Appunti Progettazione di Software Sicuro

- Prodotto: cosa realizziamo
- Processo: come lo realizziamo

Caratteristiche prodotto SW:

- Intangibile: difficile descrivere e valutare
- Malleabile: trasformabile
- Human intensive: no manifattura classica

Qualità processo SW:

- Produttività: misura la velocità di consegna del SW
- Tempestività: capacità di rispettare i termini di consegna
- Trasparenza: capacità di capire lo stato attuale

Qualità SW:

- Esterne (black box view):
  - Correttezza: rispetta le specifiche funzionali di progetto
  - Affidabilità: si comporta come previsto
  - Efficienza: se usa sapientemente le risorse
  - Usabilità: facilità d'uso da parte dell'utente
  - Portabilità: se può funzionare su più piattaforme
  - Interoperabilità: coesiste e coopera con altri sistemi
  - Robustezza: si comporta in modo ragionevole anche in condizioni non previste
- Interne (white box view):
  - Riusabilità: se è riutilizzabile in altri sistemi
  - Verificabilità: permette di dimostrare a posteriori la correttezza o altre caratteristiche
  - Facilità di manutenzione: facile realizzare adattamenti/correzioni. Modifiche correttive / perfettive / adattive

Sicurezza: significato relativo al contesto / stabilire una policy di sicurezza per proteggersi da chi / cosa

Qualità del SW sicuro:

- Prevenzione: anticipare possibili attacchi
- Tracciabilità: possibilità di stabilire causa/effetto tra due elementi diversi
- Auditing: controllo del sistema sulla base di politiche e procedure stabilite a priori per sapere se qualcuno ha fatto il furbo (deterrente)
- Monitoraggio: Auditing in tempo reale (attenzione ai falsi allarmi). Approcci semplici (identificare un attacco) o complessi (monitoraggio codice mediante asserzioni)
- Privatezza: possibilità di stabilire se, come, quando e a chi riguardano le informazioni
- Confidenzialità: assicurarsi che certi servizi o info siano accessibili solo ad utenti autorizzati
- Sicurezza a più livelli: più livelli di segretezza delle informazioni
- Anonimato: possibilità di mantenere segreti alcuni dati
- Autenticazione: proprietà di conoscere l'identità di chi accede a un servizio (proprietà critica)
- Integrità: non essere modificato una volta creato

Processo di produzione: sequenza di attività per: progettare, realizzare, consegnare, modificare SW

Ciclo di vita di un sw: insieme di stadi in cui si può trovare il sistema

Modelli di processo: organizzazione delle diverse parti dello sviluppo

Fasi di sviluppo:

- Studio fattibilità: produzione di un documento (Feasibility Study Document [FSD] che valuti costi e benefici). Nel documento:
    - Definizione del problema
    - Possibili soluzioni
    - Per ogni soluzione, stima dei benefici, costi, risorse, tempi
- Specifica (analisi dei requisiti): produce Requirements Analysis and Specification Document (RASD). Occorre:
    - Capire l'interfaccia tra l'applicazione e l'esterno
    - Indentificare il target di utenza e comprendere le aspettative
    - Stabilire le qualità
    - Il RASD:
        - Permette al cliente di verificare le caratteristiche specificate
        - Consente al progettista di sviluppare il SW (quindi comprensibile, preciso ecc.)
        - Processo incrementale che richiede interazione con il cliente, rispettando le 5W inglesi.
- Progettazione: produzione di un documento con l'architettura del SW (globale e moduli). Fase di design -> Design Document. Obiettivi della progettazione: separare le responsabilità. Nel documento:
    - Definizione dei moduli
    - Relazioni ed interazioni tra loro
- Implementazione e test moduli: per ogni modulo fornire codifica, documentazione, test effettuati
- Integrazione e test: assemblare i moduli insieme
- Consegna: agli utenti finali, prima beta tester e poi distribuzione
- Manutenzione: correzione + evoluzione del sw (correttiva/adattiva/perfettiva)

Modelli di processo / cicli di vita:

- A cascata: l'output della fase precedente diventa l'input della fase successiva
    - Bisogna finire una fase per iniziarne un'altra
    - Modello black box
    - Non tiene conto che i requisiti possano cambiare
- Evolutivi (processi incrementali): tengono conto dei feedback (validazione) adattandosi ai cambiamenti (consentono il riciclo). Es:
    - Prototipazione: modello approssimato per fornire feedback sulle caratteristiche del sistema (fare leva sulla riusabilità delle componenti sviluppate). Massimizzare la modularizzazione.
    - A fasi di release: si parte da sottoinsiemi critici sui quali chiedere feedback del cliente
    - A spirale: processo ciclico tra:
        - Analisi dei rischi
        - Sviluppo
        - Testing
        - Revisione
        - Ad ogni ciclo il costo aumenta

Analisi di sicurezza: bilancio tra sicurezza e funzionalità. Modello ottimale -> spirale. Nell'analisi si usano artifatti (oggetti) e stakeholders (persone).

Individuare ciò che va protetto, da chi, per quanto, a che prezzo

Analisi dei rischi: identificare i possibili rischi e prevenirli/affrontarli (classificare i rischi in base alla gravità)

Specifica: deve includere soluzioni ai rischi trovati

- Dettagliata
- Formale (non ambigua)
- Eseguibile (feedback immediato)

Design (security plan in questa fase):

- Verificare comunicazione tra componenti
- Utenti e ruoli
- Relazioni di fiducia tra i componenti

Implementazione: fase critica. L'ingegnere della sicurezza deve individuare le linee guida del codice e controllarlo (code audit)

Testing:

- Funzionale: vedere se il sistema funziona a dovere in situazioni normali o critiche
- Di sicurezza: provare a far casini
- Code coverage: vedere se ci sono parti di codice mai eseguite che lasciano bachi sfruttabili

Common criteria: un insieme di classi e comportamenti che combinati tra l'oro definiscono profili di protezione per ogni prodotto IT.

Common evaluation methodology: modalità di valutazione di un sistema in base ai common criteria.

Processo di valutazione (es):

- Wall Street produce un profilo di protezione (secondo i common criteria) per il firewall
- Il profilo viene valutato in base alla common evaluation methodology per garantire completezza e consistenza
- Ottenuta la valutazione il profilo viene pubblicato (target di sicurezza)
- Un vendor ne realizza la sua versione (target di valutazione)
- Il target di valutazione viene inviato ad un istituto accreditato per la valutazione rispetto al target di sicurezza
- L'istituto applica la common evalutation methodology per vedere se il target di valutazione soddisfa quello di sicurezza
- Esito positivo: la documentazione di testing viene inviata al NVLAP (National Voluntary Laboratory Accreditation Program) per controllarne la correttezza
- Approvati gli atti il prodotto ottiene la certificazione di prodotto valutato in base ai common criteria

Limiti common criteria: definiscono il cosa e non il come difendersi

Ciclo di vulnerabilità:

- Viene scoperta una falla
- I cattivi la sfruttano
- I buoni cercano una soluzione, la sviluppano e distribuiscono una patch
- I media possono riferire della falla se era grave
- La patch distribuita viene installata
- I tecnici della sicurezza cercano vulnerabilità simili

Attacco: atto malevolo contro un sistema. Si distinguono:

