

# Ingegneria del Software

## 21. Verifica e validazione

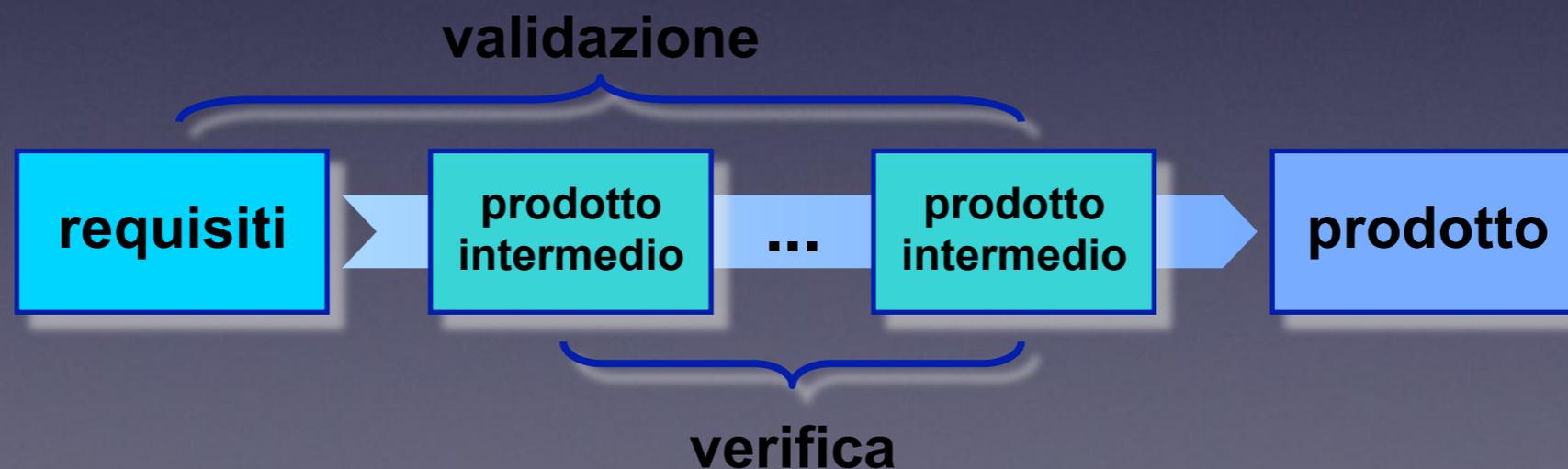
Dipartimento di Informatica  
Università di Pisa  
A.A. 2014/15

# roadmap

- Concetti e terminologia
- Verifica, validazione, integrazione e collaudo
- Verifica statica
- Inspection e walkthrough

# verifica e validazione

- Attività necessarie per essere sicuri
  - per essere sicuri di non aver introdotto difetti
  - per essere sicuri di aver realizzato il prodotto giusto



da adattare rispetto al  
modello del ciclo di vita

# errori, difetti, malfunzionamenti

- Errore
  - causa (umana) — meccanico distratto
  - può non essere rilevante
  - Magari latente, da eliminare
- Difetto (*bug*) [errore in *Binato et al.*]
  - Introdotto da un errore — bullone avvitato male
  - Malfunzionamento
    - prodotto da un difetto — perdita di una ruota
    - di solito è un danno

# controlli interni/esterni

- Interni alla azienda che ha sviluppato il sw
  - alfa
- Esterni
  - beta
  - collaudo tipico esempio

# verifiche statiche/dinamiche

- Verifiche statiche
  - non prevedono l'esecuzione del codice
  - usate specificamente per attività di verifica
  - eseguite sulle componenti
- Verifiche dinamiche (prove)
  - comportano l'esecuzione del codice
  - usate per attività di verifica o di validazione
  - eseguite sulle componenti o sul sistema (validazione)

