Lez 36 (08/05)

Un aspetto molto importante nella produzione del software è nel **Testing**.
All'inizio abbiamo visto come nel ciclo di vita del software, nella macrofase di sviluppo, non appare una fase dedicata al testing. Questo perché l'attività è così importante da non dover essere trattata come fase a sé stante ma deve essere integrata con ogni altra fase.

Il testing si articola in **Verifica** e **Convalida** (V&V, *Boehm, lo stesso di modello a Spirale e COCOMO*), dove **Verifica** == **controllo di correttezza del prodotto, che sia conforme con la sua specifica** (*alla fine di ogni fase verifico che quanto ho fatto sia corretto con ciò che ho ricevuto in input dalla fase precedente*), "***are we building the product right***" mentre **Validation** == "***are we building the right product***", **ciò che sto realizzando deve essere conforme alle aspettative dell'utente**.
Possiamo quindi realizzare un software corretto (mai al 100% perché nessuno strumento di testing potrà mai garantire che il software sia privo di difetti, obiettivo pulire quanto più possibile da difetti latenti) *ma non valido*.

Abbiamo anche detto che convenzionalmente, **il termine testing viene usato spesso per identificare le attività condotte in modo dinamico** (**Testing Dinamico**, *ossia ho artefatti come codice che posso eseguire per effettuarvi sopra controlli*), si parla invece di **Testing Statico** *quando devo eseguire il testing su artefatti non eseguibili* (**analisi documentale**).

Si parla di **Software Inspections** **quando si fa riferimento a tecniche V&V che riguardano proprio Testing Statico**, mentre di **Software Testing** per **Dinamico**.
Uno dei documenti più importanti dello sviluppo software, il **Test Plan**, *contiene proprio le istruzioni da seguire per poter adottare le tecniche V&V*.
Nel Test Plan, parlando di Software Testing, devono essere descritti i Test Cases *ossia in modo rigoroso quali dati devono essere forniti in input al software e quali gli output attesi*
*-> **una volta definito il Test Plan e i Test Cases l'operazione di Testing è relativamente meccanica*** (si danno gli input descritti dal test case e si osserva se l'output è conforme).

Abbiamo infatti detto che la produzione di Test Case è anche una tecnica di convalida dei requisiti (*se non riesco a definire un test case a partire da una specifica significa che il requisito è scritto male es. requisito: si vuole che il sistema risponda velocemente quando l'utente scrive username e password. Sbagliato, che significa velocemente? Sarebbe corretto se scrivessi "entro 5 ms" e sarebbe possibile definire il test case*).

Abbiamo fatto cenno nella prima parte del corso al fatto che le attività di testing possono avere obiettivi diversi, a seconda del tipo di software in particolare si usano tecniche specifiche di testing.

Quindi **esistono diversi tipi di testing**, vediamo i principali:
- **Validation Testing** == **è il testing utilizzato per dimostrare che il software soddisfa i requisiti utente**. Es. per il requisito dei 5ms, la validazione è verificare nel pratico che inserendo username e password il requisito è soddisfatto.
Normalmente il validation testing viene fatto alla fine dello sviluppo, quando abbiamo a disposizione il codice da testare. Fare validation durante lo sviluppo è molto più difficile, se non si ha a disposizione il codice si deve infatti lavorare con modelli di simulazione del software (*tuttavia vantaggio perché se mi accorgo che la stima data dal modello non soddisfa il requisito posso fermarmi e correggere risparmiando*).
- **Statistical Testing** == **abbiamo detto nella prima parte del corso che dal punto di vista dell'affidabilità** (*probabilità che funzioni correttamente nel mission time*) **il software si comporta in modo molto diverso da qualsiasi altro prodotto** -> *è stato necessario introdurre una teoria di affidibilità software che nulla poteva ereditare dalla teoria preesistente hardware.*
Validare un requisito di affidabilità (es. l'affidabilità della funzione di autenticazione deve essere pari a 99% in tre anni, dove i 3 anni sono il mission time) è ben più difficile che validare un requisito di efficienza (es. tempo di risposta).
Chiaramente non si può validare il requisito di affidabilità aspettando 3 anni per verificare pragmaticamente, si devono fare analisi probabilistiche -> **testing statistico che si basa molto sull'Operational Profile** (descrive come le varie classi di utenti usano il software, con quale probabilità usano certe funzioni etc…) *grazie al quale si possono localizzare e risolvere diversi difetti.*
Questa attività permette, oltre che risolvere difetti, anche di salvare le istanze di fallimento. *Una volta ottenuta la lista di queste istanze dopo il periodo di testing, questa viene data in input a un modello di stima di affidabilità del software che potrà finalmente fornire la stima di affidabilità per capire se il requisito è soddisfatto o meno.*
Quindi software statistico doppio obiettivo: **pulire il software dai difetti** e **ottenere la stima di affidabilità per convalidare i requisiti di affidabilità**.
Tuttavia questo ***testing è molto costoso e tipicamente utilizzato solo per quei software di tipo critico*** (dove i requisiti di affidabilità sono molto stringenti).
- **Defect Testing**: *è quello su cui ci focalizzeremo*. **Racchiude tutte le attività di testing improntate soltanto alla scoperta di difetti latenti.**
Sappiamo infatti che il software può presentare difetti causati da errori umani che possono manifestarsi in guasti.

