-condizione dice che abs è uguale al valore assoluto di A

Esercizio 2: la morra cinese

Scrivere un programma che consenta di giocare alla morra cinese. Il programma deve ricevere in ingresso le mosse dei due giocatori, codificate con due `char`:

- se il carattere è `'f'` la mossa del giocatore è forbice
- se il carattere è `'c'` la mossa del giocatore è carta
- se il carattere è `'s'` la mossa del giocatore è sasso

Se viene inserita una mossa non valida il programma deve scrivere sullo schermo `mossa non valida` e terminare l'esecuzione. Se entrambe le mosse sono valide, il programma deve dichiarare il vincitore scrivendo sullo schermo `vince il giocatore 1` oppure `vince il giocatore 2`, oppure `pareggio`.

Proviamo a scrivere la PRE-condizione e la POST-condizione

Esercizio 2: PRE e POST

- `PRE=(cin contiene due caratteri qualsiasi C1 e C2)`
- `POST=(se C1 e C2 sono 's','c', o 'f' allora il programma stampa quale giocatore vince oppure se pareggiano, altrimenti stampa mossa non valida)`
- la `PRE` consente qualsiasi carattere come input
  - il programma deve controllare che C1 e C2 siano mosse valide

Esercizio 2: cambiamo un po' il testo

Scrivere un programma che consenta di giocare alla morra cinese. Il programma deve ricevere in ingresso le mosse dei due giocatori, codificate con due `char`:

- se il carattere è `'f'` la mossa del giocatore è forbice
- se il carattere è `'c'` la mossa del giocatore è carta
- se il carattere è `'s'` la mossa del giocatore è sasso

~~Se viene inserita una mossa non valida il programma deve scrivere sullo schermo `mossa non valida` e terminare l'esecuzione. Se entrambe le mosse sono valide,~~ il programma deve dichiarare il vincitore scrivendo sullo schermo `vince il giocatore 1` oppure `vince il giocatore 2`, oppure `pareggio`.

Proviamo a scrivere la PRE-condizione e la POST-condizione

Esercizio 2: PRE e POST

- `PRE = (cin contiene due caratteri C1 e C2 che possono avere valore 's','c', o 'f' )`
- `POST = (il programma stampa quale giocatore vince oppure se pareggiano)`
- la `PRE` restringe l'input alle sole mosse valide
  - il programma NON deve controllare l'input
  - può dichiarare il vincitore sapendo che le mosse saranno sempre valide

Esercizio 4: i lati del triangolo

Scrivere un programma che controlli se tre stuzzicadenti di lunghezze diverse possono essere disposti in modo da formare un triangolo oppure no.

. . . . . .

Il programma deve chiedere all'utente le lunghezze dei tre stuzzicadenti, controllare che siano tutte e tre positive e, se non lo sono stampare `"Input sbagliato"`, mentre se lo sono deve stampare `"Si"` se si possono disporre a triangolo e `"No"` altrimenti.

Due possibili PRE e POST

1. La PRE-condizione permette qualsiasi valore di input:

   ```
   PRE = (cin contiene tre valori interi A, B e C)

   POST = (se A, B, e C sono positivi, il prog.
   stampa SI se esiste un triangolo con lati di
   lunghezza pari ad A, B, e C, altrimenti stampa NO.
    Se A, B, e C non sono tutti positivi, allora
   stampa Input sbagliato)
   ```

2. La PRE-condizione permette solo valori positivi:

   ```
   PRE = (cin contiene tre valori interi POSITIVI A,
   B e C)

   POST = (il prog. stampa SI se esiste un triangolo
   con lati di lunghezza pari ad A, B, e C,
   altrimenti stampa NO.)
   ```

Il ciclo `while`

Il ciclo `while` permette di ripetere più volte operazioni simili:

```
int n = 1;
while(n < 11) {
    cout << n << endl;
    n = n + 1;
}
cout << "Ho contato da 1 a 10" << endl;
```

Il flusso di esecuzione del ciclo è:

1. determina se la condizione (`n < 11`) è vera o falsa
2. se è vera si eseguono le istruzioni del corpo del ciclo e poi si ritorna al punto 1
3. se è falsa, si esce dal ciclo

