DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.

CoScienze
Associazione

Antonio La Marca

NOTA: Non sono trattati tutti gli argomenti (es. Risk Management e Quality Management)

IGES

Ripasso pre-esame

ANTONIO LA MARCA

Sommario

Introduzione

La manutenzione del software rappresenta la totalità dell'attività richiesta per fornire supporto cost-effective ad un sistema software, sia durante la fase di pre-delivery così come nella fase di post-delivery (cambiamento del software).

Abbiamo quattro tipi di manutenzione:

- **Correttiva**, corregge i malfunzionamenti, i RF o RNF non vengono rispettati; vengono isolati e corretti gli elementi difettosi, effettuati a seguito di una modification request dagli utenti
- **Adattiva**, permette al sistema di adattarsi al cambiamento del contesto in cui si trova.
- **Perfettiva**, vengono migliorati gli attributi interni/esterni del SW.
- **Preventiva** (a volte considerato sottoinsieme della perfettiva) bisogna prevenire l'insorgere di un problema, importante per la sicurezza del SW. Viene svolto quando il SW si rivolge ad un mercato preciso. Viene considerato un ringiovanimento del SW.

Kitchenham et al. organizzare le attività di modifica della manutenzione in due categorie:

- Correzioni: comprendono Correttiva e Preventiva
- Miglioramenti: comprendono Adattiva e Perfettiva

In generale **manutenzione** significa preservare il software dal declino o dal fallimento, mentre **evoluzione** significa un software che cambia continuamente da uno stato peggiore a uno migliore.

L'evoluzione del software nasce nel 1965, tra le sue caratteristiche troviamo che è il risultato della manutenzione del software, non è un attività e non viene effettuata direttamente su richiesta e fornisce un feedback sul processo di manutenzione del SW. Gli approcci di studio sono:

- Spiegazione(What/Why): l'approccio tenta di spiegare la causa dell'evoluzione del software da un punto di vista scientifico.
- Miglioramento del processo(How): l'approccio tenta di gestire gli effetti dell'evoluzione sviluppando metodi e strumenti migliori, ovvero progettazione, manutenzione, refactoring e reingegnerizzazione, da un punto di vista ingegneristico.

Evoluzione e Manutenzione sono due facce della stessa medaglia; faccio manutenzione perché mi viene chiesto di cambiare qualcosa, l'effetto dell'intervento di manutenzione è l'evoluzione del software, poiché i l software si è evoluto.

Il processo di manutenzione è costituito da diverse componenti, quali:

- **prodotto manutenuto**, che comprende prodotto, inteso come collezioni di artefatti, miglioramento del prodotto, rispetto ad una certa baseline, e artefatti, che possono essere documenti, codice, ecc
- **tipi di manutenzione**, che comprendono attività di investigazione, modifica, management e quality assurance, e ogni attività prende artefatti come input e ne ritorna altri o nuovi
- **processi di organizzazione della manutenzione**, a livello individuale o a livello di organizzazione; una volta approvata una CR viene firmato SLA con clienti (Service Level Agreement (SLA) è un contratto tra i clienti e il fornitore di un servizio di manutenzione. Gli obiettivi della manutenzione sono specificati all'interno dello SLA.)
- **peopleware**, perché l'uomo è centrale in ogni processo realizzato (Le attività di manutenzione non possono ignorare l'elemento umano, perché la produzione e la manutenzione del software sono attività ad alta intensità umana.)

La tassonomia SPE serve per classificare come un programma varia nelle sue caratteristiche evolutive, e distinguiamo:

- **specified programs**, dove tutte le proprietà funzionali e non sono definite e la sua correttezza dipende solo dalla conformità a questi
- **problem programs**, costruiti per astrarre e cercare di trovare soluzione ad un problema, vengono valutati rispetto alla loro efficacia di risolvere il problema
- **evolving programs**, incorporati nel mondo reale per meccanizzare attività umane, e la loro valutazione dipende solo da una valutazione basata su giudizio

Leggi di Lehman

Le leggi dell'evoluzione del software sono state formulate da Lehman, come risultati di studi su sistemi CSS, e sono relative a sistemi E-type. Il loro scopo è studiare la natura dell'evoluzione del software e le traiettorie prese dall'evoluzione.