Defect Testing

Ha come obiettivo scoprire quanti difetti possibile nel programma*, diverso dal validation testing che è più improntato a dimostrare al cliente che quanto stiamo facendo soddisfa i suoi requisiti.*

Come realizzare il Defect Testing?
Si inquadrano due fasi principali di testing. (*tutte queste attività rientrano all'interno della* **Software Quality Assurance** *di cui abbiamo parlato la scorsa lezione*).
La prima fase è la **Component Testing**. ***Questa fase è svolta dallo stesso sviluppatore software e fa riferimento al testing di unità e modulo.***
In pratica il programmatore a cui viene data da codificare l'unità/modulo soluzione della progettazione architetturale (che può anche essere una singola funzione) deve ***verificare che funzioni correttamente*** *basandosi sulle sue competenze*.

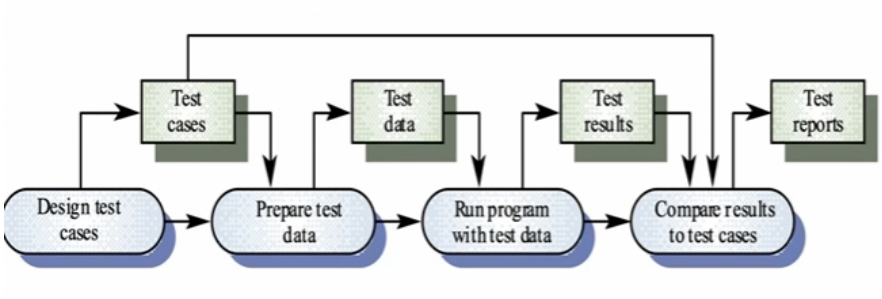
Ma i moduli fanno parte di un'architettura e interagiscono tra loro secondo specifiche relazioni di dipendenza, a questo punto quindi interviene l'Integration Testing.
L'Integration Testing è **svolto a livello di sistema e sottosistema da parte di un testing team indipendente** (gerarchia in termini di grandezza unità – modulo – sottosistema e sistema. Possono anche esserci più sottosistemi in base alla complessità del software).
L'integration Testing si occupa quindi di verificare che l'integrazione tra moduli è corretta o meno.

Infine si ha un'altra fase, che però non fa parte del Defect Testing, che è lo **User Testing** (per verificare che il software fa quanto atteso tramite feedback da utenti).

Si ricordi ancora una volta che il testing esaustivo è impossibile (non è possibile avere la certezza che non vi siano difetti) -> essendo consapevoli di ciò bisogna massimizzare il risultato delle attività di testing -> testare situazioni tipiche è più importante che testare casi specifici.
Per questa ragione il testing deve esser basato su un sottinsieme ben identificato di possibili test cases, in accordo con le politiche dichiarate dal team di testing indipendente (e non il team di sviluppatori).

Come anticipato le attività di testing sono basate sui **Test Cases** e sui **Test Data**.
Test Cases == ***gli input che devono essere forniti dal sistema e l'output atteso in base al requisito che si vuole soddisfare***.
I Test Cases sono tipicamente generati manualmente (costoso!) dato che è difficile generare automaticamente gli output in base a specifiche informali (i requisiti possono essere informali), tuttavia recentemente ia come supporto.
Test Data == ***gli input generati per cercare di identificare quanti più difetti possibile. I test data sono generabili automaticamente.***

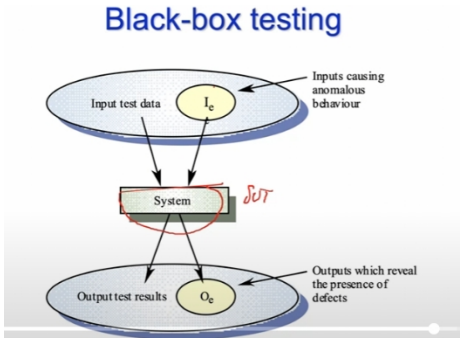
Sia i Test Cases che i Test Data sono utilizzati nel processo di Defect Testing.



Abbiamo visto come il processo software sia costituito da altre attività che a loro volta possono essere sottoprocessi (es. Risk Management Process).
Il defect testing process è uno di questi sottoprocessi, atto a pianificare, eseguire e gestire l'attività di testing.
Rettangoli == documenti, cerchi == attività, freccia == flusso dati e di controllo.
Si parte con la progettazione dei casi di test ottenendo così la lista di test cases che sarà input dell'attività successiva dove preparo gli input (test data), questi saranno presi in input in fase di esecuzione dei test, si memorizzano i risultati dei test e si comparano con i test cases definiti prima per produrre infine il ***test report*** *che mette in evidenza i difetti rilevati.*

Questo appena visto è il processo di defect testing generico, ora vediamo nello specifico come il processo cambia in base al modo in cui lo voglio applicare.
In generale in questo senso distinguiamo due approcci: **Black Box Testing**, *se non guardiamo dentro al software ma lo usiamo semplicemente come farebbe un utente esterno*, e **White Box Testing**, *se invece analizziamo il codice internamente per progettare i test in base alla logica e alla struttura del programma*.

