TEST STRUTTURALE

Test strutturale

- White box, glass box
  - Scelta dei dati di test basata sulla struttura del codice testato
- È complementare al testing funzionale, ed è il solo modo per avere la certezza di sollecitare tutte le parti del codice
- Si cerca di trovare dati di test che consentano di percorrere “tutto il programma”
  - Per trovare un errore nel codice bisogna usare dei dati che “percorrono” la parte erronea
- Il concetto di percorrenza corrisponde al concetto di **cammino**
  - Sequenza di istruzioni attraversata durante un’esecuzione

Esempio Testing strutturale

```
static int maxOfThree (int x, int y, int z) {
1.      if (x > y)
2.              if (x > z)
3.                      return x;
4.              else return z;
5.      if (y > z)
6.              return y;
7.      else return z; }
```

- Se gli ingressi variano su intervallo di *n* elementi, ci sono n^3 possibili combinazioni; ma i **cammini** possibili sono solo 4:

  1 2 3; 1 2 4; 1 5 6; 1 5 7

- Servono dati di test per completarli tutti:
- Ad esempio, per 1 2 3 servono x>y && x>z (cioè ad esempio (9,8,6))

Copertura dei cammini

- Si deve scegliere un insieme di dati di test che consente di percorrere ("esercitare") tutti i cammini attraverso il programma; se si riesce si è raggiunta la **copertura totale dei cammini**

- **Copertura totale dei cammini** per l'esempio ottenuta con dati di test partizionati in 4 classi; per ognuna si sceglie un dato di test rappresentativo
  - (3, 2, 1) $\in$ {(x, y, z) | x > y > z} (cammino 1 2 3)
  - (3, 2, 4) $\in$ {(x, y, z) | x > y && x <= z} (cammino 1 2 4)
  - (1, 2, 1) $\in$ {(x, y, z) | x <= y && y > z} (cammino 1 5 6)
  - (1, 2, 3) $\in$ {(x, y, z) | x < y && y <= z} (cammino 1 5 7)
- Nota: esistono strumenti di supporto per queste attività…

Problemi copertura dei cammini

- Copertura dei cammini può non essere sufficiente a trovare gli errori

```
static int maxOfThree (int x, int y, int z) {
  if (x > y) return x; else return y;
}
```

  - Test contenente solo i dati (2, 1, 1) e (1, 3, 2) copre tutti i cammini ma non trova l'errore!
  - Il motivo è che nel programma "manca" un cammino che tratta la variabile z
  - Errore viene trovato facilmente con test funzionale
- In generale, test strutturale non può scoprire assenza di cammini, ma solo (eventualmente) trovare errori nei cammini esistenti

  $\Rightarrow$ Test strutturale va **sempre** complementato con quello funzionale

Test Strutturale: cicli

- Copertura totale dei cammini è impossibile da raggiungere, in pratica
  - Un cammino è infatti un percorso che può ripassare più volte su stessa istruzione durante un ciclo
- Esempio:

```
j = k;
for (int i = 1;  i <= 100;  i++  )
    if (Tests.pred(i*j)) j++;
```

- Predicato **`pred`** può essere vero o falso, indipendentemente, per qualsiasi valore i*j, con 1<=i<=100;
  - Quindi, per ogni cammino che porta alla i-sima iterazione, ci sono 2 cammini che portano alla (i+1)-sima iterazione; in tutto, 2^{100} possibili cammini

Copertura con i cicli

- Copertura totale impossibile, ci si accontenta di "approssimazione": si preparano dati per poche iterazioni (p.es. 2)

```
j = k;
for (int i = 1;  i <= 100;  i++  )
   if (Tests.pred(i*j)) j++;
```

- Trasformato in:

```
j = k;
for (int i = 1;  i <= 2;  i++  )
   if (Tests.pred(i*j)) j++;
```

