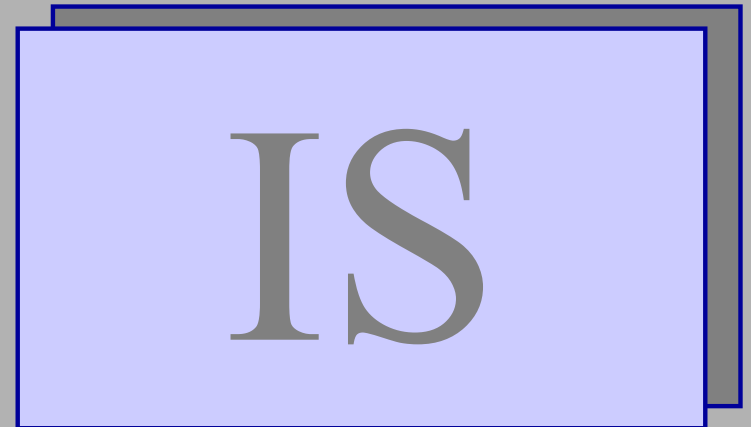
Verifica e validazione: analisi statica



Anno accademico 2020/2021

Ingegneria del Software

Tullio Vardanega, tullio.vardanega@unipd.it

Premessa – 1

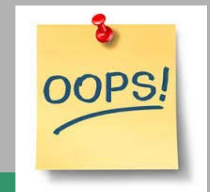
- **Un SW di qualità deve possedere**
  - **Tutte le capacità <u>funzionali</u> specificate nei requisiti, che determinano cosa il sistema debba fare**
  - **Tutte le caratteristiche <u>non-funzionali</u> necessarie per garantire che il sistema lavori sempre come previsto**
- **Ciò si dimostra tramite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

Premessa – 2

- La codifica deve aiutare la verifica, non ostacolarla
  - Pochi linguaggi la facilitano attivamente
  - Per questo serve la disciplina del programmatore
- La ricerca del potere espressivo (funzionalità) confligge con il costo di verifica (integrità)
  - La prima via è affidarsi a componenti «*black-box*»
  - L'attenzione al costo ricerca pieno controllo sull'esecuzione
- L'uso del linguaggio di programmazione adottato va normato in funzione del suo impatto sul bilancio tra funzionalità e integrità

Scrivere programmi verificabili – 1

- Dotarsi di norme di codifica coerenti con le esigenze di verifica
  - Promuovendo buone prassi e ponendo vincoli sui costrutti di programmazione inappropriati
  - Verificandone attivamente il rispetto
- La verifica retrospettiva è <u>insufficiente</u>



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

Costo di correzione di errori

Incremento di costo

200
150
100
50
0

Requirements	Design	Code	Unit Test	Acceptance Test	Maintenance
1	5	10	20	50	200

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
  1. Per assicurare comportamento predicibile
  2. Per usare buoni principi di programmazione
  3. Per ragioni pragmatiche
- Vediamo ciascuna di queste tre dimensioni

1. Comportamento predicibile

- **Codice sorgente senza ambiguità**
  - **Effetti laterali (p.es. di sottoprogrammi)**
    - Invocazioni della stessa azione che producano effetti 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!

2. Principi di programmazione

- Riflettere l'architettura (*design*) nel codice
  - Usare programmazione strutturata per esprimere componenti, moduli, unità come da progettazione, e facilitare l'integrazione
- 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 ai dati
- Usare tipi specializzati per specificare dati
  - La composizione e la specializzazione aumentano il potere espressivo del sistema di tipi del programma

3. 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 facilita l'analisi del suo effetto sullo stato**
- **La verifica di un programma relaziona frammenti di codice con frammenti di specifica**
  - **La sua verificabilità è funzione inversa dell'ampiezza del contesto**
    - Più cresce il secondo, più diminuisce la prima: confinare *scope* e visibilità
  - **Una buona architettura facilita la verifica**
    - P.es. tramite incapsulazione dello stato e controllo di accesso

Tracciamento – 1

- Dimostra completezza ed economicità del prodotto
  - Nessun requisito dimenticato
  - Nessuna funzionalità superflua
  - Nessun componente ingiustificato