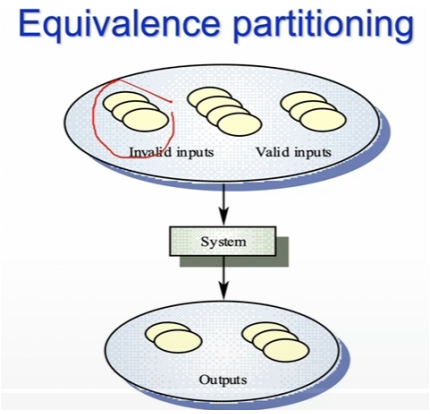
Nel black box testing il software è preso in considerazione semplicemente dal punto di vista di esecuzione, senza vedere cosa avviene al suo interno.
In questo caso i test case sono derivati dalla specifica del sistema e i tester semplicemente forniscono l'input alla componente/sistema e confrontano l'output con quello atteso del test case. Se non corrispondono -> test ha avuto successo e ho riscontrato un difetto.
Questo testing black box è *anche detto* **Testing Funzionale** *in quanto fa riferimento alle funzionalità del software e non alla sua implementazione*.



Quando si forniscono gli input Ie si osserva il comportamento anomalo e si ottiene Oe. *A partire dalla fine di questa attività chiaramente parte il* **debugging**, per cercare di scovare nel codice dove si trovano i difetti che hanno causato l'output anomalo.
Quindi il black box testing mette in evidenza solo il comportamento anomalo, che dovrà essere successivamente investigato per trovare i difetti.

Quali sono le tecniche per cercare di massimizzare la probabilità di trovare input che causino comportamenti anomali?
Immaginiamo di dover testare la funzione che calcola la radice quadrata. Testare per ogni numero intero è impossibile -> si devono usare tecniche che mi permettano quali valori dare in input per scovare più facilmente difetti.

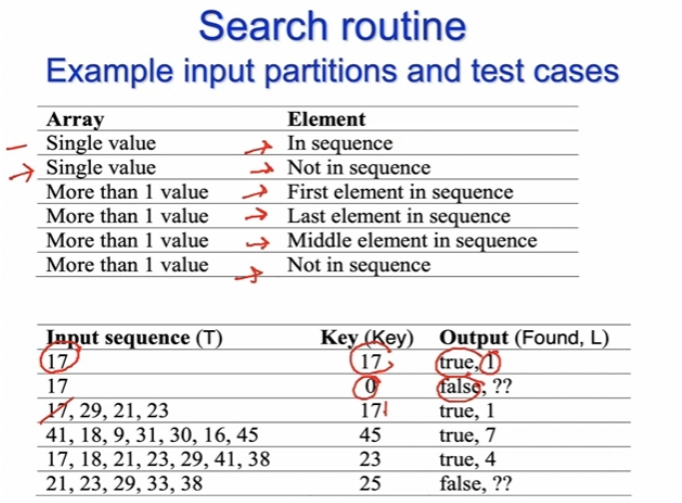
Una di queste tecniche è **l'Equivalence Partitioning**.
Questa tecnica si basa sul fatto che i dati di input e i corrispondenti output possono essere divisi in classi di dati, all'interno delle quali i valori portano il programma a comportarsi in modo simile. **Ognuna di queste classi è chiamata partizione d'equivalenza**, e scegliendo input che fanno parte di ogni partizione riuscirò a coprire buona parte dei comportamenti del programma riducendo l'effort nel determinare tanti test cases.
L'idea è di scegliere alcuni valori da ognuna di queste partizioni per definire i Test Cases.



Come funziona? Tornando all'esempio della radice, supponiamo di doverlo testare con un intero a 5 cifre (tra 10000 e 99999). *La tecnica suggerisce di testare il programma anche per cifre diverse da quelle che si devono testare* (valori non validi), quindi si identificano come partizioni l'insieme di cifre inferiori a 10k, quelle tra 10k e 99k e quelle superiori a 99k.
Quindi il software non si testa solo con input invalidi con questa tecnica.
Nell'esempio ci viene suggerito in particolare di utilizzare come possibili valori input il valore appena inferiore a 10k (9999), 10k, 50k, 99999, 100k.
In generale quindi l'equivalence partitioning è una tecnica che ha senso utilizzare in caso si faccia riferimento a requisiti numerici o in casi in cui le pre-

condizioni e post-condizioni della funzione ci permettono di capire i possibili scenari e quindi identificare le partizioni.

