Verifica e validazione: analisi statica

Anno accademico 2019/2020
Ingegneria del Software

Tullio Vardanega, tullio.vardanega@unipd.it

Premessa – 1

- Un SW di buona qualità deve possedere
  - Tutte le capacità funzionali specificate nei requisiti, che determinano cosa il sistema debba fare
  - Tutte le caratteristiche non funzionali necessarie per garantire che il sistema lavori sempre come previsto
- Ciò richiede verifica di possesso di proprietà
  - Di costruzione: architettura, codifica, integrazione
  - D'uso: esperienza utente, precisione, affidabilità
  - Di funzionamento: prestazioni, robustezza, sicurezza

Premessa – 2

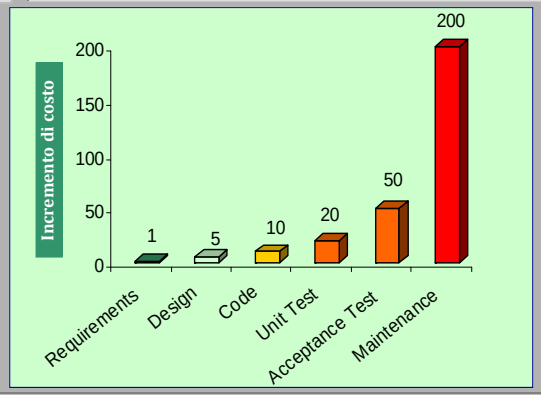
- La programmazione non deve ostacolare la verifica
  - Pochi linguaggi la facilitano attivamente
  - La disciplina del programmatore è necessaria
- Vi è tensione tra aumento del potere espressivo (funzionalità) vs. costo di verifica (integrità)
  - La prima via è affidarsi ad API «*black-box*»
  - La seconda via persegue pieno controllo sull'esecuzione
- I costrutti del linguaggio di programmazione scelto vanno valutati in funzione del loro impatto su quel bilancio

Scrivere programmi verificabili – 1

- Dotarsi di uno standard di codifica coerente con le esigenze di verifica
  - Promuovendo buone prassi e ponendo vincoli sui costrutti di programmazione inappropriati
- La verifica retrospettiva è insufficiente

- Il costo di rilevazione e correzione di errori cresce con l'avanzare dello sviluppo

Costo di correzione di errori



Scrivere programmi verificabili – 2

- **L'approccio reattivo alla verifica è ingenuo, pigro, ottimistico**
  - ***Seeking correctness by correction***

- **Sostenere lo sviluppo con la verifica costituisce un approccio costruttivo**
  - ***Pursuing correctness by construction***

Scrivere programmi verificabili – 3

- **Regolamentare l'uso del linguaggio di programmazione tramite principi da riflettere nelle Norme di Progetto**
  - **Per assicurare comportamento predicibile**
  - **Per usare criteri di programmazione ben fondati**
  - **Per ragioni pragmatiche**
- **Vediamo ciascuna di queste tre dimensioni**

Comportamento predicibile

- **Codice sorgente senza ambiguità**
  - **Effetti laterali (p.es. di sottoprogrammi)**
    - Invocazioni della stessa azione che producano esiti diversi
  - **Ordine di elaborazione e inizializzazione**
    - L'effetto del programma può dipendere dall'ordine di elaborazione delle sue parti
    - Esempio: l'impredicibilità dell'attivazione di *thread* in Java
  - **Modalità di passaggio dei parametri**
    - La scelta di una modalità di passaggio (per valore, per riferimento) può influenzare l'esito dell'esecuzione

Funziona?

```
class Swapper{
  public static void swap(int Left, int Right)
  {
    int tmp = Left;
    Left = Right;
    Right = Left;
  }

  public static void main(String args[])
  {
    int Source = 1;
    int Destination = 3;
    swap(Source, Destination);
  }
}
```

In Java, i nomi sono riferimenti, ma le chiamate sono per valore!