- Goal: danno causato
- Sottogoal: obiettivi intermedi necessari al raggiungimento del goal (esempio permessi di admin)
- Attività: attività svolte per raggiungere goal o sottogoal
- Eventi: occorrenza di attacchi causati dalle attività
- Conseguenze: conseguenza del verificarsi di un evento
- Impatto: effetti di business (es: infangare la reputazione di un'azienda)

Tipi di attacco:

- In fase di progettazione
- In fase di implementazione (mentre si sta scrivendo il sw)
- In fase di operazione (mentre il sw è in produzione)

Attacchi:

- Man-in-the-middle: l'attaccante intercetta una trasmissione e si finge uno dei due host
    - Difesa: usare encryprion, session checksums e shared secrets
- Race condition attack: l'output di un processo dipende in modo inatteso dalla sequenza temporale di altri eventi
    - Difesa: distinguere tra operazioni atomiche e non
- Replay attack: l'attaccante cattura parte di una conversazione e la riproduce a scopo maligno
    - Difesa: come il man in the middle
- Sniffer attack: memorizza tutto il traffico di una rete locale
    - Difesa: attenta configurazione di rete, uso di switch, massimizzare crittografia
- Hijacking: prendere il controllo di una connessione già stabilita
    - Difesa: crittografia e logging
- Session killing attack: uccidere prematuramente una sessione
    - Difesa: essere in grado di ristabilire una connessione

Attacchi in fase di implementazione:

- Buffer overflow: l'attaccante causa overflow in input avendo accesso a porzioni di memoria non normalmente accessibili
    - Difesa: uso di linguaggi come java, evitare stringhe indeterminate
- Backdoor: punti di accesso secondari al sw
    - Difesa: controllarne l'assenza
- Parsing error: l'applicazione accetta input senza controllarne la forma
    - Difesa: riutilizzare codice già controllato, testare i comandi in input

Attacchi in fase di oprazione:

- Denial-of-service: sovraccaricare il sistema che negherà il servizio a richieste legittime
    - Difesa: progettare una moderata richiesta di risorse, implementare un meccanismo per monitoraggio e verifica di caricamento eccessivo
- Default accounts attack: usare user e pass di default per acceder ad un sistema
    - Difesa: rimuovere gli account di default
- Password cracking: crackare le password
    - Difesa: scegliere password intelligenti o usare tool per rendere robuste le password

Architettura sicura: insieme dei principi e delle decisione per la progettazione di sistemi sicuri:

- Grado di sicurezza sufficiente
- È un framework per la progettazione sicura
- Definita mediante insieme di documenti
- Riusabile

Architettura sicura != policy di sicurezza. Policy:

- Stabilisce le regole su chi ha accesso a cosa
- Stabilisce le procedure di testing per certificare il sw
- Va stabilita a priori ed è la linea guida per l'architettura sicura di un'applicazione

L'architettura deve definire:

- Organizzazione del programma
- Strategie di cambiamenti
- Decisioni su acquisto vs sviluppo
- Principali strutture dati
- Algoritmi chiave
- Oggetti principali
- Funzionalità generica
- Processamento dell'errore
    - Correttivo
    - Determinativo
- Robustezza attiva o passiva
- Tolleranza ai guasti (fault tollerance)

Principi base di un'architettura sicura:

- Iniziare a farsi delle domande:
    - Su vulnerabilità: cosa può andare storto, cosa voglio proteggere
    - Sulle Risorse: l'architettura è sicura? È già collaudata?
    - Sul sw: dove si trova rispetto alla chain of trust? Qualcuno fa affidamento di lui per la sicurezza?
    - Sui goals: che impatto avrebbe un attacco di sicurezza?
- Decidere il livello di sicurezza sufficiente:
    - Just secure enough
- Identificare le assunzioni che diamo per scontatamente buone
- Progettare con in mente il nemico
- Conoscere e rispettare la chain of trust

    - Non invocare programmi non fidati, validare l'input, ripulire l'output0
- Definire privilegi adeguati
    - Principio del minimo privilegio
- Testare le possibili azioni contro la policy
- Suddividere la progettazione in moduli
- Non offuscare le informazioni
    - La segretezza non deve essere l'unica difesa
- Mantenere in memoria lo stato minimale (lo stretto indispensabile)
- Garantire un adeguato livello di fault tollerance:
    - Resistance: capacità di impedire un attacco
    - Recognition: capacità di riconoscere attachi
    - Recovery: capacità di ripristinare i servizi dopo l'attacco
- Pianificare l'error-handling
- Applicare il principio di graceful degradation
    - Il sistema deve continuare a lavorare con funzionalità ridotte
- Garantire che un fallimento lasci il sistema in una configurazione sicura
- Applicare misure di sicurezza adeguate all'utente
    - Accettabilità psicologica
- Responsabilizzare l'utente nelle sue azioni
    - Con autenticazione e ogni utente deve avere chiare le proprie responsabilità
- Controllare e limitare il consumo di risorse
    - Usare tecniche di graceful degradation
- Garantire la possibilità di ricostruire gli eventi
    - Principio di auditability
- Pianificare livelli multipli di difesa
- Non considerare un'applicazione come monolitica
    - Suddividere in moduli il più semplici possibili
- Riusare sw sicuro certificato
- Utilizzare tecniche di ingegneria standard
- Progettare la sicurezza sin dalle fasi iniziali del ciclo di vita

Criteri di scelta di tecnologie sicure:

- Linguaggio di programmazione:
    - Efficienza: in fatto di performance, meglio sacrificarne un po' per la sicurezza
    - Portabilità
    - Sicurezza: non tutti sono sicuri (c non è affidabile, java lo è di più a spese dell'efficienza)
        - Java: no puntatori, controllo accesso ai buffer, controllo dei tipi, eccezioni, sandbox, librerie di sicurezza
- Piattaforma distribuita
    - Corba
        - Livello1: Sicurezza gestiita solo dall'orb
        - Livello2: Servizi avanzati di sicurezza
    - Dcom
        - Autenticazione, integrità e segretezza sono imposte dall'autentication level (fino a 7), ogni livello è costruito su quello inferiore.
        - Servizio di impersonificazione

    - Ejb e rmi

        - Comunicazione attraverso RMI, controllo accessi + cifratura offerti dal server
        - Sicurezza Rmi debole
- Sistema operativo
    - Separazione tra kernel space e user space
    - Separazione tra processi
- Tecnologie di autenticazione
    - Autenticazione in base all'host
    - Autenticazione in base a oggetti fisici
        - Attenzioni ai furti
    - Autenticazione biometrica

Java sandbox

- Applet: codice scaricato da Internet, da assumersi malizioso, eseguito nella sandbox. Policy in java1:
    - Non può accedere al filesystem
    - Può connettersi solo alla sorgente
    - Non può eseguire programmi locali
    - Marcato come untrusted
    - Non può estendere le classi base
- Applicazioni: programmi in locale
    - Accesso illimitato, codificano le loro policy