# test ideale

- Prova formale di correttezza
  - corrisponderebbe all'esecuzione del sistema con tutti i possibili input (“A convincing demonstration of correctness being impossible as long as the mechanism is regarded as a black box, our only hope lies in not regarding the mechanism as a black box”, Edsger W. Dijkstra, 1970)
  - è chiaramente impossibile (“Program testing can be used to show the presence of bugs, but never to show their absence!”, *idem*)
- Due soluzioni possibili
  - verifiche statiche su modelli o con astrazioni dei dati in ingresso
  - verifiche dinamiche con “buona” selezione dei dati in ingresso

# la verifica statica

- Verifica senza esecuzione del codice
- Metodi manuali
  - basati sulla lettura del codice (*desk-check*)
  - più comunemente usati
  - più o meno formalmente documentati
- Metodi formali or Statica supportata da strumenti
  - model checking
  - esecuzione simbolica

# perché la verifica statica

- Praticità e intuitività (desk-check)
- Ideale per alcune caratteristiche di qualità
- Convenienza economica
  - costi dipendenti dalle dimensioni del codice
  - bassi costi di infrastruttura (desk-check)
  - buona prevedibilità dei risultati

# metodi di lettura del codice

- Inspection e Walkthrough
- Sono metodi pratici
  - basati sulla lettura del codice
  - dipendenti dall'esperienza dei verificatori
  - per organizzare le attività di verifica
  - per documentare l'attività e i suoi risultati
- Sono metodi complementari tra loro

# inspection

- Obiettivi

- rivelare la presenza di difetti
- eseguire una lettura mirata del codice

- Agenti

- verificatori diversi dai programmatori

- Strategia

- focalizzare la ricerca su aspetti ben definiti (*error guessing*)
- off-by-one error (aka Obi-Wan error)

# attività dell'inspection

- Fase 1: pianificazione
- Fase 2: definizione della lista di controllo
- Fase 3: lettura del codice
- Fase 4: correzione dei difetti

# walkthrough

- Obiettivo
  - rivelare la presenza di difetti
  - eseguire una lettura critica del codice
- Agenti
  - gruppi misti ispettori e sviluppatori
- Strategia
  - percorrere il codice simulandone l'esecuzione

# attività di walkthrough

- Fase 1: pianificazione
- Fase 2: lettura del codice
- Fase 3: correzione dei difetti

# inspection vs walkthrough

- Affinità

- controlli statici basati su desk-test
- programmatori e verificatori contrapposti
- documentazione formale

- Differenze

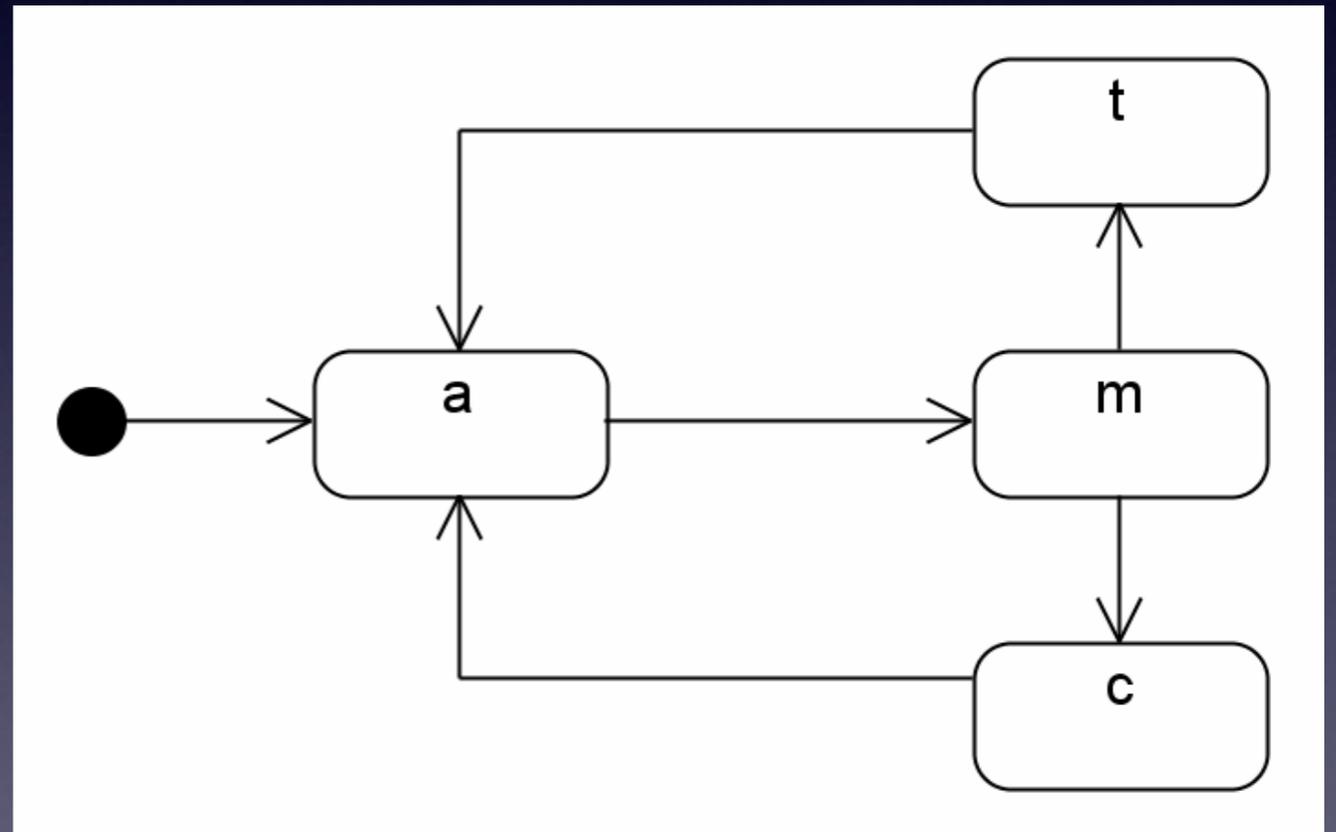
- inspection basato su errori presupposti
- walkthrough basato sull'esperienza
- walkthrough più collaborativo
- inspection più rapido