1	**Cambiamento continuo**	Un programma usato in un ambiente reale avrà necessariamente bisogno di cambiamenti o diventerà meno utile in quell'ambiente
2	**Crescita della complessità**	Al cambiare di un sistema, la sua struttura tende a diventare più complessa, a meno che non venga svolto del lavoro mirato ad abbassarla
3	**Auto regolazione**	L'evoluzione è un processo auto-regolato. Attributi come dimensione, intervallo tra release e numero di errori individuati, sono approssimativamente invariati
4	**Conservazione della stabilità organizzativa**	Il tasso di sviluppo effettivo medio su un sistema in evoluzione non cambia nel tempo, cioè la quantità media di lavoro che si impiega in ogni release è all'incirca la stessa
5	**Conservazione della familiarità**	Il numero di nuovi contenuti in ogni release del sistema tende ad essere costante o a diminuire nel tempo
6	**Crescita continua**	Il numero di funzionalità in un sistema incrementerà nel tempo, al fine di soddisfare gli utenti
7	**Declino della qualità**	Un sistema sarà percepito come un sistema che perde qualità nel tempo, a meno che la sua progettazione non venga attentamente mantenuta e adattata ai nuovi vincoli operativi
8	**Feedback system**	Il successo dell'evoluzione di un sistema software richiede il riconoscimento che il processo di sviluppo è un sistema di feedback multi-loop, multi-agente e multilivello.

I sintomi dell'invecchiamento del software sono pollution, conoscenza integrata, dizionario scarso e accoppiamento. Il codice si dice decaduto se è molto difficile modificarlo, e si riscontra in rapporto ai tre fattori, quali costo del cambiamento, tempo e qualità del software modificato.

I sistemi FOSS presentano un'evoluzione molto diversa rispetto a struttura del team, processi e rilasci.

Una componente COTS è un'unità software indipendente con interfacce e dipendenze esplicite e specificate da contratto. Il loro utilizzo è fondamentale oggigiorno per lo sviluppo software. L'unico codice sorgente scritto e modificato è quello necessario ad integrare i sistemi basati su COTS.

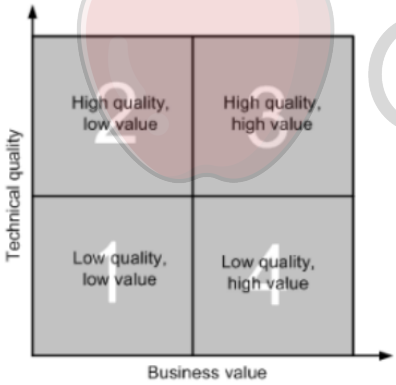
Legacy System

I sistemi legacy sono sistemi vecchi business critical in quanto le funzionalità offerte sono vitali per l'azienda. Spesso resistono significativamente alle modifiche e all'evoluzione che servono ad andare incontro ai requisiti di business che cambiano ed è quindi un sistema difficile da far evolvere.

Esistono differenti soluzioni applicabili ai LIS che sono:

- **Freeze**: i sistemi non evolvono più per cui vengono incapsulati, venendo utilizzati così come sono, magari facendo solo manutenzione correttiva.
- **Carry on maintenance**: Nonostante tutti i problemi di supporto, l'organizzazione decide di continuare la manutenzione, sia correttiva che evolutiva, per un altro periodo.
- **Discard and redevelop**: Buttare via tutto il software e sviluppare l'applicazione ancora una volta da zero
- **Outsource**: la manutenzione è fatta da un'azienda di terze parti o costruisco o compro un prodotto che possa sostituire il sistema legacy
- **Wrapping**: Wrapping e freeze possono essere usate insieme, usando le funzionalità del vecchio sistema che non si evolve più mediante un wrapper.
- **Migrate**: effettuando la migrazione non si esclude il wrapping, in quanto si può migrare wrappando prima le componenti. Incrementalmente, il sistema legacy viene wrappato verso la nuova piattaforma.

Per decidere che soluzione applicare andiamo ad effettuare un'analisi del sistema sulla base della qualità tecnica e del valore per l'azienda.