Sicurezza in java

- Sicurezza dei tipi
- Verifica del bytecode (java compilato): viene accettato solo codice java legittimo
    - Attenzione che il bytecode sia prodotto dal compilatore
    - Il bytecode non deve eseguire conversioni illegali
    - Non modificare puntatori
    - Non viola restrizione di accesso
    - Usa correttamente gli oggetti
    - Chiama i metodi correttamente
- Controllo runtime delle classi create (lo fa la jvm)
- Accesso mediato alle risorse (mediante la jvm)

Efficienza: verifica del bytecode staticamente, a runtime la jvm può assumere che alcuni controlli siano già stati fatti

Sicurezza gestita da 2 classi:

- SecurityManager: definisce i metodi di controllo dei permessi di sicurezza chiamati dal sistema
    - Creo la mia policy creando una sottoclasse di questa
- ClassLoader: carica effettivamente le classi
    - Dato il nome della classe trova la sua definizione
    - Prima cerca tra quelle standard
    - Ogni classe mantiene un riferimento a chi la creata

Jdk1.1: applet firmate, permessi come le applicazioni

Java2: controllo sugli accessi basato su:

- Security policies
- Permessi

- Meno restrittivo del tutto o niente di jdk 1, non è necessario scrivere codici particolari per modificare la SecurePolicy, separazione tra espressione della policy e la sua applicazione
- La classe CodeSource rappresenta da dove proviene il codice (url della classe, array di certificati). È immutabile.
- La classe Permission rappresenta la gerarchia di permessi
- ProtectionDomain Class: un dominio di protezione viene creato da un CodeSource e un insieme di permission: definisce l'insieme di permessi dati alla classe; ogni classe appartiene ad una istanza di protectionDomain (impostata alla creazione)
- Policy class: interfaccia per definire politiche di sicurezza particolari
    - Dato un codeSource restituisce un permissionCollection

Quando una classe C1 chiede di caricare una nuova classe C2:

1. Il classLoader di C1 cerca il C2.class, lo carica e chiama il bytecode verifier
2. Il codeSource di C2 viene determinato
3. L'oggetto Policy del classLoader restituisce le permission dato il codeSource di C2
4. Se un protectionDomain già esistente ha lo stesso sourceCode e permissions viene riusato altrimenti viene creato un nuovo protectionDomain
5. La classe viene istanziata, continuano gli altri controlli

Se una classe C invoca il metodo M:

- La classe M richiede il permesso P
- Questo attiva un accessController che controlla il chiamante di M, cioè C
    - Se il protectionDomain di C contiene P allora ok

Linguaggi per la specifica:

- Informali
    - Linguaggio naturale
- Semi formali (es: uml)
    - Possibilmente grafici
- Formali
    - Formalismi operazionali: definiscono il sistema descrivendone il comportamento (più intuitivo)
    - Formalismi dichiarativi: definiscono il sistema dichiarando le proprietà che esso deve avere (non ambiguo)

UML: Linguaggio di Modellazione Unificato. Semi formale, grafico, complesso di viste, sintassi mediante meta-modello, semantica in prosa

Specifica dei requisiti: descrizione non ambigua dei requisiti, cosa un sw deve fare e come deve farlo. Accordo tra il produttore sw e il suo committente. Requisiti:

- Funzionali
- Non funzionali
- Del processo e manutenzione

Sviluppare una specifica richiede un'analisi cooperativa e l'impiego di linguaggi formali

Qualità delle specifiche:

- Chiarezza
- Non ambiguità
- Consistenza: non deve contenere contraddizioni

- Completezza
  - Interna: definire ogni concetto nuovo
  - Esterna: completa rispetto ai requisiti
- Incrementalità: sviluppata in più passi successivi
- Compensibilità

Macchina a stati finiti FSM: notazione formale che permette la rappresentazione astratta del comportamento di un sistema.

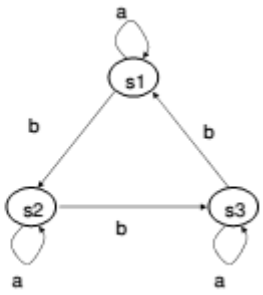
- I nodi rappresentano gli stati del sistema
- Gli archi i passaggi di stato (transizioni)

Definizione FSM: tupla (S, I, δ);

- S: insieme degli stati
- I: insieme degli input
- δ: funzione di transizione (SxI)

Esempio:

S = {s1, s2, s3}
I = {a, b}
δ= { (s1,a,s1), (s1,b,s2),
(s2,a,s2), (s2,b,s3),
(s3,a,s3), (s3,b,s1)}

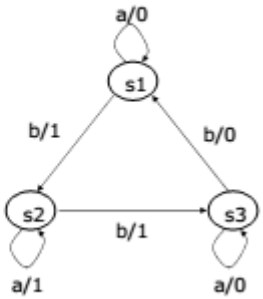


FSM con output:

- di Mealy: output su ciascuna transizione (S, I, O, [δ, λ] T):
  - S: stati
  - I input
  - O: output
  - δ: funzione di transizione (SxI=S)
  - λ: funzione di output (SxI=O)
    - T: insieme finito di transizioni (s, i, o, s')
      - s: stato sorgente
      - i: input
      - o: output
      - s': stato destinatario

    ogni arco è etichettato con i/o

S = {s1, s2, s3}
I = {a, b}
O = {0, 1}
T = {(s1,a,0,s1),
(s1,b,1,s2),
(s2,a,1,s2),
(s2,b,1,s3),
(s3,a,0,s3),
(s3,b,0,s1)}



- di Moore: output su ciascuno stato (S, I, O, δ, λ, F):
  - S: stati
  - I: input
  - O: output
  - δ: funzione di transizione
  - λ: funzione di output
  - F: insieme di stati finali

Limiti delle fsm:

- È possibile rappresentare solo un numero finito di stati

- Limite di composizionalità (crescita esponenziale quando le voglio combinare). Le soluzioni (ottenibili mediante estensioni) invece permettono:
    - Composizione sequenziale
    - Composizione parallela
    - Modularità

Fsm estese: introducono le variabili. Tupla (S, I, O, V, T):

- S: insieme di stati
- I: insieme degli input
- O: insieme degli output
- V: insieme delle variabili
- T: insieme delle transizioni (s, i, o, g, a, s'):
    - S: stato sorgente
    - I: input
    - O: output
    - G: guardia (predicato sulle variabili)
    - A: azione (assegnamento ad una variabile)
    - S': stato target

Stato globale: coppia (s, σ):

- S un suo stato
- σ una valutazione delle variabili ("screenshot delle variabili")

Transizione globale: tupla ((s, σ) i, o, (s, σ,)) -> da uno stato globale, con dell'I/O, ad un altro

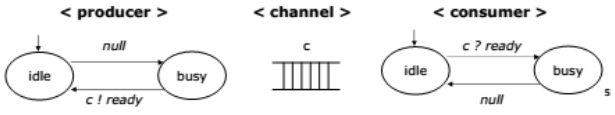
Grafo di raggiungibilità: grafo direzionato in cui i nodi sono gli stati globali e gli archi le transizioni globali (è anch'esso una fsm)

Macchine di comunicazione (Communicating finite state machines [CFSM]): coppia (C, P) dove:

- C: numero finito di canali
- P: numero finito di processi (tupla (S, I, O, T)):
    - S: stati
    - I: Input
    - O: Output
    - T: Transizioni (s, null, s') (s, c?i, s') (s, c!o, s')

Esempio

- C = {c}
- P = {producer, consumer}
  - producer = (S, I, O, T)
    - S = {idle, busy}, I = {}, O={ready}
    - T = {(idle,null,busy), (busy,c!ready,idle)}
  - consumer = (S', I', O', T')
    - S' = {idle, busy}, I' = {ready}, O' ={}
    - T' = {(idle, c?ready,busy), (busy,null,idle)}



Stato globale: coppia (θ, σ) con:

- Θ=<s1, s2, ecc> stati dei processi p1, p2 ecc.
- σ è una valutazione su C

Esempio

- (<idle, idle>, c=empty)
- (<idle, idle>, c=ready)
- (<busy, idle>, c=ready)

sono stati globali

Transizione globale: coppia ((θ, σ), (θ', σ'))

Grafo di raggiungibilità: grafo direzionato in cui i nodi sono gli stati globali e gli archi sono le transizioni globali.