- Ora bastano dati per i 4 casi
  1. pred(k) && pred(2k+2)
  2. pred(k) && ! pred(2k+2)
  3. ! pred(k) && pred(2k)
  4. ! pred(k) && ! pred(2k)

Copertura strutturale

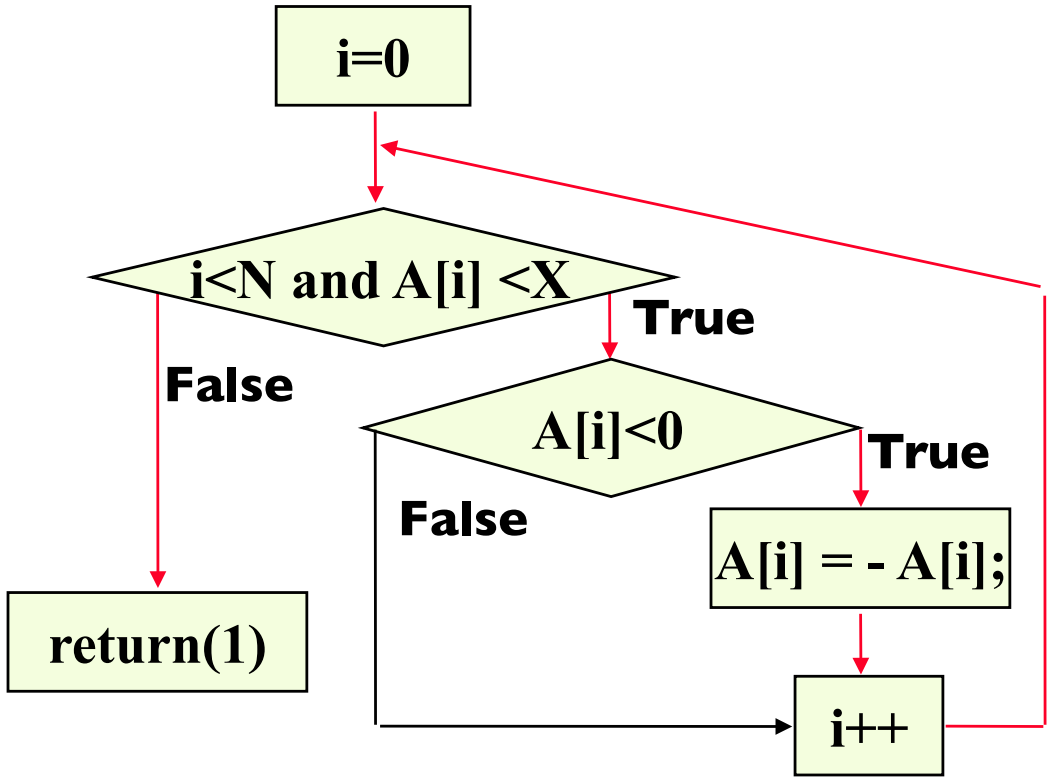
- Criterio di (in)adequatezza
  - Se parti significative della struttura del programma non sono coperte, il testing è inadeguato
- Criteri glass box = copertura strutturale del flusso di controllo
  - Copertura dei cammini (**path coverage**)
  - Copertura delle istruzioni (**statement coverage**)
  - Copertura delle diramazioni (**edge coverage**)
  - Copertura delle condizioni (**condition coverage**)

Copertura delle istruzioni

- Selezionare un insieme T di dati di test tali per cui ogni istruzione viene eseguita almeno una volta da qualche dato di T
  - Fissato il criterio, si cerca di trovare il T di cardinalità minima che soddisfa il criterio
- Razionale
  - Se certe istruzioni non sono mai state eseguite, si sospetta che possano essere causa di errore
  - Comunque, la copertura di tutte le istruzioni senza che insorgano malfunzionamenti NON assicura l'assenza di errori