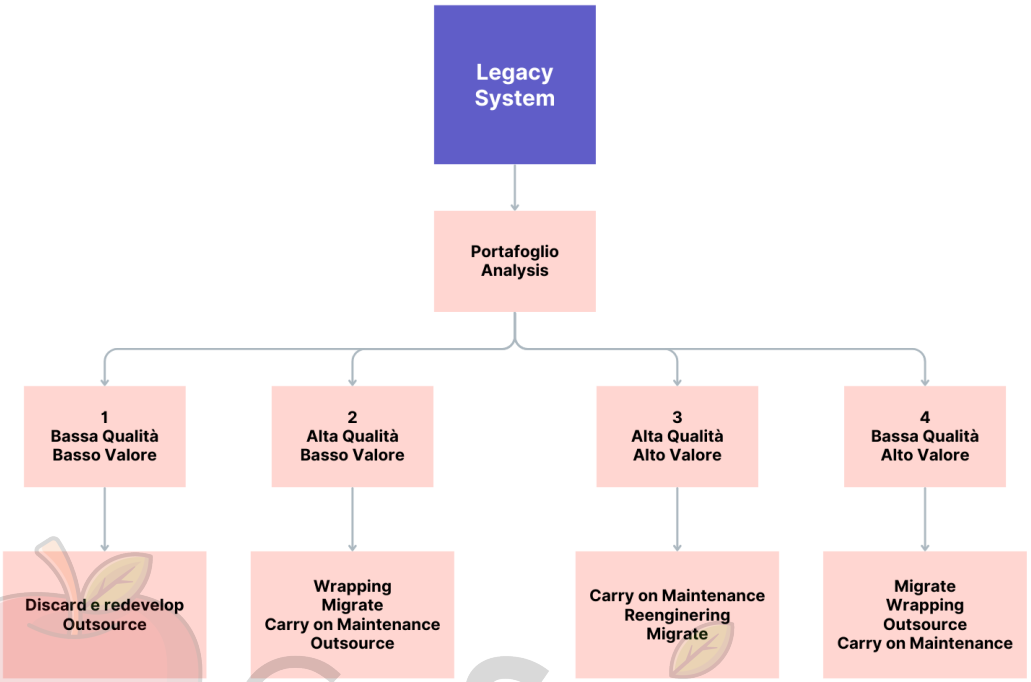
La qualità tecnica la misuriamo sul livello di obsolescenza tecnologica, quando è decomponibile il sistema e sull'architettura. I sistemi possono essere divisi in tre categorie: **decomponibili**, se le componenti che svolgono la logica applicativa sono indipendenti l'una dall'altra e interagiscono con componenti che si occupano di gestire i dati e mostrarli all'utente; **semi decomponibili** se solo l'interfaccia utente presenta componenti separate; **non decomponibili** se tutte le componenti del sistema sono accoppiate tra loro. Inoltre, facendo evoluzione del sistema molto spesso, la qualità tecnica potrebbe diminuire (2 legge di Lehman).

Per il valore aziendale invece, si fa riferimento a quanto il software fa quello che deve fare per l'azienda.

L'analisi ci permette di scegliere l'attività da effettuare in base al sistema che stiamo analizzando, definendo un grafo ed individuando quattro casistiche:

- **Quadrante 1** (bassa qualità e basso valore): possiamo fare **discard and redevelop** in quanto il sistema è obsoleto sia tecnicamente che dal punto di vista del valore aziendale; quindi, è più conveniente abbandonarlo e sviluppare da zero una nuova soluzione; oppure anche **freeze**
- **Quadrante 2** (alta qualità e basso valore): avendo una buona architettura è possibile rimpiazzare solo alcune delle sue componenti che non si sono evolute alle esigenze di business. Si potrebbe in alternativa fare manutenzione per riallineare le componenti ai requisiti. Avendo un buona architettura posso fare migrazione o sostituzione delle componenti obsolete.

- **Quadrante 3** (alta qualità ed alto valore): finché si sta in questo quadrane, il sistema sta evolvendo e si devono fare operazioni per mantenere la qualità del sistema alta (refactoring e reenginering)
- **Quadrante 4** (bassa qualità ed alto valore): Il sistema è evoluto come doveva ma non sono state fatte operazioni di ristrutturazione per migliorarne la qualità; quindi, dobbiamo fare refactoring e reengineering



Reengineering

Il reengineering è l'analisi e la ristrutturazione di un sistema software esistente per ricostituirlo in una nuova forma. Lo scopo é capire gli artefatti esistenti per migliorare funzionalità, proprietà e qualità del sistema o effettuare migrazioni. Se miglioro le funzionalità, il sistema che ho realizzato non deve necessariamente avere le stesse funzionalità del sistema di partenza. Si potrebbero quindi modificare le funzionalità del sistema. Spesso si fa reengineering per migliorare la manutenibilità, per preparare il software ad un miglioramento funzionale o per migrare il software verso una nuova tecnologia. In tal caso si potrebbero aggiungere funzionalità in quanto, se si migra un'applicazione da stand-alone a Client-Server, è chiaro che si possano modificare le funzionalità. La migrazione verso una nuova piattaforma è propedeutica all'aggiunta di funzionalità.