# model checking

- “Given a model of a system, exhaustively and automatically check whether this model meets a given specification” [wiki]
- ~1980 per circuiti hw
  - circuito modellato con macchine a stati finiti
- In seguito applicato anche (soprattutto) al sw
- Specifica comportamento (modello)
  - normalmente con sistemi di transizioni (nodi/archi)
- Proprietà specificate con logica temporale
  - A. Pnueli, E. M. Clarke, E. A. Emerson and J. Sifakis

# model checking: un modello

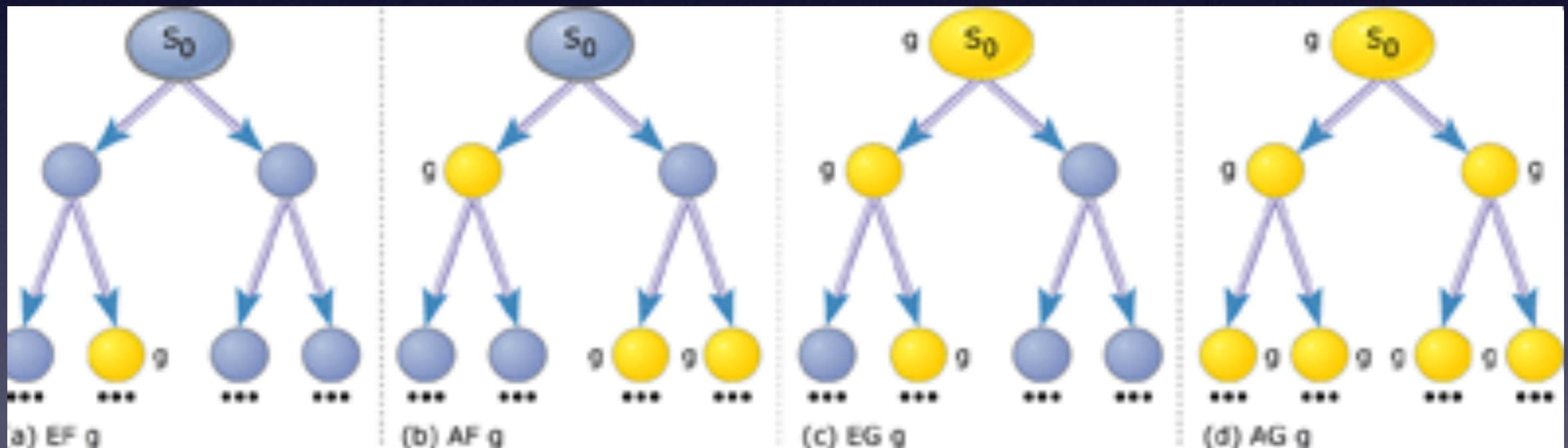
- Inizialmente in stato di attesa (a)
- Dopo l'inserimento di una moneta si passa a uno stato "moneta inserita" (m)
- Infine è consegnato un caffè (c) o un tè (t)