Quando si effettua l'attività di testing possiamo farci guidare da delle **Testing Guidelines** *per limitare il numero di input da testare*, soprattutto per quel che riguarda il testing di sequenze (array, liste etc…). (**NB le Testing Guidelines derivano proprio dall'identificazione di partizioni via equivalence partitioning**).
Esempi di queste linee guida: fare il test con sequenze che hanno solo un singolo valore, usare sequenze di dimensioni differenti, utilizzare sequenze dove il primo, l'ultimo e l'elemento in mezzo siano utilizzati (acceduti) e testare liste vuote.
Qui sotto un semplice esempio di testing su una funzione di ricerca in un array:



Search routine

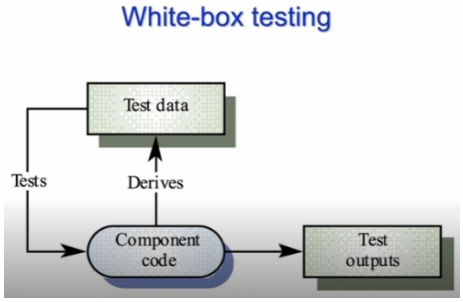
Example input partitions and test cases

Array	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

L'altro tipo di testing, diverso dal Black Box, è lo **Structural Testing** (anche detto **White Box Testing**). In questo caso durante il testing **si ha accesso all'intero codice del programma e i relativi test cases sono derivati dal programma stesso, e non dalla specifica.**
Mentre con il black box l'obiettivo era trovare input con output anomali, con la white box *l'obiettivo è testare almeno una volta quanti possibili program statements* (non tutti i possibili percorsi di esecuzione, anche se vedremo che un tipo di white box testing ricopre questo campo).
Si vogliono quindi identificare i casi di test affinché ogni program statement sia stato testato almeno una volta.

Si identifica in particolare come obiettivo un certo numero, detto **Testing Coverage**, che **rappresenta la percentuale di program statement (istruzioni) che si vogliono raggiungere** (*in base ai tempi, costi, budget a disposizione il numero cambia*).
NB Program Statement == riga/unità di codice, istruzioni.



Lez 37 (13/05)

Abbiamo iniziato ad introdurre tecniche per il Defect Testing che permettano di individuare il maggior numero possibile di difetti in tempi ragionevoli (se dovessi testare tutte le possibili variabili con tutti i loro possibili valori la cosa sarebbe assolutamente ingestibile per software medio grandi).

Con il white box testing l'obiettivo è come anticipato di testare quante istruzioni del programma possibile. **Inoltre in questo caso i test cases non vengono derivati dai requisiti ma dalla struttura del programma**.

Nella figura sopra: dal programma si ottengono i test data che vengono utilizzati per testare lo stesso codice producendo il test outputs che permettono di fare valutazioni sul comportamento del codice.

Mentre la scorsa volta per il black box abbiamo visto la ricerca in un array come esempio, vediamo per fare un esempio del white box testing la bin search.
Binsearch prevede che venga passato in input un valore da cercare, l'array dove cercarlo e un oggetto risultato che contiene due attributi: index (indice dove si trova il valore cercato) e campo boolean found per dire se si è trovato o meno.
L'array viene diviso ricorsivamente in due parti cercando di capire se l'elemento si trova nella prima o nella seconda metà, e la precondizione è che l'array sia ordinato.
Se l'elemento non si trova valore index in Result -1.

```
public static void search ( int key, int [] elemArray, Result r )
{
    int bottom = 0 ;
    int top = elemArray.length - 1 ;
    int mid ;
    r.found = false ; r.index = -1 ;
    while ( bottom <= top )
    {
        mid = (top + bottom) / 2 ;
        if (elemArray [mid] == key)
        {
            r.index = mid ;
            r.found = true ;
            return ;
        } // if part
        else
        {
            if (elemArray [mid] < key)
                bottom = mid + 1 ;
            else
                top = mid - 1 ;
        }
    } //while loop
} // search
} //BinSearch
```

Così come nel black box, dove le linee guida ci suggerivano come la ricerca di un elemento in un array, **nel white box è possibile utilizzare la tecnica di equivalence**

partitioning. *Ma come identificare le partizioni di equivalenza*?
Sappiamo benissimo, conoscendo in dettaglio il codice in questione, che la binsearch suddivide lo spazio di ricerca in tre parti: il punto di mezzo, gli elementi inferiori e quelli superiori.
Quello che si fa quindi è semplicemente identificare ognuna di queste tre parti come partizione di equivalenza -> si identificano un insieme di test case nei quali il valore da cercare sta al confine di ognuna di queste partizioni (*si assume facendo ciò, come per black box, che tali valori siano altamente rappresentativi* per cui se binsearch funziona per loro funzionerà anche per gli altri).

Tra le partizioni di equivalenza che quindi individuiamo per la binsearch: precondizioni soddisfatte, elemento da cercare presente nell'array, poi uguale ma senza che l'elemento sia nell'array, poi precondizioni non soddisfatte e elemento in array e poi non in array, poi se l'array ha un solo valore, se ha numero pari di valori, se ha numero dispari di valori

Input array (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Questa figura molto simile a quella vista la scorsa volta, unica differenza che si provano sempre array ordinati -> non soddisfa tutti i casi della linea guida derivativa dalle partizioni di equivalenza.