Il **forward engineering** è il processo in cui sviluppiamo un sistema software, partendo dal più alto livello di astrazione ed andando verso il livello più basso. Analogamente, il movimento verso l'alto attraverso gli strati di astrazioni è chiamato **reverse engineering**, con cui si analizza il software per determinare i suoi componenti e le relazioni fra essi, e si rappresenta il sistema ad un livello più elevato di astrazione.

Il reenginering si basa sui concetti di **astrazione**, che serve per ridurre il livello di complessità di comprensione del sistema nascondendo dettagli a basso livello, viceversa il **raffinamento**. Possiamo individuare, come livelli di astrazione, concettuale, requisiti, design e implementazione.

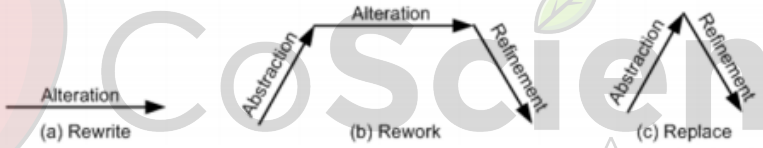
Un ulteriore processo é **l'alterazione**. In teoria, facendo astrazione e raffinamento, quello che ottengo è quello che avevo all'inizio. Non è detto che io ottenga lo stesso codice, ma otterrò

fondamentalmente codice che farà le stesse cose, poiché non si stanno cambiando i requisiti. Con il principio di alterazione, si decide il livello a cui si vuole fare reengineering, facendo astrazione fino a quel livello. Si fa poi un'alterazione della rappresentazione di quel livello e si fa poi un'operazione di raffinamento.

Posso fare quindi reengineering a diversi livelli di astrazione (ferro di cavallo):

- **rethink (livello concettuale):** in cui si vanno a modificare le caratteristiche concettuali del sistema, per cambiare il dominio applicativo del problema in uno diverso, ad esempio se passaggio da cellulare a smartphone.
- **respecify (livello dei requisiti)**, in cui modifico le caratteristiche dei requisiti nella forma o nello scope.
- **redesign (livello di design)** vado a rivedere l'architettura del sistema. Modifiche comuni sono: ristrutturazione dell'architettura, modifica, rimpiazzare un algoritmo o una procedura con una più efficiente.
- **recode (livello implementazione)**: Le caratteristiche di implementazione del programma sorgente vengono modificate riscrivendolo. Le modifiche al livello del codice sorgente vengono eseguite mediante:
    - **riformulazione**, conservando il linguaggio di programmazione originale, ad esempio facendo refactoring
    - **traduzione**, trasformando il linguaggio di programmazione, ad esempio facendo compilazione, decompilazione e migrazione

Abbiamo poi diverse strategie di reingegnerizzazione:



- **rewrite** riflette il principio dell'alterazione, dunque un sistema viene trasformato in uno nuovo mantenendo lo stesso grado di astrazione
- **rework**, che prevede utilizzo di astrazione per analizzare codice e generare CFG, alterazione per trasformare CFG in uno strutturato, e raffinamento per tradurre quest'ultimo nel linguaggio originale
- **replace** prevede applicazione di astrazione per nascondere dettagli implementativi della caratteristica considerata ed applicare raffinamento successivamente

Abbiamo diversi modelli di processo di reingegnerizzazione, quali:

- **approccio big-bang**, con cui si sostituisce l'intero sistema in una sola volta., ed una volta avviato lo sforzo di reengineering, esso viene continuato fino a quando tutti gli obiettivi del progetto sono raggiunti e il sistema di destinazione è costruito; consuma troppe risorse per grandi sistemi e ci vuole tempo prima che il sistema sia visibile
- **approccio incrementale**, con cui il sistema è reingegnerizzato gradualmente, uno step alla volta più vicino al sistema obiettivo, permettendo di rilevare più velocemente progressi ed errori, ma richiede più tempo per essere completato
- **approccio parziale**, con cui solo una parte del sistema é reingegnerizzata e integrata nel sistema, prevedendo che il sistema venga diviso in due parti, si riprogetta tramite Big Bang o incrementale e si uniscono; ha lo svantaggio che non prevede modifiche all'interfaccia fra le due parti

- **approccio iterativo**, con cui ogni operazione di reengineering dura poco e viene ripetuta sui vari componenti in diverse fasi; permette la coesistenza di vecchi componenti, componenti attualmente in riprogettazione, componenti riprogettati e nuovi componenti; ha il vantaggio di garantire il funzionamento del sistema durante la sua esecuzione, ma richiede tracciamento continuo
- **approccio evolutivo**, modo diverso di vedere l'approccio iterativo e incrementale. In questo caso le componenti sono raggruppate in base alle funzionalità e reingegnerizzate in nuove componenti