Esempio

```
int select(int A[], int
N, int X)
{
  int i=0;
  while (i<N && A[i] <X)
  {
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



Un caso (N=1, A[0]=-7, X=9) è sufficiente a garantire il criterio
Eventuali errori nel gestire valori positivi di A[i] non verrebbero rilevati

Copertura strutturale

- Criterio di (in)adequatezza
  - Se parti significative della struttura del programma non sono coperte, il testing è inadeguato
- Criterio di Copertura delle Istruzioni (statement coverage)

$$\frac{\textbf{Numero-Istruzioni-Coperte}}{\textbf{Numero-Totale-Istruzioni}} \times 100$$

Come cercare di “coprire” un’istruzione?

- Data un’istruzione non coperta, come faccio a cercare di coprirla?
  - Esamino un cammino che porti ad essa
  - Calcolo la condizione sui dati associata a quel cammino (**path condition**)
  - Cerco di sintetizzare dati che rendono vera la condizione
  - Se non ci riesco, provo con un altro cammino...

41

Esempio

```
1. get(x); get(y)
2. while (x!=y) do {
3.          if (x>y) then
4.                x=x-y;
else
5.               y=y-x;
    }
6. put(x);
```

In generale

- Dato un cammino ed eventualmente una iniziale pre-condizione, si può calcolare la path condition associata a un path
- Questa in generale è una formula del calcolo dei predicati del primo ordine
- Trovare dei valori che la rendano soddisfacibile non può essere fatto in generale in maniera algoritmica (SAT indecidibile)
- Molti dei problemi teorici connessi al testing risultano indecidibili!
  - Sono necessarie euristiche!

Coperture non fattibili

- 100% di copertura potrebbe NON essere raggiungibile
  - Codice irraggiungibile (morto), cammini non fattibili, programmazione difensiva
- Ci si accontenta di coperture tipo “90% delle istruzioni” (magari ispezionando manualmente le parti non coperte)

```
if (x>0) {
        if (x=0) {
                . . .
        }
        . . .
}
```

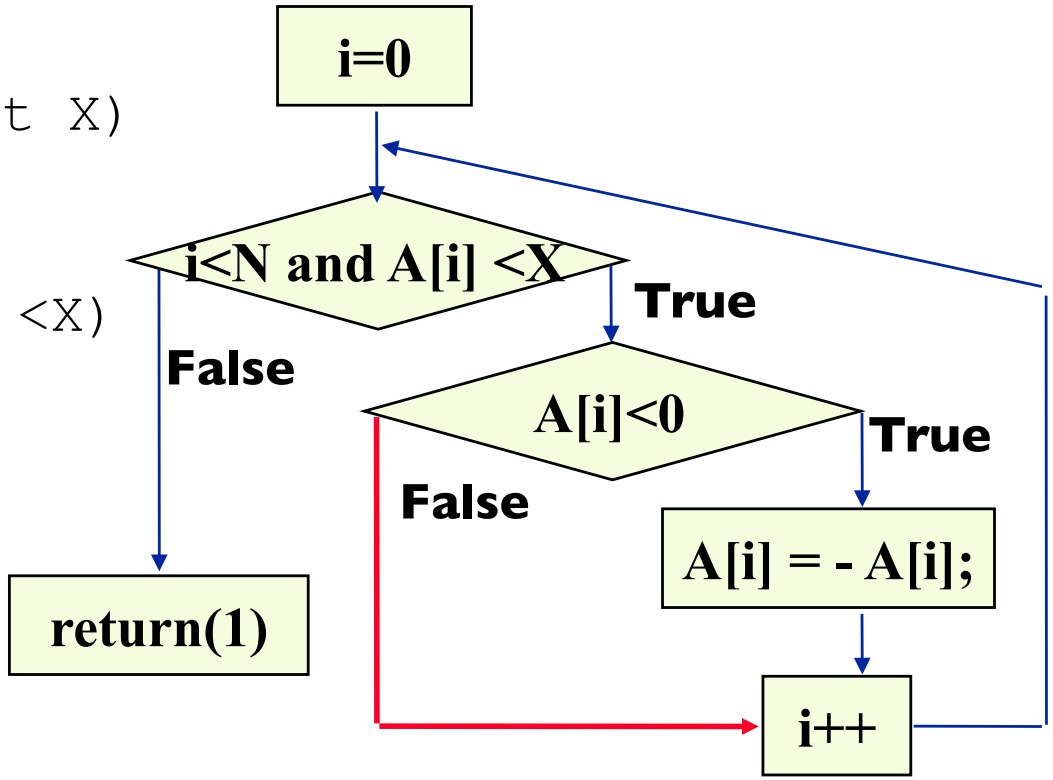
codice morto: fenomeno molto comune in code soggetto a continue modifiche di manutenzione

Criterio di copertura delle diramazioni (**branch** coverage)

- Selezionare un insieme T di dati di test tale che ogni diramazione del flusso di controllo venga selezionata almeno una volta da qualche elemento di T

Esempio