CFSM estesa: coppia (C, P):

- C: canali
- P: processi tupla (S, I, O, V, T)
    - S: stati
    - I: input
    - O: output
    - V: variabili
    - T: transizioni nella forma:
        - (s, g, a, s'), (s, g, c?i, s'), (s, g, c?i, s')
        - g: guardia (controllo)
        - a: azione (assegnamento)

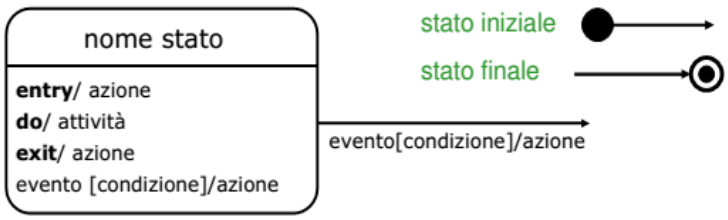
CFSM temporale: è una CFSM estesa in cui le transizioni dei processi possono contenere info circa l'intervallo di tempo entro cui l'azione può avvenire una volta entrati nello stato

- Transizione nella forma (s, g, a|I/O, s', [t1, t2])
    - [t1/t2] indica l'intervallo di tempo entro cui il processo può eseguire un'operazione interna A o un'operazione di I/O (c?i, c!o) DOPO che è entrato in S e PRIMA di raggiungere S'

Macchine di stato in UML: consentono modellazione dinamica, descrive il comportamento di un oggetto di una classe. I nodi rappresentano gli stati e gli archi le transizioni.

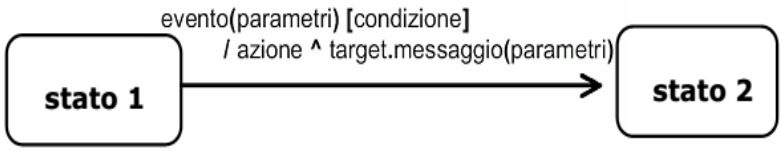
Stato: condizione in cui si trova l'oggetto

- Azioni: operazioni atomiche:
    - Entry: azione eseguita entrando nello stato
    - Exit: operazione eseguita uscendo dallo stato
- Attività: operazioni non atomiche



Transizione: passaggio di stato

Etichettata con:



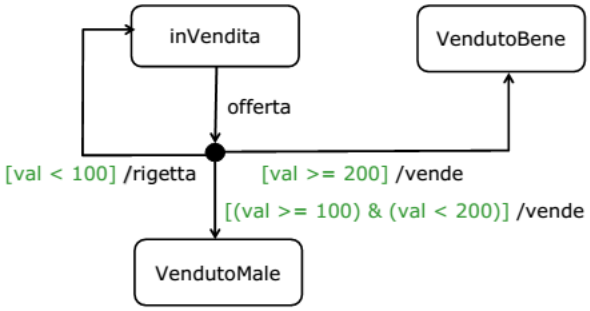
- Gli eventi che comportano il cambiamento di stato (e le condizioni sotto cui ha effetto)
- Le azioni che l'oggetto esegue PRIMA di cambiare stato

Una transizione scatta quando occorre un evento (interno o esterno) e l'eventuale guardia sulla transizione è vera

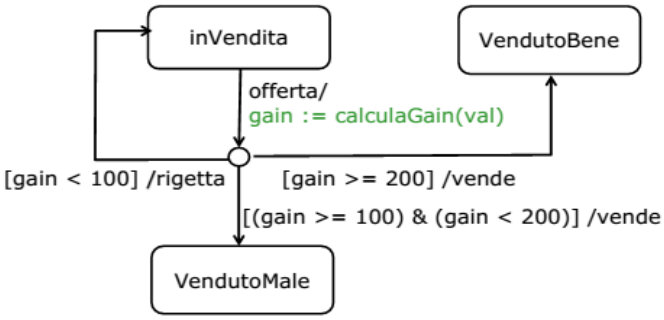
Tipi di evento:

- Interazione con altri oggetti
    - Call event: un oggetto invoca un'operazione propria o di un altro
    - Single event: un oggetto riceve un oggetto segnale da un altro oggetto
- Occorrenza di istanti di tempo
    - Time event: intervallo temporale
- Cambiamento di valore
    - Change event: es: when(bilancio<0)

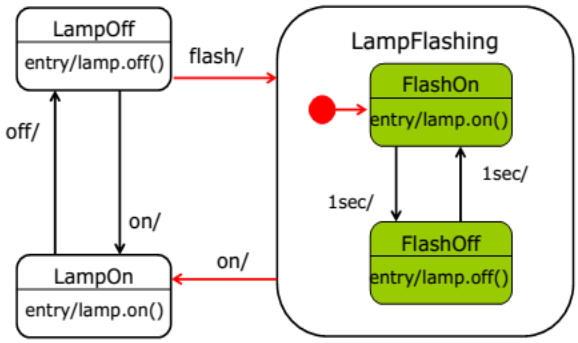
Junction pseudostate: punti di decisione



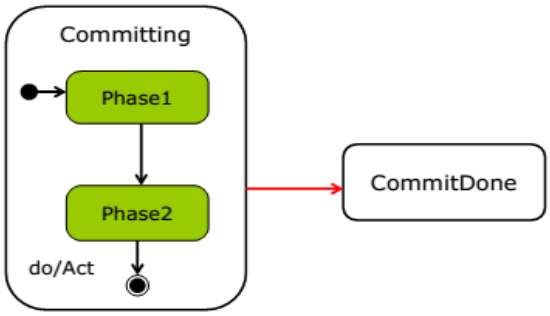
Choice pseudostate: le guardie vengono valutate nell'istante in cui il punto di decisione è raggiunto



Stati composti sequenziali: uno stato può contenere una macchina di stato



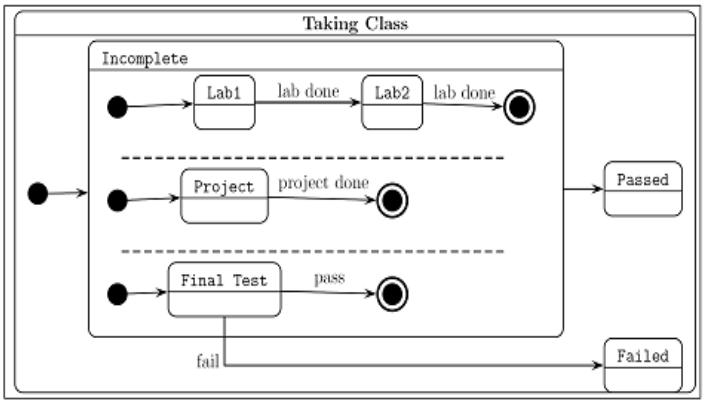
Transizioni di completamento: generato quando una macchina annidata ha finito la sua attività interna