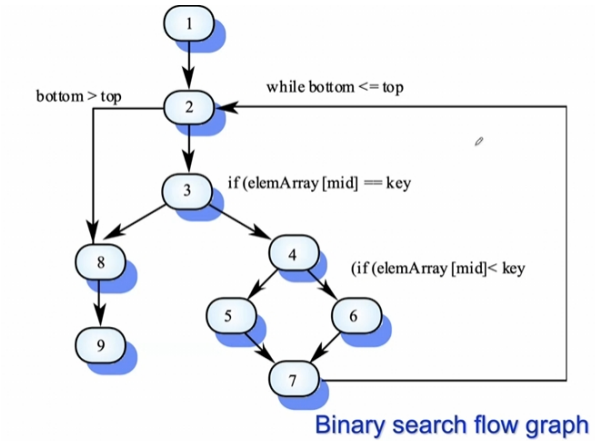
In definitiva anche in questo caso, come per black box ma in generale per defect testing, non si testa la funzione (ossia si derivano test cases) al fine di valutare il soddisfacimento di uno o più requisiti ma soltanto la correttezza della funzione in sé.
L'obiettivo è in questo caso a differenza del black box però è esercitare tutte le istruzioni del programma.

Un approccio diverso, sempre per white box (quindi che richiede la conoscenza della struttura in dettaglio del programma), *è quello orientato al* **testare non tutte le possibili istruzioni ma tutti i possibili percorsi.**
In questo caso implicitamente si effettuerebbe anche il white box testing classico in quanto se si effettuano tutti i percorsi è ovvio che si eseguano anche almeno una volta tutte le istruzioni del programma.

Questo tipo di testing è chiamato **Path Testing**, **l'obiettivo chiaramente è cercare di determinare un insieme di test cases per cui ogni percorso del programma sia**

eseguito almeno una volta ed è chiaramente molto più costoso dell'approccio "classico".
Infatti il numero di possibili percorsi presenti in un programma/flow graph sappiamo essere rappresentato dalla metrica di **Complessità ciclomatica** (introdotta parlando delle metriche di struttura). *Maggiore la dimensione del programma -> maggiore la complessità ciclomatica -> difficile gestione del path testing*.

Il punto di partenza per effettuare Path Testing è proprio il flow graph del programma, che rappresenta tutti i suoi possibili percorsi.
Ciò che si fa quindi è derivare dal codice il grafo di flusso secondo quanto visto in precedenza, vediamo l'esempio proprio con binary search:



Binary search flow graph

L'obiettivo del path testing è in realtà individuare ed eseguire tutti i possibili percorsi indipendenti (da qui il legame con Complessità Ciclomatica), dove con indipendenti si intende il fatto che nel percorso che ho individuato esiste almeno una freccia che non faceva parte di un altro percorso.
Quindi per prima cosa dobbiamo individuare questi percorsi indipendenti.

Una volta individuati, l'obiettivo è identificare casi di test affinché tutti i percorsi in questione siano eseguiti almeno una volta.
Il numero di percorsi indipendenti equivale alla complessità ciclomatica (e-n+2, dove e numero di archi, n numero di nodi) che nel nostro caso è uguale a 4 -> 4 percorsi indipendenti:

- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2
- 1, 2, 3, 4, 5, 7, 2
- 1, 2, 3, 4, 6, 7, 2, 8, 9

Ovviamente in questo caso molto semplice ricavarli, ma tipicamente i flow graph sono molto più grandi e complessi -> si utilizzano **Dynamic Program Analysers** o **Profilers** *che sono strumenti di testing che lavorano assieme ai compilatori ed aiutano i tester a capire quante istruzioni sono state eseguite e quale percorso* (in pratica quando si esegue il programma grazie al profiler si identifica un execution

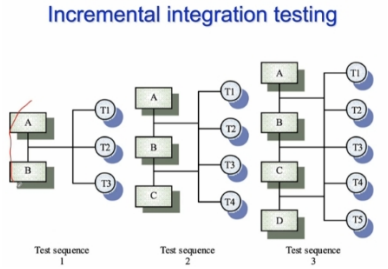
profile che mostra quali parti del programma sono stati eseguiti dal test case, per capire se si sta procedendo secondo quanto pianificato o meno).