```
int select(int A[],
           int N, int X)
{
  int i=0;
  while (i<N && A[i] <X)
  {
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



Aggiungiamo il test (N=1, A[0]=7, X=9) per coprire il ramo "falso". Questo rileva errori nel caso A[i] positivo o nullo, ma non rileva errori dovuti all'uscita con A[i] <X falso

Anche qui

- Valutazione della copertura

$$\frac{\textbf{Numero-Branch-Coperti}}{\textbf{Numero-Totale-Branch}} \times 100$$

- ….potrebbe essere lontana da 100%

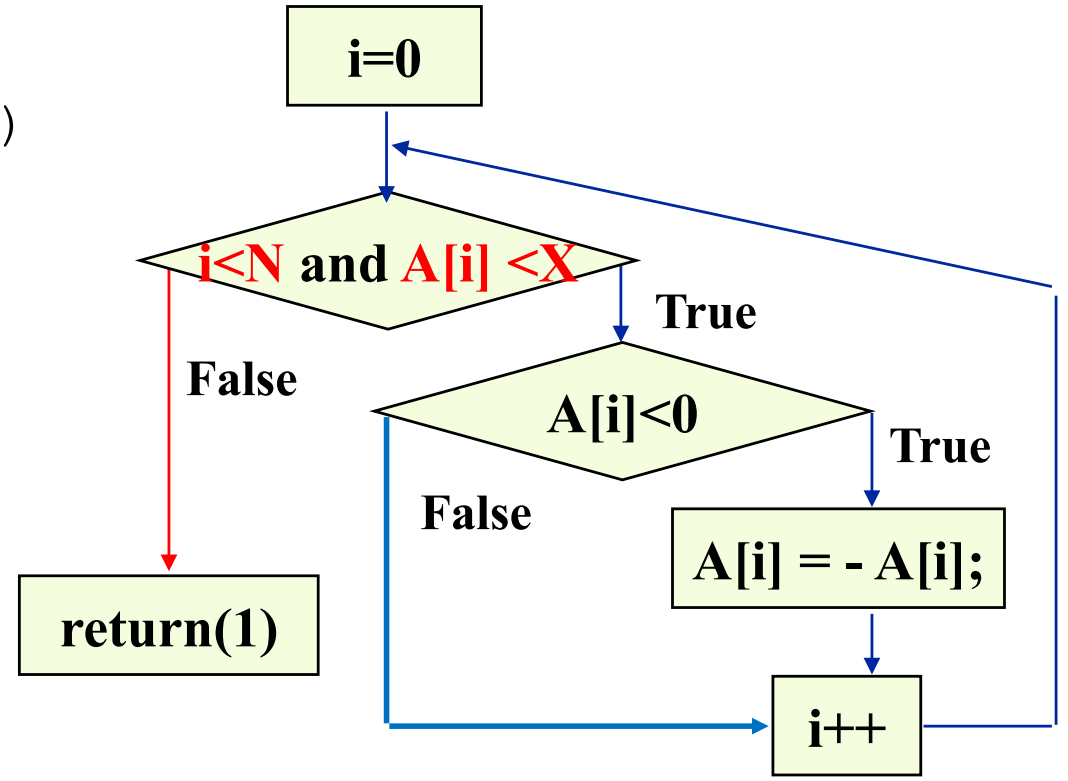
- Se il goal è coprire un certo branch occorre trovare dati che percorrano un cammino che arriva a quel branch
- Occorre poi trovare la condizione sui dati di ingresso che consente che tale cammino venga percorso
- Infine occorre trovare un dato che soddisfa la condizione