Se due transizioni sono attivate dallo stesso evento hanno precedenza le transizioni più interne

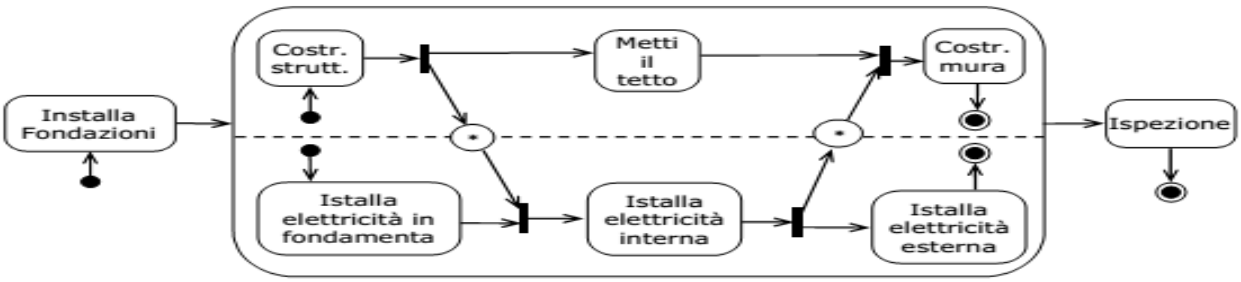
Stati composti paralleli: uno stato che contiene più regioni ortogonali. Es:

Un corso richiede, per essere superato, la frequenza di 2 laboratori tra loro propedeutici, un progetto, ed una prova finale da superare con successo. Se la prova finale fallisce, il corso non è superato



Come si sincronizzano le diverse regioni?

- Condizioni o variabili condivise: una guardia è un predicato sullo stato di un'altra regione
- Eventi: le azioni di un diagramma diventano eventi a cui altri diagrammi possono reagire
- Stati di sincronizzazione: non si passa da una sbarra finchè tutte le condizioni non sono rispettate



Design by contract: un contratto è un accordo che lega cliente e fornitore. Esso:

- Lega le due parti
- È esplicito
- Specifica gli obblighi e i benefici di entrambi
- Gli obblighi di una parte diventano i benefici per l'altra parte
- Non contiene clausole nascoste

(Es: linguaggio Eiffel, contiene il linguaggio per scrivere contratti)

Nel sw: oggetto del contratto= il sw (insieme di classi)

- Cliente: chi usa il programma (classe)
- Fornitore: chi scrive il programma (classe)

Pre e post condizioni. In un contratto posso definire:

- Cosa ogni metodo chiede = PRECONDIZIONI (obbligo per il CLIENTE)
- Cosa ogni metodo fornisce = POSTCONDIZIONI (obbligo per il FORNITORE)

(Es: operazione di inserimento in un array: precondizione= array non pieno, post condizione= l'array ha un elemento in più)

Precondizioni:

- Deboli: tutte le complicazioni saranno gestite dal metodo
- Forti: cerco di risolvere a monte possibili guai. Meglio avere metodi semplici che metodi che cerchino di risolvere ogni magagna

Cliente e fornitore controllano le precondizioni e le post condizioni in maniera separata. Il cliente controlla le PRE il fornitore controlla le POST. Entrambi assumono che l'altro faccia il suo lavoro.

Proprietà interne ad una classe: INVARIANTI: è vera dopo la creazione dell'oggetto e dopo ogni operazione. Es: per la dimensione di un array 0<=n_elementi<=size. Conta come ulteriore precondizione.

I contratti possono essere inseriti anche prima dell'implementazione del metodo. Dai contratti si possono estrapolare pre, post e invarianti.

Eccezioni: si sollevano quando il contratto è violato. Equivale alla manifestazione di un difetto cioè un bug.

Vantaggi del DbC:

- Il processo di sviluppo diventa più concentrato
- Scrive nelle specifiche
- Getta le basi per del software riusabile
- Aiuta a comprendere le eccezioni precisando cosa è normale e cosa no
- La documentazione dell'interfaccia è sempre up to date
- Documentazione generata automaticamente
- Gli errori capitano vicini alla loro fonte, li si trova facilmente e in fretta
- Guida per la generazione di casi di test black-box

Jml:

- Annotazioni come commenti con questa formattazione: /*@...@*/
- Condizioni scritte come espressione booleane di java
- Alcune parole chiave:
    - Precondizioni ai metodi: requires <espressione booleana>

- Postcondizioni ai metodi: ensures <espressione booleana>
- Invarianti di classe: invariant <espressione booleana>

Es: questo Richede che l'ammontare sia positivo ma non garantisce nulla:
/*@ requires amount >= 0; ensures true;@*/
public int debit(int amount){...}

Comandi jml:

Espressione	Significato
`a ==> b`	a implica b
`a <== b`	a consegue da b (cioè b implica a)
`a <==> b`	a se e solo se b
`a <=!=> b`	non (a se e solo se b)
`\old(E)`	valore di E prima della chiamata
`\result`	risultato della chiamata del metodo

* es: sotto mi restituisce il risultato se e solo se j è minore di n:
//@ ensures \result <==> j < n;
boolean minore(int j, int n) {return j < n;}

Quantificatori:

- Universali: \forall → per ogni elemento
- Esistenziali: \exists → esiste un elemento
- Generali: \sum, \product, \min, \max
- Numerici: \num_of

Sintassi:
<quantificatore><tipo><variabile>;<range predicate>;<epressione>

Es: tutti gli studenti contenuti nella collection juniors devono avere "advisor" (cioè getAdvisor() != null):
(\forall Students s; juniors.contains(s); s.getAdvisor()!=null)

Altro es: Metodo per trovare il minimo di un array:
Public static int find_min(int a[])
Pre: array non null con almeno un elemento: //@ requires a!=null && a.lenght >=1
Post: l'elemento minore è effettivamente tale (ed appartiene all'array): //@ensures (\forall int i; 0<=i && i<a.lenght; result <=a[i]) && (\exist int i; 0<=i && i<=a.lenght; \result ==a[i])

Operatore non_null: server per quelle invarianti, pre e post che non devono mai essere null. Es:
private /@non_null@*/ File[] files;*

\old (variabile): per accedere al valore di una variabile prima che il metodo fosse stato eseguito. Es:
public class Contatore {
integer n;
//@ ensures n == \old(n)+1
public void incrementa() { n ++; }

Pure: per dichiarare metodi usabili in Jml come funzioni matematiche. Es:

```
static /*@pure@*/ int abs (int x){
if (x>=0) return x; else return –x;
}
```

Spec_public: specifica che una variabile può essere usata in una specifica pubblica di Jml. Es:

```
public class Bag{
private /*@ spec public @*/ int n;
//@ requires n > 0;
public int extractMin(){ ... }
...
}
```