Cicli e terminazione

- il corpo del `while` deve cambiare il valore di una o più variabili ad ogni iterazione
- in modo tale che la condizione prima o poi diventi falsa
- altrimenti il ciclo si ripete all'infinito

```
int n = 1
while(n < 11) {
    cout << n << endl;
}
cout << "Questa istruzione non viene mai eseguita" <<
    endl;
```

Ctrl-C interrompe l'esecuzione del programma

Contatori ed accumulatori

Contatori

La variabile `n` usata per contare da `1` a `10` è un contatore:

- viene inizializzata prima dell'esecuzione del ciclo
- ad ogni iterazione il suo valore aumenta o diminuisce di un valore fisso

Accumulatori

Una variabile può essere un accumulatore:

- utilizzata ad esempio per calcolare totali e somme
- viene inizializzata prima dell'esecuzione del ciclo
- ad ogni iterazione il nuovo valore non sostituisce quello vecchio, ma si accumula a quelli già presenti in precedenza

Esempio: le potenze di due

Esercizio

Dato un numero $n > 0$, calcolare 2^n senza usare la funzione pow.

```
// PRE=(cin contiene un intero n > 0)
int main() {
    int n, k=1, pot=2;
    cin >> n;
    while(k < n)
    {
        k=k+1;
        pot=pot*2;
    }
    cout << pot << endl;
}
// POST = (pot = 2^n)
```

Correttezza del `while`: gli invarianti

- Per dimostrare che un ciclo `while` è corretto dobbiamo definire un invariante per il ciclo
- Cioè una condizione che è vera per ogni passo di computazione

L'invariante deve rispettare tre condizioni:

1. **Condizione iniziale:** dev'essere vero subito prima di entrare nel ciclo
2. **Invarianza:** dev'essere alla fine di ogni iterazione del ciclo
3. **Condizione di uscita:** assieme alla negazione della guardia del `while` deve implicare la POST-condizione

Un invariante per l'esempio delle potenze

Proviamo con la condizione `R = (pot = 2^k)`

1. Rispetta la condizione iniziale?
   - ✓ Si: all'inizio `k = 1` e `pot = 2 = 2^1`
2. Rispetta l'invarianza?
   - ✓ Si: dopo ogni iterazione `k` aumenta di uno e `pot` viene moltiplicato per 2: `2*pot = 2*2^k = 2^(k+1)`
3. Rispetta la condizione di uscita?
   - ✗ NO: se `k >= n` e `pot = 2^k` allora non è vero che `pot = 2^n` (`k` potrebbe essere più grande di `n`)

L'invariante completo

Ci serve una condizione in più: `R = (pot = 2^k && k <= n)`

1. Rispetta la condizione iniziale?
   - ✓ Si: la `PRE` ci dice che `n > 0` e quindi all'inizio `k = 1 <= n` e `pot = 2 = 2^1`
2. Rispetta l'invarianza?
   - ✓ Si: all'inizio dell'iterazione `k < n` e poi aumenta di uno, mentre `pot` viene moltiplicato per 2: `2*pot = 2*2^k = 2^(k+1)` e `k+1 <= n`
3. Rispetta la condizione di uscita?
   - ✓ Si: all'uscita dal ciclo `k >= n` (negazione della guardia) e `k <= n` (dall'invariante), quindi `k == n`. Poiché `pot = 2^k = 2^n` abbiamo dimostrato la `POST = (pot = 2^n)`

Esempio: minimo numero di bit

Esercizio

Dato un intero $x > 0$, trovare il minimo intero n tale che $2^n \geq x$.

```
// PRE=(cin contiene un intero x > 0)
int main() {
    int x, n=0, pot=1;
    cin >> x;
    while(pot < x) // R = (pot = 2^n && 2^(n-1) < x)
    {
        n=n+1;
        pot=pot*2;
    }
    cout << n << endl;
}
// POST=(2^(n-1) < x <= 2^n)
```