Criterio di copertura delle condizioni

- Selezionare un insieme T per cui si percorre ogni diramazione e tutti i possibili valori dei costituenti della condizione che controlla la diramazione sono esercitati almeno una volta

Copertura delle Condizioni

```
int select(int A[],
           int N, int X)
{
  int i=0;
  while (i<N && A[i] <X)
  {
     if (A[i]<0)
        A[i] = - A[i];
     i++;
  }
  return(1);
}
```



Non basta che (i<N), (A[i]<X) siano entrambe vere (entrata nel ciclo) ed una delle due falsa (uscita dal ciclo); occorre anche che siano l'una vera e l'altra falsa, l'una falsa e l'altra vera, ma non rileverebbe comunque errori che sorgono dopo parecchie iterazioni del ciclo

Confronto white-box/black box

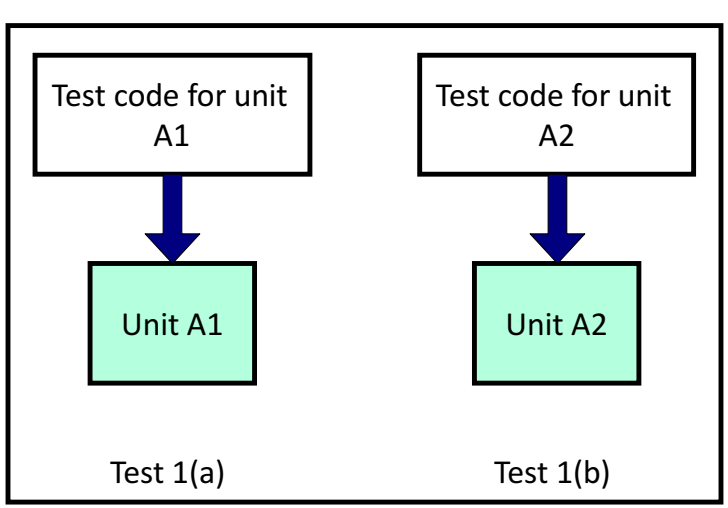
- Black box più semplice, più intuitivo e più diffuso
  - Non è necessario essere specialisti, basta conoscere il sistema e gli strumenti di testing
  - Però richiede una buona specifica
- White box è complementare e consente di arrivare ad avere una maggiore confidenza sulla correttezza
  - Vi fidereste di software in cui certe istruzioni non sono mai state eseguite durante il testing?
  - In pratica è fattibile solo dall'organizzazione che ha prodotto il software (codice proprietario vs. open source)