Quanto visto finora RIGUARDA IL TESTING DI SINGOLI COMPONENTI, *riassumendo per testare i singoli componenti due approcci: black e white box, utile tecnica per ridurre il numero di test cases da effettuare è l'equivalence partitioning e mentre con il black box l'obiettivo è determinare se a fronte di un certo input l'output è corretto ed equivale quindi a quanto ci si aspettava, con white box testing si vogliono individuare i test case che permettono di esercitare il programma al fine di eseguire tutte le istruzioni o tutti i percorsi nel caso Path Testing usando ovviamente un numero quanto più possibile ridotto di test cases.*

Questo per quanto riguardava il testing dal punto di vista di componenti. Sappiamo tuttavia che nello sviluppo di software medio-grandi il numero di queste componenti è significativo e nonostante l'obiettivo di una buona modularità è minimizzare il coupling e massimizzare la cohesion (minime interazioni tra moduli diversi per poter garantire migliore manutenibilità) ***sappiamo che dopo la fase di progettazione dettagliata e dopo la fase di codifica questi moduli dovranno essere integrati nella fase di integrazione***.
In questa fase è importante spendere effort nell'**Integration Testing**, per convalidare l'integrazione di tutti i moduli.

Per il testing d'integrazione non ci interessa più entrare in dettaglio nei singoli moduli -> **testing di tipo black box con casi di test derivati direttamente dal Documento di Specifica** (*qui si sottolinea ancora una volta l'importanza di questo documento*).
La difficoltà principale in questo tipo di testing è identificare gli errori, in quanto possono esservi interazioni complesse tra le componenti del sistema e quando si scopre un output anomalo può esser complesso capire il perché.
Per tale ragione **si suggerisce di adottare un approccio incrementale** nell'Integration Testing.

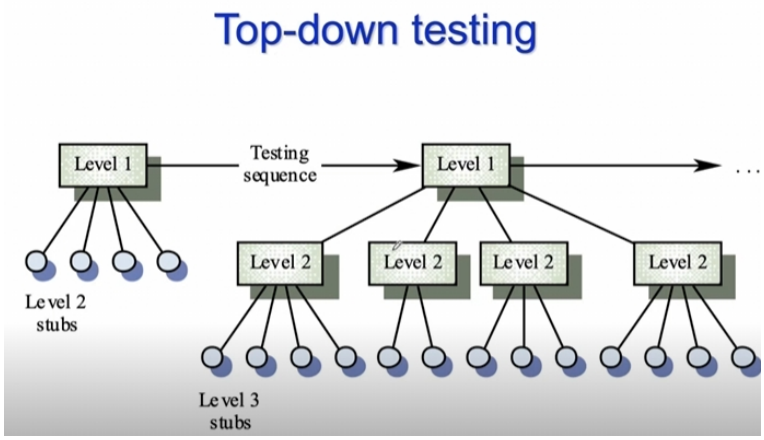
Per iniziare la fase di Integration Testing non è necessario che tutte le componenti siano disponibili, ma solo la parte d'interesse. Per questo è possibile procedere con il testing incrementale:



A, B… componenti e T1, T2… attività di testing. **Vantaggio: se nella test sequence 1 non si trovano errori ma nella 2 sì -> è più semplice localizzare l'errore** (*probabilmente dipende da C che è la nuova componente testata insieme ad A e B*). Se avessi testato subito il sistema finale avrei avuto molte difficoltà a scovare gli errori.

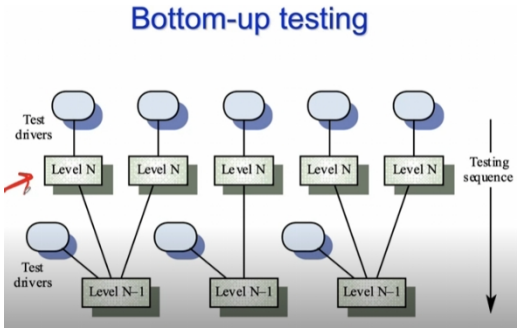
Vediamo ora quali approcci per il testing d'integrazione si possono adottare, sapendo che le componenti possono essere integrate a vari livelli come visto (**livello gerarchico**, dalle componenti a basso lvl fino alle più importanti ad alto lvl):
- **Top-down testing**: **si parte dai livelli più alti di gerarchia fino ad arrivare alla radice per componenti "meno rilevanti"**. Poiché si parte da un insieme vasto di componenti è possibile che queste non siano presenti, in tal caso sono sostituite con degli **stubs**, *ossia delle funzioni che emulano il comportamento della componente*.
- **Bottom-up testing**: **si procede dall'integrare le componenti individuali a basso livello di gerarchia fino al sistema completo** (come visto nell'esempio).

Non esiste un approccio migliore, tipicamente anzi queste strategie sono combinate in base alle situazioni (es. disponiblità di componenti, stub etc…).



Come detto nell'approccio top-down le componenti ad alto livello sono sviluppate prima di quelle a basso livello -> si devono usare gli stub che hanno la stessa interfaccia del componente ma funzionalità limitate (un emulatore della funzione che sarà integrata dal modulo).
Si parte quindi dalle componenti a livello più alto sostituendo quelle di basso livello non ancora disponibili con stubs. Quando le componenti saranno disponibili, si sostituiranno e così via.