Assert: per richiedere che una condizione sia verificata ad un certo punto del codice. Es:

```
if (i <= 0 || j < 0) { ...
} else if (j < 5){
//@ assert i > 0 && 0 <= j && j < 5;
...}
else {
//@ assert i > 0 && j > 5;
…}
```

Errori tipici della programmazione: errori di implementazione (sbagli nello scrivere il SW). Soluzioni:

- Essere informati (seguire le vulnerabilità, leggere libri, guardare altro codice)
- Trattare l'input con cautela (ripulire I dati in ingresso, attenzione alla modifica dei file di configurazione e dei cookie, non fidarsi di url e html, controllare sql)
- Trattare I dati con cautela (attenzione all'inizializzazione, ai nomi, a dove li memorizzi)
- Riusare codice affidabile preesistente
- Revisionare il codice (magari farlo fare ad altri o usando strumenti automatici)
- Pensa ad una facile manutenzione (usa degli standard, rimuovi codice obsolete, testa tutto)

Cosa NON fare:

- Non usare nomi di file relative
- Non riferirti 2 volte allo stesso file con lo stesso nome
- Non dare per scontato che l'utente sia affidabile
- In caso di errore termina in modo controllato
- Non assumere sempre successo nelle tue operazioni
- Non fare troppo affidamento ai numeri pseudo casuali
- Non autenticare con cose "non autentiche" (mac, ip ecc)
- Non usare file temporanei accessibili
- No dati sensibili non protetti
- Non mostrare le pass a schermo
- Non spedire pass via mail (o altri dati sensibili)
- Non mettere user e pass nel codice
- No pass in chiaro
- No pass su canali insicuri
- Non fare decisioni sui diritti di accesso in base a parametri passati da linea di commando o da variabili d'ambiente

- Evita sw altrui per le parti critiche

Tipo: insieme di valori omogenei + operazioni possibili. Servono ad organizzare I concetti e assicurarsi che I bit in memoria vengano correttamente interpretati.

Errori tipici:

- Confondere dati con programmi
- Confondere dati semplici
- Errori semantici

Problema del c con I tipi (nonostante il c sia flessibile e veloce)

- Accesso non valido alla memoria
    - Violazione spaziale della memoria: out of bound
    - Violazione temporale della memoria: dangling pointers (zona di memoria liberata per essere riutilizzata. Soluzione: malloc, garbage collector)
- Deferenziazione del null (accedo ad una cella puntata da un puntatore nullo)
- Type cast non controllato (conversione non controllata da un tipo all'altro)
- Pointer arithmetic (puntare a zone di memoria con tipi diversi)

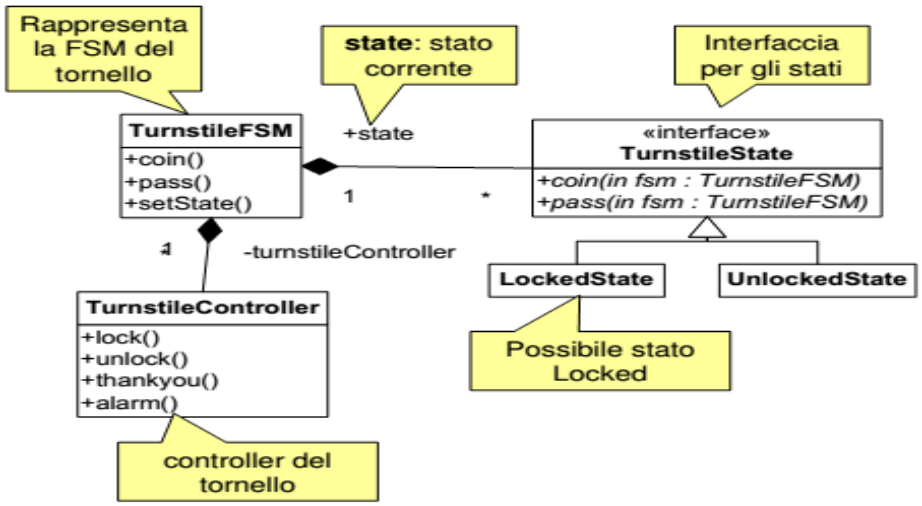
Posso rendere il c più safe con tool come purify, safec o ccured, usando librerie o usare varianti del c come cyclone o vault

Quando fare il type checking:

- Run-time (rallenta esecuzione)
- Compile time (flessibilità limitata) (come fa java. Dove è insicuro introduce anche un controllo run-time)

Svantaggi nell'usare java: prestazioni inferiori, impiego maggiore di memoria, annotazione dei tipi, il porting da c (perché java è simile ma non identico).

Fsm in java: STATE PATTERN: permette di cambiare il comportamento di un oggetto quando cambia il suo stato interno. Si usa quando il suo comportamento dipende dal suo stato. Si definisce una classe per ogni stato con un'interfaccia comune che reagisce ad ogni possibile input. Gli oggetti di stato possono richiamare azioni sull'oggetto e decidono il prossimo stato (transizioni). Es: tornello:



Delega gestine degli eventi a questo (gerarchia di stati)

Delega la gestione delle azioni a questo

Gerarchia di stati: ogni sottoclasse della gerarchia di classi implementa (nell'esempio) turnstileState e definisce un oggetto costante che rappresenta quello stato. Es:

```
class LockedState implements TurnsileState{
public static TurnstileState LOCKEDSTATE = new LockedState();
Classi del tornello:
/* rappresenta lo stato del tornello*/
interface TurnstileState {
void coin(TurnstileFSM t);
void pass(TurnstileFSM t);
}
/* rappresenta lo stato LockedState */
class LockedState implements TurnstileState {
// dice cosa accade quando il tornello è locked
// e accade l'evento coin
public void coin(TurnstileFSM t) {…}
// idem quando pass
public void pass(TurnstileFSM t) {…}
}
/* rappresenta lo stato UnLockedState */
class UnlockedState implements TurnstileState {…}
```

Il cambiamento di stato e l'azione quando accade l'evento E è delegato al metodo E dello stato corrente. Es:

```
evento coin mentre è in locked state:
class LockedState …
public void coin(TurnstileFSM t) {
t.setState(UnlockedState.UNLOCKEDSTATE);
t.unlock();
}
```

Per azionare la FSM la devo passare come parametro

Esistono tool automatici che trasformano sfm in codice java.

Riassumendo:

- la FSM è rappresentata da una classe che delega le azioni ad un controller e le transizioni di stato a una gerarchia di stati
- ogni possibile stato è una sottoclasse di una interfaccia e implementa un metodo per ogni evento (input) possibile
- nel metodo del singolo stato c'è il codice che gestisce il cambio di stato e l'azione (output) quando la FSM si trova in quello stato

Testing: eseguire un programma con alcuni casi di test per vedere se si comporta correttamente.