Criteri di programmazione

- **Riflettere l'architettura (*design*) nel codice**
  - Usare programmazione strutturata per esprimere componenti, moduli, unità come da progettazione, e facilitare il riuso
- **Separare le interfacce dall'implementazione**
  - Fissare bene le interfacce già a partire dall'architettura logica
  - Esporre le prime, nascondere la seconda
- **Massimizzare l'incapsulazione (*information hiding*)**
  - Usare membri privati e metodi pubblici per l'accesso
- **Usare tipi specializzati per specificare dati**
  - La composizione e la specializzazione aumentano il potere espressivo del sistema di tipi del programma

Considerazioni pragmatiche

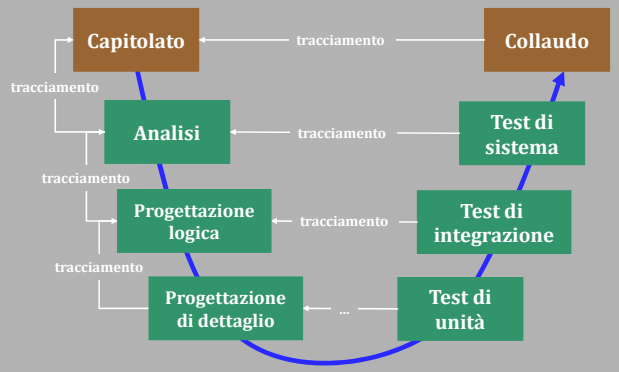
- **L'efficacia dei metodi di verifica è funzione della qualità di strutturazione del codice**
  - **Esempio: una procedura con un solo punto di uscita è più facilmente analizzabile per il suo effetto sullo stato**
- **La verifica di un programma relaziona frammenti di codice con frammenti di specifica**
  - **Verificabilità, funzione inversa dell'ampiezza del contesto**
    - Più cresce il secondo, più diminuisce la prima: confinare *scope* e visibilità
  - **Una buona architettura facilita la verifica**
    - Esempio: incapsulazione dello stato e controllo di accesso

Tracciamento – 1

- **Per dimostrare completezza ed economicità della soluzione**
  - **Nessun requisito dimenticato**
  - **Nessuna funzionalità superflua**
  - **Nessun componente ingiustificato**

- **Applicarlo**
  - **Su ogni passaggio dello sviluppo (ramo discendente)**
  - **Su ogni passaggio della verifica (ramo ascendente)**
- **Può essere altamente automatizzato**

Tracciamento – 2



Tracciamento sul codice

- Le regole di programmazione possono facilitare il tracciamento
  - Assegnare singoli requisiti elementari a singoli moduli del programma richiede una sola procedura di prova
  - Semplificando la verifica e facilitando il tracciamento
- Maggiore l'astrazione (potenza espressiva) di un costrutto maggiore la quantità di codice oggetto eseguito per esso e maggiore l'onere di dimostrazione di corrispondenza
  - Bassa astrazione: addizione tra interi
  - Alta astrazione: attivazione di *thread*

Tipi di analisi statica del codice

A. Flusso di controllo
B. Flusso dei dati
C. Flusso dell'informazione
D. Esecuzione simbolica
E. Verifica formale del codice
F. Verifica di limite
G. Uso dello *stack*
H. Comportamento temporale
I. Interferenza
J. Codice oggetto

Prima di e in aggiunta all'analisi dinamica

Analisi di flusso di controllo

- Accertare
  - Logica: il codice esegue nella sequenza specificata
  - Visibilità e propagazione: il codice è ben strutturato
- Localizzare codice non raggiungibile
- Identificare segmenti d'esecuzione che possano non terminare
  - L'analisi dell'albero delle chiamate (*call-tree analysis*) mostra se l'ordine di chiamata corrisponda alla specifica e rileva la presenza di ricorsione diretta o indiretta
  - Divieto di modifica di variabili di controllo delle iterazioni

Analisi di flusso dei dati

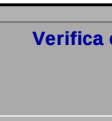
- Accertare che nessun cammino d'esecuzione del programma acceda a variabili prive di valore
  - Concentrando l'analisi di flusso di controllo sulla sequenza e le modalità di accesso alle variabili (lettura, scrittura)
- Rilevare possibili anomalie
  - Esempio: più scritture successive senza letture intermedie
- Evitare presenza e uso di dati globali raggiungibili da più parti del programma
  - Violazione di incapsulazione

Analisi di flusso d'informazione

- Determinare quali dipendenze tra ingressi e uscite risultino dall'esecuzione di una unità di codice
  - Per identificare effetti laterali inattesi o indesiderati
- Le sole dipendenze ammissibili sono quelle previste o implicate dalla specifica
- Può limitarsi a un singolo modulo oppure estendere a più unità correlate oppure anche all'intero sistema