- Va applicato
  - Su ogni passaggio dello sviluppo (ramo discendente)
  - Su ogni passaggio della verifica (ramo ascendente)
- Va automatizzato il più possibile
  - Per diminuirne il costo all'aumentare della sua intensità

Tracciamento – 2

Capitolato

tracciamento

Collaudo

tracciamento

Analisi

tracciamento

Test di sistema

tracciamento

Progettazione logica

tracciamento

Test di integrazione

tracciamento

Progettazione di dettaglio

…

Test di unità

Tracciamento – 3

- Tracciare i requisiti su progettazione di dettaglio e codifica aiuta a valutare il costo di verifica
  - Assegnare N requisiti elementari a 1 singolo modulo SW richiede N procedure di prova per quel modulo

    (1 prova per 1 requisito aiuta a rendere le prove decidibili)
  - Al crescere di N crescono la criticità e il costo di quel modulo
- Maggiore il potere espressivo di un costrutto, maggiore la sua complessità di esecuzione, maggiore il costo di dimostrarlo corretto
  - Basso potere espressivo: addizione tra interi, …
  - Alto potere espressivo: attivazione di *thread*, invocazione di API *black-box*, …

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

- **Per accertare**
  - **Logica: il codice esegue nella sequenza specificata**
  - **Visibilità e propagazione: il codice è ben strutturato**
- **Per localizzare codice non raggiungibile**
- **Per identificare rischi di non terminazione**
  - **L'analisi dell'albero delle chiamate (*call-tree analysis*), mostrando se l'ordine di chiamata corrisponda alla specifica, segnala anche la presenza di ricorsione diretta o indiretta**
  - **La modifica di variabili di controllo delle iterazioni è fonte di vulnerabilità**

Analisi di flusso dei dati

- **Per 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)**
- **Per rilevare possibili anomalie**
  - **Più scritture successive senza letture intermedie**
  - **Letture che precedano scritture**
- **Conviene evitare l'uso di dati globali raggiungibili da più parti del programma**
  - **In violazione del principio di incapsulazione**

Analisi di flusso d'informazione

- Per determinare le dipendenze tra ingressi e uscite determinate dall'esecuzione di singole unità di codice
  - Identificando effetti laterali inattesi o indesiderati
  - Le sole dipendenze ammissibili sono quelle previste o implicate dalla specifica
- Può limitarsi a singoli moduli/unità ma anche estendere alla loro integrazione fino 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

- Per verificare che i valori del programma restino entro i limiti del loro tipo e della precisione desiderata
  - L'*overflow* produce valori maggiore del massimo rappresentabile e possono causare eccezione o silenziosamente produrre valori errati
  - L'*underflow* produce valori più piccoli del minimo rappresentabile e possono causare eccezione o importanti errori di arrotondamento
  - Rispetto dei limiti (*range checking*) nell'attraversamento di strutture dati
- Alcuni linguaggi permettono di assegnare limiti statici a tipi discreti per consentire al compilatore di fare verifiche sulle corrispondenti variabili
  - Più difficile farlo con tipi enumerati e reali

Analisi d'uso di *stack*

- Per determinare la massima domanda di *stack* richiesta a tempo d'esecuzione in relazione con la dimensione della memoria assegnata al processo (programma in esecuzione)
- Per 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 massima è fissata a configurazione
  - Il suo contenuto è determinato dagli oggetti globali creati a tempo d'esecuzione

Stack & *heap* – 2

Lo *stack* cresce verso l'heap …

L'heap è organizzato in blocchi …

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

- Per 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*)
  - …

Analisi d'interferenza

- Per mostrare l'assenza di effetti di interferenza tra parti isolate ("partizioni") del sistema
  - Non necessariamente limitate a componenti SW
- Veicoli tipici di interferenza
  - Memoria virtuale condivisa: parti separate di programma lasciano traccia di dati abbandonati ma non distrutti
    - Fenomeno detto ***memory leak***, mitigato dall'azzeramento dei *page frame* rilasciati dal programma
  - I/O programmabile (p.es., DMA) e registri di periferiche
    - Variabili di tipo ***volatile***