Verifica e validazione: analisi statica

Anno accademico 2018/19
Ingegneria del Software

Tullio Vardanega, tullio.vardanega@math.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

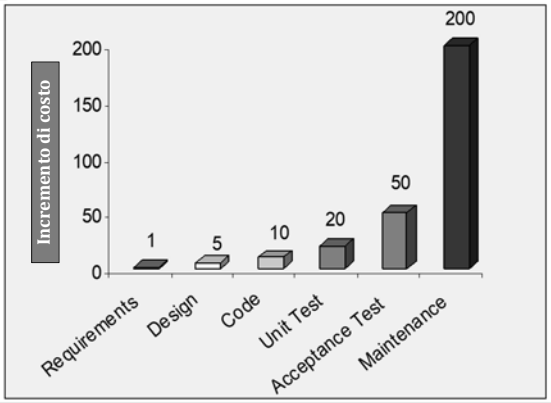
- La programmazione non deve ostacolare la verifica
  - Pochi linguaggi la facilitano attivamente
  - La disciplina del programmatore è fattore critico
- Occorre bilanciare la spinta all'aumento del potere espressivo con il costo di verifica
  - Funzionalità vs. integrità
  - La prima è, p.es., affidarsi ad API «*black-box*»
  - La seconda vuole consapevolezza di come sarà l'esecuzione richiesta
- Fissato il linguaggio di programmazione, occorre sempre scegliere quali costrutti usarne in funzione del loro impatto sull'equilibrio di quel bilancio

Scrivere programmi verificabili – 1

- Serve dotarsi di uno standard di codifica coerente con le esigenze di verifica
  - Buone prassi attive e restrizioni sui costrutti inappropriati
- La verifica solo retrospettiva (a valle dello sviluppo) 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

- **Un approccio di verifica *as-late-as-possible* è ottimisticamente retrospettivo**
  - ***Seeking correctness by correction***

- **Sostenere la produzione 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 possono produrre risultati diversi
  - **Ordine di elaborazione e inizializzazione**
    - L'effetto del programma può dipendere dall'ordine di elaborazione delle sue parti
    - Pensate all'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);
  }
}
```


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

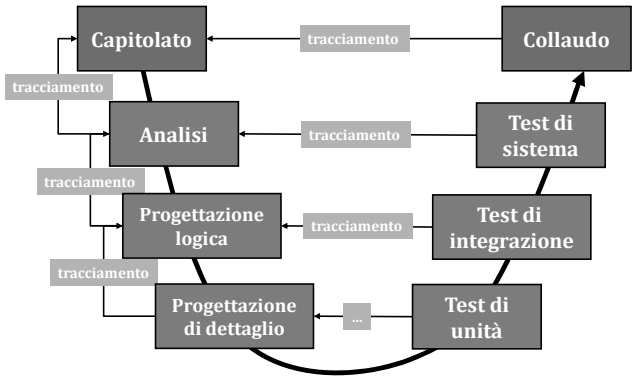
- **L'efficacia dei metodi di verifica è funzione della qualità di strutturazione del codice**
  - **Esempio: una procedura con un solo punto di ingresso e un solo punto di uscita è più facilmente analizzabile per il suo effetto sullo stato**
- **La verifica di un programma relaziona frammenti di codice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

Tracciamento – 1

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

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

Tracciamento – 2



Tracciamento sul codice

- **Particolari stili di programmazione facilitano il tracciamento**
  - **Assegnare singoli requisiti elementari a singoli moduli del programma richiede una sola procedura di prova**
  - **Questo semplifica la verifica e facilita 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**

Analisi di flusso di controllo

- **Accertare che**
  - **Il codice esegua nella sequenza specificata**
    - Controllo di logica
  - **Il codice sia ben strutturato**
    - Controllo di visibilità
- **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
  - Consente l'identificazione di effetti laterali inattesi o indesiderati
- Le sole dipendenze consentite sono quelle previste dalla specifica
- Può limitarsi a un singolo modulo oppure estendere a più unità correlate oppure anche all'intero sistema

Esecuzione simbolica

- Verificare proprietà del programma mediante manipolazione algebrica del codice sorgente
  - Combinando tecniche di analisi di flusso di controllo, flusso di dati, flusso di informazione
- Si esegue effettuando "sostituzioni inverse"
  - Sostituendo progressivamente ogni LHS di un assegnamento con il suo RHS
- Questo trasforma il flusso sequenziale di esecuzione in un insieme di assegnamenti paralleli il cui esito è funzione degli ingressi

Esempio

Assumendo assenza di *aliasing* e di effetti laterali di funzioni