# alcune proprietà

- È sempre vero che se è stata inserita una moneta, allora in tutti i cammini del modello si incontra prima o poi uno stato nel quale viene consegnato un tè o uno stato nel quale viene consegnato un caffè
- Non vengono mai consegnati allo stesso tempo sia il tè che il caffè
- È sempre vero che se è stata inserita una moneta, allora in tutti i cammini del modello si incontra prima o poi uno stato nel quale viene consegnato un tè

# alcuni operatori temporali



# alcune proprietà

- È sempre vero che se è stata inserita una moneta, allora in tutti i cammini del modello si incontra prima o poi uno stato nel quale viene consegnato un tè o uno stato nel quale viene consegnato un caffè
  - $AG [m \rightarrow AF (c \text{ OR } t)]$
- Non vengono mai consegnati allo stesso tempo sia il tè che il caffè
  - $AG \sim (c \text{ AND } t)$
- È sempre vero che se è stata inserita una moneta, allora in tutti i cammini del modello si incontra prima o poi uno stato nel quale viene consegnato un tè
  - $AG [m \rightarrow AF t]$

# 1986: algoritmo Clarke Emerson Sistla

- Data una specifica  $M$  e una formula  $f$ , si verifica se  $M$  soddisfa  $f$  ( $M \models f$ ), cioè, se  $M$  è un modello per  $f$
- Se la risposta è negativa, di solito viene prodotto un contro-esempio, ovvero, un cammino nel modello che non verifica la (sotto)formula
- L'algoritmo attraversa lo spazio degli stati etichettandoli con sotto-formule, ed è quadratico rispetto al numero degli stati (lineare in varianti)

# problema

- State explosion
  - numero di stati esponenziale nel numero dei processi coinvolti
  - algoritmi di mc classici non vanno oltre il milione di stati
  - possibile soluzione: interleaving vs. true concurrency

# esecuzione simbolica

- L'esecuzione di un programma viene simulata
  - si rappresentano insiemi di possibili dati d'ingresso con simboli algebrici, piuttosto che con valori numerici reali

# esec. simbolica: un esempio

```
public static int abs (int i) { //L1
int r;
if (i<0)
    r = -i; //L2
else
    r = i; //L3
//L4
    return r;
}
```

PS Cosa direbbe il compilatore Java?

- L'esecuzione simbolica simula l'esecuzione, associando a ogni stato un'espressione che ne descrive le proprietà, in funzione dei valori assunti dalle variabili

passi	stato	note
L1	$i = l$	è un intero
L2	$i = l \ \& \ (l < 0 \ \& \ r = -l)$	
L3	$i = l \ \& \ (0 \leq l \ \& \ r = l)$	
L4	$i = l \ \& \ ((l < 0 \ \& \ r = -l) \   \ (0 \leq l \ \& \ r = l))$	I rami del condizionale si ricongiungono

- In generale, la valutazione dei predicati di questo genere è un problema non decidibile. In tanti casi però le espressioni simboliche possono essere semplificate, ed i risultati come quello appena visto possono essere ottenuti da un *theorem prover*

# controlli dinamici o di testing

- Progettazione (input)
  - ambiente di test
  - analisi dei risultati (output ottenuto con l'esecuzione vs output atteso)
  - debugging (vs verifica statica)