Test di unità, di integrazione e di sistema

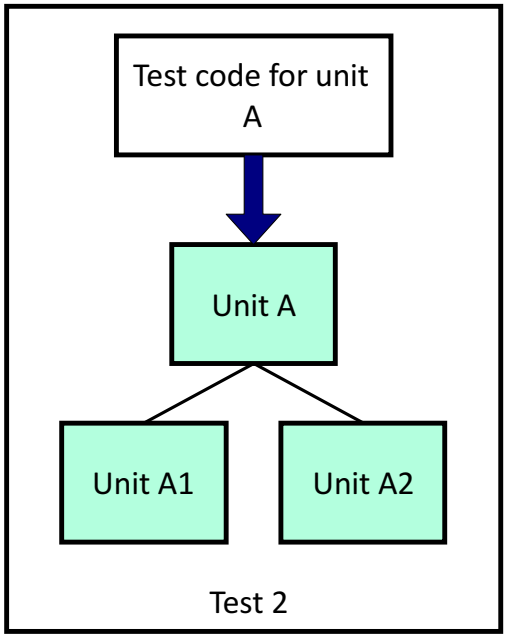
- **Test di unità**:
  - Ogni modulo viene verificato e testato isolatamente
  - Si continua fino a quando si ritiene che i moduli siano stati testati abbastanza
- **Test di integrazione**:
  - I moduli vengono gradualmente integrati in sottosistemi, effettuando opportune verifiche della loro corretta interazione
- **Test di sistema**:
  - Il sistema completo e finito viene convalidato rispetto ai suoi requisiti funzionali (specifica) e non funzionali (prestazioni, affidabilità)
- **Test di regressione**:
  - Controllo del comportamento di release successive, o a cavallo di fasi di refactoring