Verifica formale del codice

- Provare la correttezza del codice sorgente rispetto alla specifica algebrica dei requisiti
  - Esplorando tutte le esecuzioni possibili
    - Non fattibile tramite analisi dinamica
- Correttezza parziale (sotto ipotesi di terminazione del programma)
  - Oggetto di verifica espresso come teorema la cui verità postula certe pre-condizioni in ingresso e certe post-condizioni in uscita
    - La prova di correttezza totale richiede prova di terminazione



Analisi di limite

- Verificare che i dati del programma restino entro i limiti del loro tipo e della precisione desiderata
  - Analisi di *overflow* e *underflow*, per evitare di trattare valori non rappresentabili dal processore
  - Analisi di errori di arrotondamento (calcolo numerico!)
  - Rispetto dei limiti (*range checking*) nelle iterazioni e nell'attraversamento di strutture dati
- Linguaggi evoluti assegnano limiti statici a tipi discreti consentendo verifiche automatiche sulle corrispondenti variabili
  - Più problematico con tipi enumerati e reali

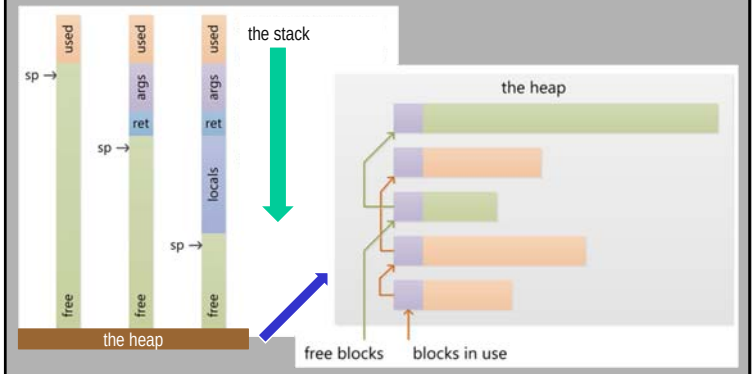
Analisi d'uso di *stack*

- Determinare la massima domanda di *stack* richiesta a tempo d'esecuzione in relazione con la dimensione della memoria assegnata al processo (programma in esecuzione)
- Verificare che non vi sia rischio di collisione tra *stack* e *heap* per qualche esecuzione

Stack & heap – 1

- Lo *stack* è l'area di memoria usata per ospitare dati locali e indirizzi di ritorno generati dal compilatore alla chiamata di sottoprogrammi
  - Ogni flusso di controllo (*main* e *thread*) ha il suo *stack*
  - La sua dimensione cresce con l'annidamento di chiamate
  - I dati in esso hanno regole di visibilità e ciclo di vita
- L'*heap* è la memoria globale del programma
  - La sua dimensione è fissata a configurazione
  - Il suo contenuto è determinato dalla dimensione degli oggetti globali creati dinamicamente

Stack & heap – 2



Cosa va nello *stack* e cosa nell'*heap*?

```
class Swapper{
  public static void swap(int Left, int Right)
  {
    int tmp = Left;
    Left = Right;
    Right = Left;
  }

  public static void main(String args[])
  {
    int Source = 1;
    int Destination = 3;
    swap(Source, Destination);
  }
}
```


Verifica e validazione: analisi statica
Analisi temporale
Studiare le dipendenze temporali (latenza) tra le uscite del programma e i suoi ingressi
Per verificare che il valore giusto sia prodotto al momento giusto
Limiti espressivi dei linguaggi e delle tecniche di programmazione complicano questa analisi
Iterazioni prive di limite statico (*while*)
Creazione dinamica di variabili (*new*)
...
Laurea in Informatica, Università di Padova
25/26


Verifica e validazione: analisi statica
Analisi d'interferenza
Mostrare l'assenza di effetti di interferenza tra parti isolate ("partizioni") del sistema
Non necessariamente limitate a componenti SW
Veicoli tipici di interferenza
Memoria (virtualmente) condivisa, dove parti separate di programma lasciano traccia di dati abbandonati ma non distrutti
Fenomeno noto come *memory leak*
Azzeramento delle pagine di memoria prima del riuso (p.es. NT v5.x)
I/O programmabile (p.es., DMA) e registri di periferiche
Variabili di tipo *volatile*
Laurea in Informatica, Università di Padova
26/26