Prova di correttezza

Invariante: `R = (pot = 2^n && 2^(n-1) < x)`

1. Rispetta la condizione iniziale?
   - ✓ Si: la `PRE` ci dice che `x > 0` e quindi all'inizio `n = 0`, `pot = 1 = 2^0` e `2^-1 < x`
2. Rispetta l'invarianza?
   - ✓ Si: all'inizio dell'iterazione `pot = 2^n < x`, poi `n` aumenta di uno e `pot` viene moltiplicato per 2: `2*pot = 2*2^n = 2^(n+1)` e `2^n < x`
3. Rispetta la condizione di uscita?
   - ✓ Si: all'uscita dal ciclo `pot >= x` (negazione della guardia). Poiché `pot = 2^n` e `2^(n-1) < x` (dall'invariante), abbiamo dimostrato che `2^(n-1) < x <= 2^n`

L'algoritmo di Euclide

- Serve per calcolare il Massimo Comun Divisore di due interi postivi a e b
- E' il più antico esempio di programma conosciuto: descritto per la prima volta da Euclide attorno al 300 a.C.
- Procede per sottrazioni successive:
  - Ad ogni passo, si sottrae il numero più piccolo da quello più grande
  - Il valore più grande viene scartato
  - Quando $a = b$ ho trovato $MCD(a, b)$
- Euclide, negli elementi, dimostra che il metodo è corretto

La proprietà fondamentale di $MCD(a, b)$

- Il metodo di Euclide si basa su una proprietà fondamentale del Massimo Comun Divisore:

se $a > b$ allora $MCD(a, b) = MCD(a - b, b)$

- Dimostriamolo!
  - Chiamiamo $m = MCD(a, b)$
  - m è un divisore sia di a che di b
  - possiamo scrivere $a = um$ e $b = vm$, per qualche $u > v > 0$
  - quindi $a - b = um - vm = (u - v)m$
  - allora m è anche un divisore di $a - b$
  - siamo sicuri che è anche $MCD(a - b, b)$?

La proprietà fondamentale di $MCD(a, b)$

- Il metodo di Euclide si basa su una proprietà fondamentale del Massimo Comun Divisore:

se $a > b$ allora $MCD(a, b) = MCD(a - b, b)$

- Dimostriamolo! (continua dalla slide precedente)
  - allora m è anche un divisore di $a - b$
  - siamo sicuri che è anche $MCD(a - b, b)$?
  - ipotizziamo per assurdo che esista un divisore comune di $a - b$ e b più grande di m
  - $a - b = xg$ e $b = yg$ per qualche divisore $g > m$ e due interi $x, y > 0$
  - ricaviamo nuovamente a da $a - b + b = xg + yg = (x + y)g$
  - quindi g è un divisore comune di a e b più grande di m
  - ASSURDO, perché $m = MCD(a, b)$

L'Algoritmo di Euclide in C++


```
// PRE=(cin contiene a e b interi positivi)
int main()
{
    int x, y;
    cin >> x >> y;
    while (x != y) // R = (MCD(x,y)=MCD(a,b))
    {
        if(x > y) {
            x=x-y;
        } else {
            y=y-x;
        }
    }
    cout << "MCD=" << x << endl;
}
// POST=(stampa MCD(a,b))
```

Correttezza dell'Algoritmo di Euclide

Invariante: `R = (MCD(x,y)=MCD(a,b))`

1. Rispetta la condizione iniziale?
   - ✓ Si: all'inizio `x = a` e `y = b`
2. Rispetta l'invarianza?
   - ✓ Si: per la proprietà fondamentale di MCD, se `x > y` allora `MCD(x - y, y) = MCD(x,y) = MCD(a,b)`, mentre se `y > x` allora `MCD(y - x, x) = MCD(x,y) = MCD(a,b)`
3. Rispetta la condizione di uscita?
   - ✓ Si: all'uscita dal ciclo `x = y` (negazione della guardia), quindi `MCD(x,y) = MCD(x,x) = x`. Per l'invariante `MCD(x,y) = MCD(a,b) = x`. Il programma stampa `x`, come richiesto da POST.

Esercizio per casa

Scrivere un programma che legge un valore intero `n > 0` e una sequenza di `n` coppie di caratteri che rappresentano mosse della morra cinese. Il programma deve dichiarare quale giocatore ha vinto alla fine della sequenza di `n` giocate, scrivendo sullo schermo `"Vince il Giocatore 1"`, oppure `"Vince il Giocatore 2"`, oppure `"Pareggio"`.

- Da fare e sottomettere sul Moodle, come gli esercizi di laboratorio