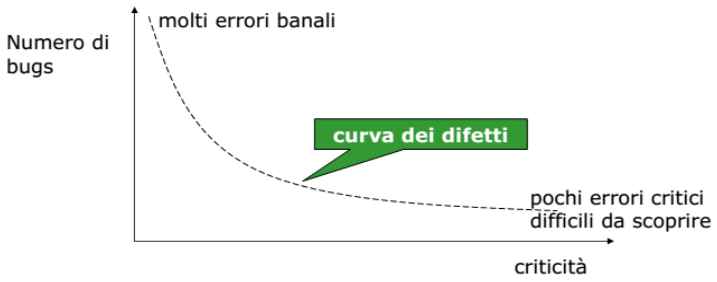
Il testing è efficace a trovare bug ma è inadeguato a provare l'assenza di bug.

Linee guida: il test dovrebbe:

- Essere automatizzato: uso di strumenti automatici per scrivere i test, eseguirli e analizzarne la correttezza
- Riguardare ogni fase di sviluppo
- Essere esteso a tutti i componenti di un sistema

- Essere pianificato
- Seguire standard ove possibile

Ipotesi della curva dei difetti: i difetti più critici sono di meno e più difficili da scoprire: scopro all'inizio molti errori banali ma a lungo andre faccio fatica a trovare quelli critici e difficili. Altri test (analisi statica) più costosi cercano di scoprire gli errori più critici prima. Attenzione: non è sempre vera questa distribuzione.



Bisogna tenere in considerazione quanto costa fare il testing e quanto costa un difetto (costo del danno che provoca * probabilità che accada).

Definizioni inerenti al testing:

- Malfunzionamento o failure o guasto: funzionamento non corretto del programma
- Difetto o anomalia o fault o bug: elemento del programma sorgente non corrispondente alle aspettative. Uno o più difetti possono causare malfunzionamenti
- Errore: deviazione del sw prodotto da quello ideale effettivamente richiesto (causa umana). Porta a difetti e quindi a malfunzionamenti

  Quando un programmatore commente un ERRORE il programma ha un DIFETTO che può causare un MALFUNZIONAMENTO

Testing vs debugging:

con il primo faccio andare il programma con dei casi di test, con il secondo correggo i difetti e trovo gli errori.

Scopi del testing:

- evidenziare bug per trovare errori
- valutare l'affidabilità di un sw

Livelli di granularità: test di:

- Accettazione: il comportamento del sw è confrontato con i requisiti dell'utente finale
- Conformità: il comportamento di tutto il sw è confrontato con le specifiche dei requisiti
- Sistema: controlla il comportamento del sistema hw + sw come monolitico
- Integrazione: controllo sulla cooperazione dei moduli (o unità)
  - Top-down: inizia dall'alto (servono stub)
  - Bottom-up: dal basso (servono drivers)
  - Big-bang
- Unità: test di comportamento delle singole unità (moduli). L'unità è una class, il test testa i singoli metodi. Per ogni metodo testato si introducono:
  - test driver: metodo che chiama il test unit con opportuni parametri (simula unità chiamante).
  - test stub (molto facolitativo): sostituisce eventuali metodi usati dal test unit per testare in modo isolato

- Regressione: test del comportamento di release successive. Obiettivi nel testare programmi modificati (mantenuti):
    - Accertarsi di eliminare i difetti segnalati
    - Non introdurne di nuovi
    - Non buttar via i vecchi casi di test
    - Riusare i vecchi casi confrontando il comportamento

Che cosa testare:

- Funzionalità
- Affidabilità
- Robustezza
- Performance
- Usabilità

Accessibilità del testing:

- White-box testing: sorgente disponibile
    - Basato sulla struttura interna del programma
    - Deriva i casi di test dal programma
    - Controlla "quanto" programma è stato eseguito o coperto
    - Non garantisce che il programma faccia quello effettivamente richiesto
- Black-box testing: guardo solo a cosa dovrebbe fare il sw
    - Deriva i casi di test dai requisiti
    - Controlla il programma attraverso l'I/O
    - Misura gli I/O utilizzati
- Grey-box: mix tra i due

Program-based testing:

1. Esamina la struttura del programma
2. Trova i casi di test che soddisfano un certo criterio di copertura
3. Applica gli input e osserva gli output
4. Controlla che gli output siano quelli attesi

    Limitazione: non riesce a trovare errori di omissione (es: un programma che dimentica un caso particolare) e non fornisce "test oracles" (modo per stabilire se il test ha evidenziato un malfunzionamento oppure no).

Specification-based testing:

1. Esamina la specifica dei requisiti
2. Seleziona un insieme di casi di test che soddisfano un certo criterio
3. Applica questi input alla specifica e colleziona gli output A (attesi)
4. Applica gli stessi input al programma e collezione gli output B (osservati)
5. Confronta gli output A con quelli B e controlla che siano uguali

    Vantaggi: la specifica funziona da oracolo; limiti: le specifiche non sono sempre disponibili devono essere formali, maggiore sforzo rispetto al program-based test.

Validazione e verifica:

- Validazione: controlla che il sw sia corretto (che soddisfi i requisiti)
- Verifica: controlla che l'implementazione corrisponda alla specifica

Validazione del progetto:

- Effettua una revisione del progetto
- Usa revisori esterni
- Usa checklist e scorecard

Valuta l'uso di metodi formali scrivendo un modello del sistema. I metodi formali permettono queste tecniche:

- Analisi sintattiche e di conformità: la specifica deve essere completa e consistente
- Verifica formale di proprietà: dimostrazione formale del rispetto delle proprietà implicite
- Simulazione ed esecuzione di scenari

Tecniche di validazione:

- Statiche: informazioni sul software senza eseguirlo
    - Analisi del flusso di controllo e del flusso dati (valutazione di qualità del codice)
    - Usata in primo luogo dal compilatore
    - Esistono svariati tool: rats, splint, flawfinder
- Dinamiche: analizzano il sw eseguendolo
    - Runtimes checker: analizzano l'esecuzione del codice prima di passare le chiamate al S.O. (es: libsafe, purify plus, immunix tools)
    - Profilers: evidenziano comportamenti anomali del sw
    - Pen test: scansione per vulnerabilità conosciute.

Un programma si può formalmente provare corretto se realizza le specifiche esattamente e formalmente. Logica di Hoare: permette di esprimere le specifiche mediante pre e post condizioni. La prova viene condotta a mano (con l'ausilio di tools) anche se esistono metodi automatici per la prova di correttezza.

Ispezione del codice secondo il modello Fagan (estendibile anche a progetto, requisiti ecc): tecnica completamente manuale.

Ruoli nell'ispezione sw:

- Moderatore: proviene da un altro progetto, fa il capo
- Autore: partecipante passivo
- Lettori / addetti: leggono il codice e cercano i difetti

Checklist: esempio:

- Ogni modulo contiene una singola funzione?
- Il codice corrisponde al progetto dettagliato
- I nomi di costanti sono maiuscoli?
- Ecc.

Un programma P è una funzione da un dominio D (degli input) ad un codominio R (degli output, soluzioni):

- Ok (P, d) (con d in D) se P è corretto per l'input d
- Ok(P) se P è corretto per ogni d in D (cioè tutti Ok (P, d))
    - Malfunzionamento: eseguo P e ottengo NOT Ok (P, d)
    - Difetto: P è stato implementato come P' (quindi sono programmi diversi)
    - Errore: motivo del difetto

Test case: elemento di D.

Test set: sottoinsieme finito di D.

Applicare il test set T al programma P significa: OK (P, T) se per ogni t in T OK (P, t) → test che non scopre errori → test negativo. Not OK (P, t) → test positivo (c'è un errore).