Test di Integrazione

Fase 1: testare unità individualmente



Integrare unità A1 e A2 a formare A



Fase 2: testare l'unità risultante

Il test deve esercitare tutte le caratteristiche di A1 e A2

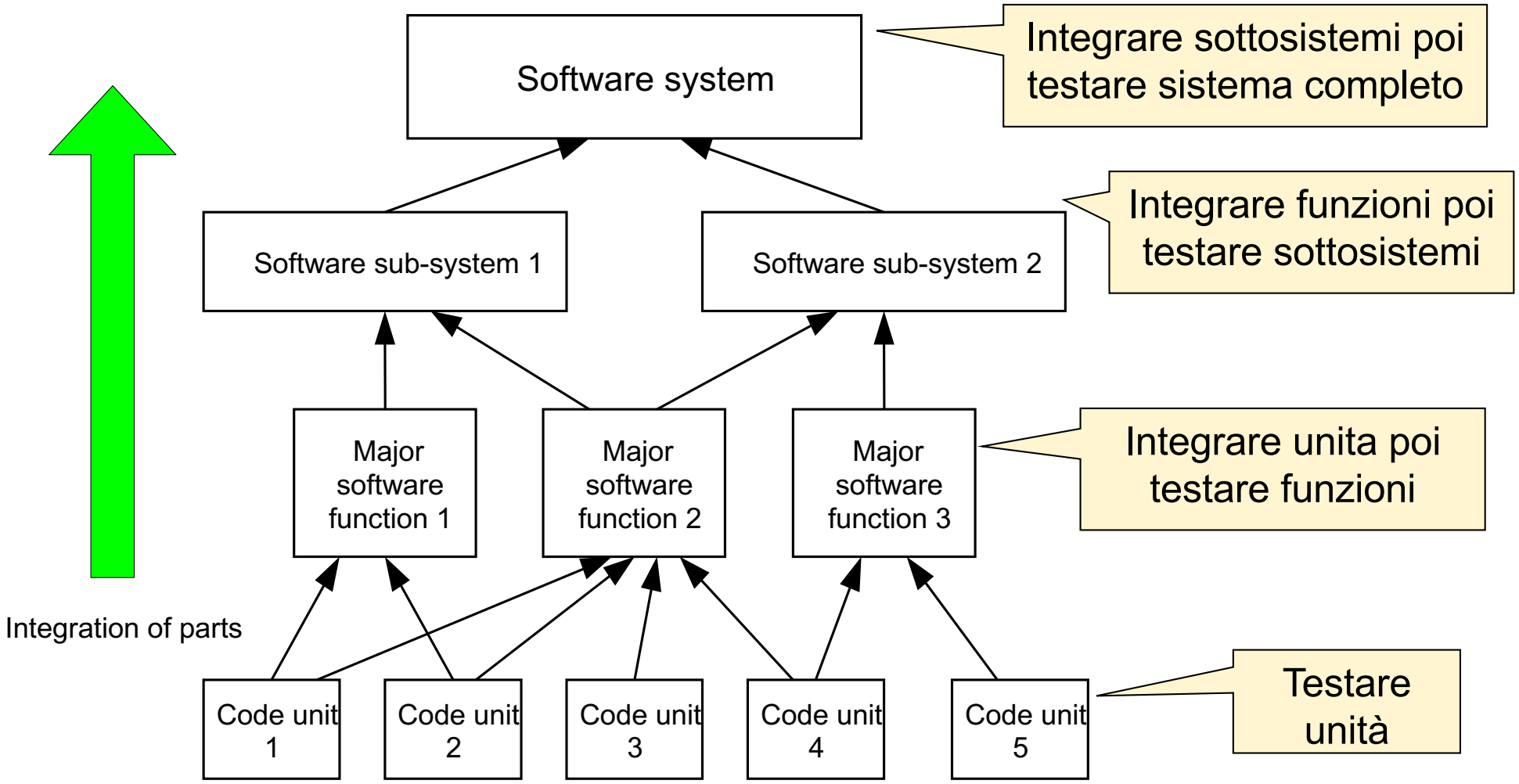
Incrementale vs. big-bang

- Approccio big bang: integrare tutti assieme i moduli precedentemente testati e verificare quindi l'intero sistema
  - Non è conveniente : se ci sono errori dovuti a "cattiva comunicazione" fra moduli come fare a trovarli?
- Approccio incrementale: integrare i moduli via via che vengono prodotti e testati singolarmente

Integrazione incrementale

- Richiede meno moduli fittizi e moduli guida
- Permette di rilevare, e quindi di eliminare, durante lo sviluppo del sistema, eventuali anomalie delle interfacce, evitando che queste permangano fino al prodotto finale
- Permette di localizzare e quindi di rimuovere più facilmente le anomalie
- Assicura che ciascun modulo venga esercitato più a lungo, perché esso viene integrato incrementalmente e quindi testato anche durante il test di integrazione di altri moduli

Passi per testing integrazione dei sistemi sw



AUTOMAZIONE DEL TEST

Esecuzione dei casi di test

- Quando si testa un programma è importante definire esattamente i risultati attesi
- Si parla di “oracolo”
- Si può automatizzare sia l'esecuzione dei test che il controllo dei risultati