Riguardo **il Bottom-up testing** invece **si parte da componenti più in basso nella scala gerarchica iterativamente fino a testare l'intero sistema. Poiché le componenti sono stavolta disponibili per il test si utilizzano dei Test Drivers, codice che eserciterà il componente per valutarne la capacità di integrazione con i livelli successivi.**



Bottom-up e Top-down possono essere confrontati da 4 punti di vista:
- **Architectural Validation**: *usando bottom-up il progetto architetturale non è validato fino alla fine del processo di testing*, mentre *con top-down è più facile identificare gli errori subito a partire proprio dall'architettura di sistema*.
- **System Demonstration**: *l'approccio top-down permette di costruire demo con funzionalità limitate* (perché devo includere gli stubs) *che può essere usata come strumento di convalida fin dall'inizio del testing.*
- **Test Implementation**: *dal punto di vista dell'effort è più facile costruire i test driver per bottom-up piuttosto che stubs per top-down*.
- **Test Observation**: *ossia valutare i risultati prodotti dall'attività di testing, in entrambi i casi è difficile capire in base agli output dove si trova l'errore e può essere necessario codice artificiale* (in più) *per valutare correttamente* (stubs e driver appunto)

Come detto nella scorsa lezione un componente può essere testato dal punto di vista della sua struttura (whitebox) oppure dal punto di vista degli output che produce (blackbox). **Per testare un componente blackbox si sfrutta l'interfaccia del componente, e i componenti interagiscono tra loro tramite l'interfaccia**!

Dal punto di vista dell'integration quindi due componenti che interagiscono tra loro conoscono solo la loro rispettiva **interfaccia**, non i dettagli implementativi.
Per queste ragioni una tecnica utile per l'Integration Testing è l'Interface Testing.

Ogni modulo o sottosistema ha un interfaccia ben definita, utilizzata per chiamare quel modulo/sottosistema da altri componenti.
L'obiettivo di questo testing dell'interfaccia è scoprire difetti introdotti a causa di errori d'interfaccia o assunzioni non valide relative all'interfaccia.
Questo testing è molto importante nel mondo object oriented perché gli oggetti sono definiti secondo la loro interfaccia.

L'obiettivo dell'interface testing è quindi identificare dei test cases che testino l'interfaccia del componente e non la sua struttura interna.

Esistono **4 tipi di interfaccia** che possono essere utilizzati per realizzare l'interazione tra componenti/sottosistemi:
- **Interfaccia basata su Parametri**: *le componenti si passano dei parametri per interagire*
- **Interfaccia basata su Memoria Condivisa**: *le componenti interagiscono accedendo in modalità rw ad un blocco di memoria condiviso tra loro*
- **Interfaccia Procedurale**: *la componente incapsula un insieme di procedure che possono essere chiamate da altre componenti*
- **Interfaccia basata su scambio di Messaggi**: *una componente richiede un servizio da un'altra componente inviandogli un messaggio e attendendo una risposta*.

Nel mondo OO e in generale in sistemi basati su tipi di dato astratti le interfacce utilizzate sono quelle Procedurali: *l'oggetto risponde a richieste specificate nell'interfaccia di classe e le operazioni sono implementate nel corpo dei metodi della classe.*
Invece **Message Passing** interfaccia *usata in architetture Client-Server*.

Gli **errori legati all'interfaccia** (quindi non interni alla struttura ma legati a come il componente interagisce con gli altri) sono i seguenti:
- **Utilizzo Scorretto dell'Interfaccia**: es. una componente ne chiama un'altra scrivendo i parametri in modo sbagliato
- **Incomprensione dell'Interfaccia**: *il componente chiamante potrebbe fare delle assunzioni errate nei confronti del comportamento del componente chiamato* es. se passo al binary search un array non ordinato (in questo caso e anche sull'utilizzo scorretto il problema non è dell'interfaccia ma dovuto al modulo chiamante)
- **Timing Error**: *le componenti lavorano a velocità differenti e quindi non sincronizzate*: es. richiedo lo stato di un attributo e ottengo uno stato vecchio perché nel frattempo è stato aggiornato (questo tipo di errore è meno comune in quanto si osserva nei sistemi real time, gli altri due più comuni)

Per ottimizzare le attività di interface testing si possono fornire linee guida, che includono: *progettare test case che facciano fallire il componente* (es. usando parametri sbagliati), *usare **stress testing*** (progettare test case che generano molti più messaggi rispetto a quelli generati normalmente, per testare la robustezza), *modificare l'ordine con cui i componenti vengono attivati*.

In generale sono le tecniche statiche quelle più adatte all'interface testing (ricorda dinamiche se eseguo il software per fare testing, statiche se mi baso sulle conoscenze che ho sul progetto es. documento di specifica etc..)

Ora di queste guidelines approfondiamo lo **Stress Testing**.
Come detto, una volta che il sistema è stato integrato, **l'obiettivo dello stress testing è pianificare una serie di test dove il carico viene incrementato di volta in volta**

finché le prestazioni del sistema diventano inaccettabili.
In questo modo misuro la **robustezza** del software e capisco *fino a che limite posso spingerlo* (es. progetto un sistema operativo per avere fino a 200 terminali separati -> con questo testing verifico se requisito soddisfatto e quanto posso spingermi oltre).