```
X = A+B;  // X dipende da A, B
Y = D-C;  // Y dipende da C, D
if (X>0)
   Z = Y+1; // Z dipende da A, B, C, D
```

$$A + B \leq 0 \rightarrow X = A + B,\ Y = D - C,\ Z = Z$$
$$A + B > 0 \rightarrow X = A + B,\ Y = D - C,\ Z = D - C + 1$$

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
  - Le condizioni di verifica sono espresse come teoremi la cui verità implica certe pre-condizioni in ingresso e certe post-condizioni in uscita
    - La prova di correttezza vale sotto l'ipotesi di terminazione del programma
    - 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
  - Rispetto dei limiti (*range checking*)
  - Analisi di limite di strutture
- Linguaggi evoluti assegnano limiti statici a tipi discreti consentendo verifiche automatiche sulle corrispondenti variabili
  - Più problematico con tipi enumerati e reali

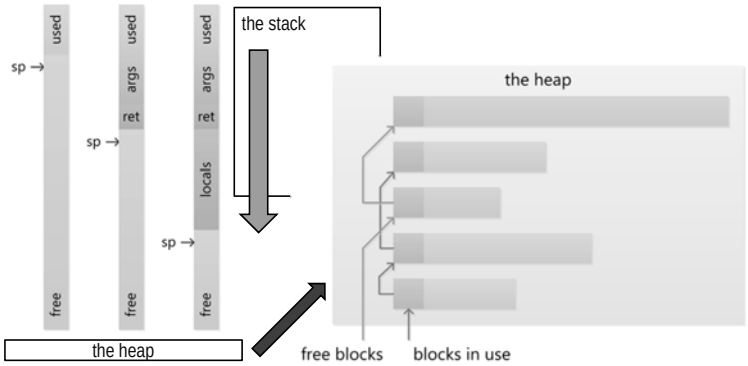
Analisi d'uso di *stack*

- Determinare la massima domanda di *stack* richiesta a tempo d'esecuzione in relazione con la dimensione dell'area di memoria assegnata al processo (cioè al 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 contenuto hanno un ben definito 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);
  }
}
```


Analisi temporale

- **Studiare le dipendenze temporali (latenza, tempestività) 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*), ricorso sistematico a strutture dati dinamiche (*new*), …**

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 e altri dispositivi programmabili con effetti a livello sistema (p.es., DMA)**

Analisi di codice oggetto

- **Assicurare che il codice oggetto da eseguire sia una traduzione corretta del codice sorgente a lui corrispondente e che nessun errore od omissione siano stati introdotti dal compilatore**
  - **Viene ancora effettuata manualmente**
  - **Viene facilitata dalle informazioni di corrispondenza prodotte dal compilatore**

Analizzabilità del sistema

- **L'analisi statica costruisce modelli astratti del programma**
  - **Rappresentandolo come un grafo diretto per studiare i cammini possibili in esso**
  - **Le transizioni tra stati (gli archi) hanno etichette che descrivono proprietà dell'istruzione corrispondente, rilevanti per l'esecuzione**
  - **La presenza di flussi di eccezione e di risoluzione dinamica di chiamata (*dynamic binding*) complica la struttura del grafo**
- **Ciascun flusso di controllo (*thread*) viene analizzato individualmente**
  - **Ma questo richiede assenza di interferenza tra loro!**