# ripetibilità

- Ripetibilità della prova
  - Ambiente definito (hardware, condizioni, ...)
  - casi di prova definiti (ingressi e comportamenti attesi)
  - procedure definite
- Strumenti
  - *Driver* componente fittizia per pilotare un modulo
  - *Stub* componente fittizia per simulare un modulo
- Registrazione e analisi dei dati di prova

# verifica di componenti e sistema

- Verifiche sulle componenti
  - approccio statico, dal *desk-check* all'*inspection*
  - approccio dinamico, con realizzazione di *driver* e *stub*
- Verifiche sul sistema
  - approccio dinamico, sul sistema completo
  - approccio dinamico, durante l'integrazione, con la realizzazione di *driver* e *stub*

# integrazione

- Metodologie diverse (già discusse)
  - big bang, bottom up, top down, sandwich
- *Stub e driver*
  - costruiti usando il grafo della *relazione di uso* dei moduli

# test sul sistema, 1

- *Facility test (test delle funzionalità)*
  - è il più intuitivo dei controlli, quello cioè che mira a controllare che ogni funzionalità del prodotto stabilita nei requisiti sia stata realizzata correttamente
- *Security test*
  - cercando di accedere a dati o a funzionalità che dovrebbero essere riservate, si controlla l'efficacia dei meccanismi di sicurezza del sistema

# test sul sistema, 2

- *Usability test*

- si valuta la facilità d'uso del prodotto da parte dell'utente finale
  - valutazione fra le più soggettive
  - prodotto + documentazione + livello di competenza dell'utenza + caratteristiche operative dell'ambiente d'uso del prodotto

- *Performance test*

- controllo mirato a valutare l'efficienza di un sistema soprattutto rispetto ai tempi di elaborazione e ai tempi di risposta
  - controllo critico per esempio per i sistemi in tempo reale, per i quali ai requisiti funzionali si aggiungono rigorosi vincoli temporali
  - I sistema viene testato a diversi livelli di carico, per valutarne le prestazioni

# test sul sistema, 3

- *Storage use test*
  - ancora un controllo legato all'efficienza di un sistema, ma mirato alla richiesta di risorse – la memoria in particolare – durante il funzionamento, e ha implicazioni sull'ambiente operativo richiesto per poter installare il sistema
- *Volume test (o load test, test di carico)*
  - il sistema è sottoposto al carico di lavoro massimo previsto dai requisiti e le sue funzionalità sono controllate in queste condizioni
    - lo scopo è sia individuare malfunzionamenti che non si presentano in condizioni normali, quali difetti nella gestione della memoria, buffer overflows, etc., sia garantire un'efficienza base anche in condizioni di massimo carico
    - le tecniche e gli strumenti come per il performance test: i due tipi di test hanno scopi molto differenti, da un lato valutare le prestazioni a vari livelli di carico, non limite, dall'altro valutare il comportamento del sistema sui valori limite

# test sul sistema, 4

- *Stress test*

- il sistema è sottoposto a carichi di lavoro superiori a quelli previsti dai requisiti o è portato in condizioni operative eccezionali – in genere sottraendogli risorse di memoria e di calcolo
  - esplicito superamento dei limiti operativi previsti dai requisiti
  - lo scopo è quello di controllare la capacità di “recovery” (recupero) del sistema dopo un fallimento

- *Configuration test*

- obiettivo: la prova del sistema in tutte le configurazioni previste
  - piattaforme di installazione diverse per sistema operativo o dispositivi hardware installati
  - insiemi di requisiti funzionali leggermente diversi

# test sul sistema, 5

- *Compatibility test*
  - obiettivo: valutare la compatibilità del sistema con altri prodotti software
    - versioni precedenti dello stesso prodotto
    - sistemi diversi, ma funzionalmente equivalenti che il prodotto deve rimpiazzare
    - altri sistemi software con i quali il prodotto deve interagire

# test di accettazione se contrattualizzato: collaudo

- Validazione del sistema
  - attività svolta dal fornitore                      alfa test o pre-collaudo
  - attività svolta dal committente                      beta test o collaudo
  - su casi di prova definiti nel contratto
- Valore contrattuale
  - il collaudo è un'attività ufficiale
  - conclusione della commessa (a meno di servizi e garanzie)
  - al collaudo segue il rilascio del sistema