Ovviamente lo stress testing deve essere calibrato affinché il sistema non fallisca in modo catastrofico, inoltre il fallimento non deve causare una perdita eccessiva di servizi e dati.
Questo testing è particolarmente importante in sistemi distribuiti, dove si possono osservare cali prestazionali a casa di sovraccarichi di rete.

Altro aspetto importante è **l'Object Oriented Testing**, dove **le componenti da testare sono classi che si istanziano in fase di esecuzione come oggetti.**
In questo caso l'unità di codice è tipicamente superiore rispetto alla singola funzione (infatti la classe tipicamente incapsula un insieme di funzioni) -> si devono estendere in questo senso gli approcci whitebox.
Inoltre un sistema OO è costituito da un insieme di oggetti che si scambiano messaggi -> non è un sistema organizzato gerarchicamente come visto in precedenza e per questo sistemi di questo tipo non particolarmente adatti ad approcci top-down o bottom-up.

In generale quindi l'obiettivo deve essere testare il sistema OO per livelli, testando anzitutto ***i metodi degli oggetti*** (qui posso usare approcci black e whitebox), ***poi gli oggetti/classi nella loro interezza***, ***poi cluster di oggetti che interagiscono tra loro*** (*qui non adatto il top-down o bottom-up*, si usa tipicamente lo **scenario testing** testare il sottosistema in base agli scenari) **fino al sistema OO completo**.

Vediamo cosa significa fare testing di classi/oggetti: **fare un test coverage completo di un oggetto significa** testare tutte le *operazioni ad esso associate*, *assegnare valori a tutti i suoi attributi e testare l'oggetto in tutti i suoi possibili stati* (qui ci viene in aiuto lo **State Diagram** che mette in evidenza tutti i possibili stati che l'oggetto può attraversare interagendo con altri oggetti).

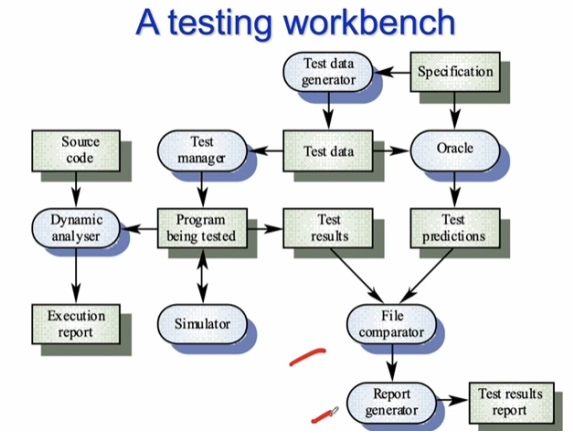
L'ereditarietà rende più difficile il testing di oggetti: se la mia classe fornisce un insieme di operazioni ereditate da sottoclassi -> tutte le sottoclassi in questione dovrebbero a loro volta essere testate.

L'integration Testing per sistemi OO riguarda non più un insieme di componenti ma un insieme di classi -> l'idea è di analizzare un cluster di oggetti che interagiscono tra di loro. Si parla quindi di **Cluster Testing**, ossia testare le classi insieme.

In questo caso come detto non esistono relazioni gerarchiche -> non ha senso usare top-down o bottom-up, esistono altri 3 approcci:
- **Use-case o scenario testing**: ***il testing è basato sulle situazioni in cui l'utente potrebbe incorrere nell'utilizzare il sistema*** (a tutti gli effetti si testano i casi d'uso)
- **Thread Testing**: ***usato in sistemi OO basati su eventi, Consiste nel verificare che il sistema risponda correttamente a una specifica sequenza di eventi***
-**Object interaction testing**: ***viene testata una sequenza di messaggi tra oggetti ben definita***

Parlando del **testing basato su scenario** ***gli scenari sono come detto identificati da casi d'uso***, che però non potrebbe includere abbastanza informazioni per derivare un insieme appropriato di test cases. **Quindi il caso d'uso può essere affiancato dal diagramma d'interazione sequence diagram** definito in fase di specifica dei requisiti, che può aiutare in questo senso.
(ricorda che i diagrammi d'interazione sono il sequence e il collaboration (comunication in UML 2) e mostrano più in dettaglio come interagiscono gli oggetti in un singolo caso d'uso rispetto a quanto fatto con il diagramma delle attività, che può esser svincolato dalle classi e oggetti che realizzano quella attività).

Per gestire l'attività di testing per software di dimensioni medio-grandi si utilizzano diversi ***strumenti***, tra cui i **Testing Workbenches**. Si tratta di un **insieme organizzato di tool che permettono di organizzare correttamente l'attività di testing anche ottimizzando l'uso di risorse in termini di tempi/costi**.
Spesso questi workbenches sono specifici per l'organizzazione -> alcuni degli strumenti sono realizzati proprio dall'organizzazione stessa



(tool in verde).