Tipi di test set:

- Ideale: non riscontra difetti
- Esaustivo: è grande quanto tutto il dominio degli input (quindi testa tutto)
    - Impraticabile (D troppo grande)
    - Impossibile (non riesco a capire se il programma si è bloccato)

Test set criteria (parametri per valutare un test set):

- Adeguato: il criterio C è adeguato se mi sa dire che T trova ogni difetto di P rispetto alla specifica S. Quindi C è:
    - Vero: Cps(T) è vero se il test set T è adeguato
    - Falso: se non lo è
- Affidabile: il criterio C è affidabile se per ogni coppia T1 e T2 adeguati secondo il criterio C se T1 becca un malfunzionamento lo becca anche T2 (e viceversa)
- Valido: un criterio C è valido se qualora il programma P non sia corretto esiste almeno un test set T (sotto criterio C) che è in grado di trovare il difetto
- Ideale: se C è affidabile e valido

Un criterio può essere considerato anche un creatore di test

Limiti del testing:

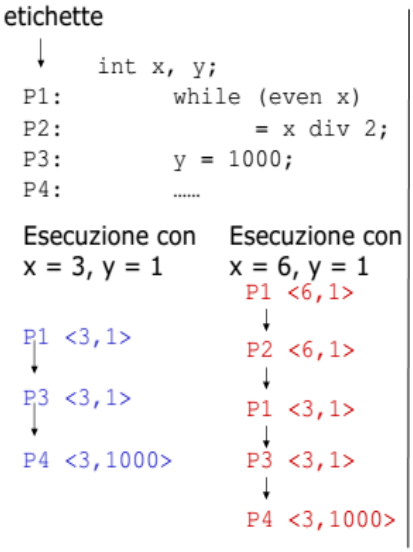
- Non posso trovare in modo algoritmico casi di test ideali
- Il testing non può dimostrare la correttezza del sw

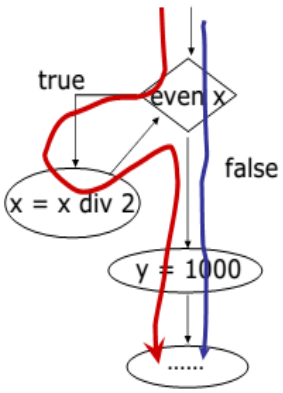
Test criteria non ideali: empirical test criteria: ho testato abbastanza e posso assumere che P sia corretto.

Trovare un insieme di input che esegua tutte le istruzioni è un problema non computabile

Testing basato sulla struttura dei programmi (testing strutturali):

- I criteri di test sono definiti dal codice sorgente
- Si ignora la specifica
- Dal codice sorgente decido quando ho già testato abbastanza
- Testing definiti considerando la copertura del codice (copertura= parte di codice eseguita dai casi di test)
    - Flusso di controllo: rappresentazione grafica del comportamento del programma (sotto forma di flowchart). Rappresenta tutte le possibili esecuzioni.
    - Ad ogni esecuzione verrà eseguita una sequenza di nodi; possiamo rappresentarla con una linea



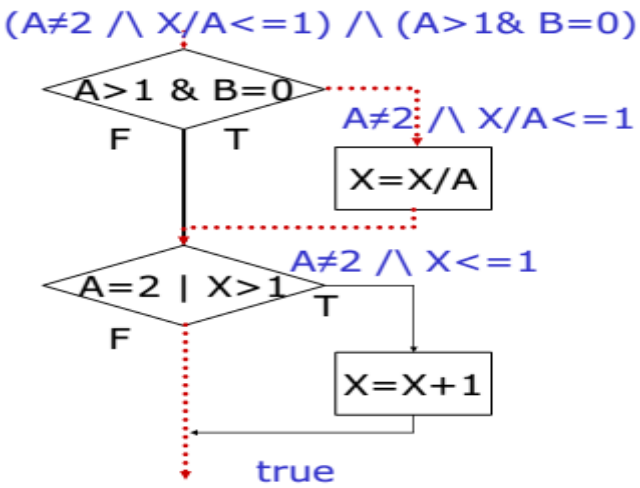


- Copertura delle istruzioni (statement coverage): ogni istruzione viene eseguita almeno una volta. Istruzioni sempre errate vengono individuate
    - Se esegue parzialmente le istruzioni si usa come misura: numero di statement eseguiti/numero di statement eseguibili
- Copertura degli archi (branch coverage): ogni arco è eseguito almeno una volta. Implica anche lo statement coverage.
    - Se esegue parzialmente gli archi si usa come misura: numero di archi eseguiti/numero di archi eseguibili

Valutare un criterio di copertura: Fault detection capability: quali errori è in grado di trovare e quali garantisce di trovare

Trovare i valori di copertura:

- Metti TRUE alla fine del cammino da coprire
- Procedi all'indietro
- Se c'è un assegnamento, sostituisci la variabile assegnata con il valore che è stato assegnato
- Se c'è un ramo vero di una decisione aggiungi la decisione alla condizione
- Se c'è un ramo falso, aggiungi la negazione



Altri criteri di test strutturali:

- Copertura delle decisioni: si agisce sul predicato = guardia di una istruzione condizionale o iterativa (tutta intera! Es: if(x>0 || y>0)): copre ogni sua decisione e la sua negazione (equivalente al branch coverage).
- Delle condizioni: espressione booleana atomica (ogni singolo pezzettino. Es: x>0). Copre ogni condizione e la sua negazione.

Short circuit evaluation (usata dai compilatori): es: a && b → se a è falso non valuto neanche b. Esempio per (y>0 or x>0) →

Se voglio coprire una certa condizione in una decisione la includo nella costruzione della condizione sugli input.

test	y>0	x>0	decisione
y = 1, x = 1	T	non valutata	T
y = 0, x = 1	F	T	T
y = 0, x = 0	F	F	F

Coprire ulteriormente le condizioni all'interno delle decisioni:

- Multiple condition coverage (MCC): un test set soddisfa mcc se testa ogni possibile combinazione di verità delle condizioni atomiche di ogni decisione. Es: per (digit_hight==-1 || digit_low==-1):

Test Case	digit_high == -1	digit_low == -1	
1	False	False	00
2	False	True	01
3	True	(non valutato)	10 11

- Modified condition/ decision coverage (MCDC): ogni condizione della decisione deve far variare in modo indipendente il valore finale della decisione.
  Es: dec= a && b:
  - Considera a e facciamo variare solo a (tengo b a vero)
    - A vero, b vero = dec
    - A falso, b vero = not dec
  - Stessa cosa con b
    - A vero, b vero = dec
    - A vero, b falso = not dec

  Dato che un caso si ripete, ne bastano 3

  Altro es:

```
If Reset = on and
   (Pressure = TooLow or Pressure = Normal) then ...
```

Reset = on	Pressure = TooLow	Pressure = Normal	Valore finale
T	T		T
F	T		F
T	T	F	T
T	F	F	F
T	F	T	T
T	F	F	F

Altri esempi di test per espressioni booleane:

- Full predicate coverage
- Missing condition coverage: se dimentico una condizione atomica faccio in modo che se non ci fosse il risultato finale sia diverso da che ci fosse.