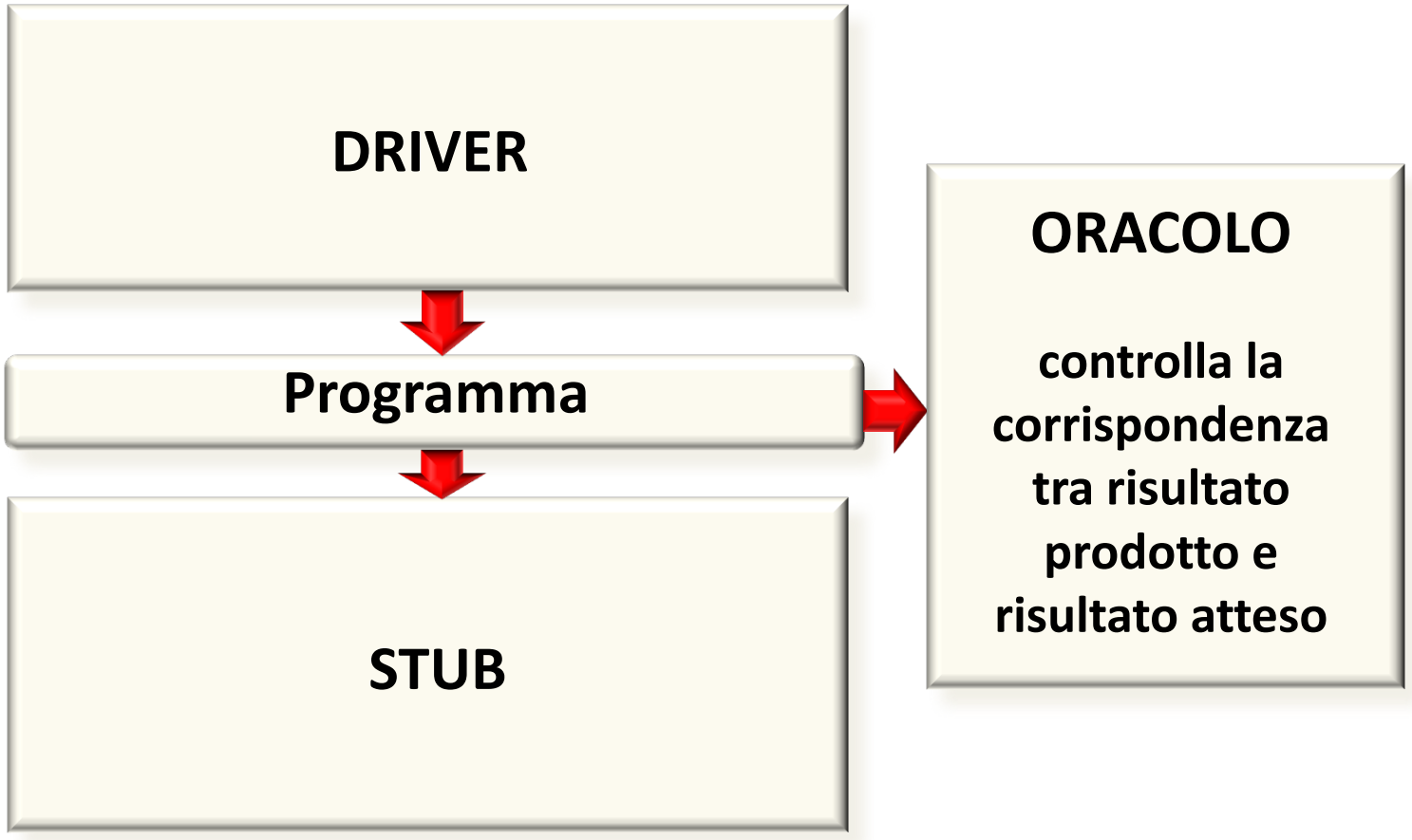
Automazione del testing

- In presenza di unità chiamante e unità chiamata servono
  - **Driver (modulo guida)**: simula la parte di programma che invoca l'unità oggetto del test
  - **Stub (modulo fittizio)**: simula la parte di programma chiamata dall'unità oggetto del test

Il problema dello scaffolding

- Lo **scaffolding** è estremamente importante per il test di unità e integrazione
- Può richiedere un notevole sforzo di programmazione
- Uno scaffolding buono è un passo importante per test di regressione efficiente
- La generazione di scaffolding può essere parzialmente automatizzata a partire dalle specifiche

Creare lo scaffolding



Automazione del testing (cont.)

- Cosa fa un driver
  - Prepara l'ambiente per il chiamato (e.g. crea e inizializza variabili globali, apre file ...)
  - Fa una serie di chiamate (può leggere i parametri da file ...)
  - Verifica risultati delle chiamate (con oracolo o usando risultati predisposti, magari su file) e li memorizza (su file)
- Cosa fa uno stub
  - Verifica ambiente predisposto dal chiamante
  - Verifica accettabilità parametri passati dal chiamante
  - restituisce risultati, esatti rispetto alle specifiche o "accettabili" per il chiamante (cioè, che gli permettono di proseguire in maniera sensata...)

Strumenti di testing di unità (unit test framework)

- Sono tool tipicamente orientati a un singolo linguaggio di programmazione
- Ad esempio, per Java esiste Junit (http://junit.org/index.htm)
  - Di basa sull'idea "first testing then coding"
  - "…test a little, code a little, test a little, …
- In C, Unity (anche per sistemi embedded)
- Consentono di costruire **test harness** (preparare driver, stub, test script, asserzioni, rapporti, ecc.)