Il **code reverse engineering** consiste nell'analisi di un programma esistente, partendo dal codice sorgente o binario, per comprenderne il funzionamento interno. Questo processo serve a identificare le componenti del sistema, le loro relazioni e a rappresentarle a un livello di astrazione più alto. Le principali applicazioni sono:

- **Ridocumentare il sistema**: creare rappresentazioni di livello superiore per semplificarne la comprensione.
- **Supportare la progettazione di basso livello**: ad esempio, generando liste di riferimento incrociato (*cross-reference list*) che mostrano dove sono dichiarate e utilizzate le variabili.
- **Effettuare design recovery**: ricostruire la struttura e l'architettura del software per risalire al progetto originale.

La maggior parte delle attività di reverse engineering si concentra sul design recovery. Tuttavia, salire a livelli di astrazione superiori, come i requisiti, è complesso e richiede intervento umano, poiché non può essere automatizzato. Ad esempio, per ricostruire i requisiti bisogna osservare il sistema in esecuzione e definire i casi d'uso.

Dal codice si possono estrarre diagrammi, come i *sequence diagrams*, ma questi rappresentano molti dettagli tecnici. Sono diversi da quelli prodotti in fase di analisi, che si concentrano sulle interazioni principali tra oggetti e dominio applicativo. Stabilire quali dettagli includere o escludere in una rappresentazione di alto livello non è semplice e per questo non automatizzabile, in quanto bisogna interpretare le scelte progettuali fatte in origine

La differenza tra reverse engineering e design recovery sta proprio nel livello di automazione: il reverse engineering è più automatizzato, mentre il design recovery richiede più intervento umano per integrare informazioni non ricavabili direttamente dal codice.

Quindi, in sintesi nel Reverse engineering si parte dall'implementazione per risalire al design, per effettuare design recovery che richiede aggiunta di informazioni e interventi manuali per ricostruire decisioni progettuali e architettura.

Il paradigma **Goals/Models/Tools** è un criterio per il reverse engineering che suddivide il processo in tre fasi.

- fase Goals, si identificano le ragioni per avviare il processo, si analizzano le informazioni necessarie e si acquisisce una conoscenza dell'ambiente
- fase Models, si creano modelli rappresentativi delle astrazioni identificate, definendo documenti e algoritmi di astrazione.
- fase Tools, vengono identificati e acquisiti gli strumenti necessari, suddivisi in tool per estrarre informazioni e tool per produrre la documentazione desiderata

Le tecniche di analisi utilizzate nel contesto del reverse engineering sono:

- **analisi lessicale**, con cui si decompone la sequenza di caratteri del codice sorgente nelle sue unità lessicali costituenti, sfruttando regole di struttura definite mediante espressioni regolari
- **analisi sintattica**, con cui si esaminano le relazioni tra le unità precedentemente identificate per controllare che aderiscono alle regole della grammatica del linguaggio
- **control flow analysis**, con cui si analizza il flusso di controllo del codice esaminato; può essere di tipo intraprocedurale se mostra l'ordine in cui gli statement sono eseguiti, o interprocedurale se mostra le relazione chiamanti fra moduli del programma
- **data flow analysis**, con cui si analizza come il valore delle variabili scorre in un programma, per identificare codice non eseguibile, variabili utilizzate senza dichiarazione; mentre la CFA permette di rilevare loop e codice irraggiungibile, questa permette di capire quale dichiarazioni siano influenzate da altre, mettendo in luce le relazioni def-use
- **program slicing**, che prevede l'individuazione di un criterio di slicing <p, v>, dove p é un punto del programma e v é un sottoinsieme della variabili del programma; una slice per un certo criterio di slicing é una porzione del programma che preserva il comportamento del programma rispetto al criterio di slicing
- **visualizzazione**, che é una strategia per mezzo della quale un sistema software è rappresentato tramite di un oggetto visivo per ottenere una certa comprensione di come il sistema è stato strutturato; distinguiamo rappresentazione di un singolo componente tramite supporti grafici, e visualizzazione che rappresenta una configurazione di un insieme correlato di rappresentazioni
- **metriche**, che servono per monitorare il processo di engineering, e si utilizzano WMC, RFC, LCOM, CBO, DIT e NOC

Un **decompilatore** prende un file binario eseguibile e tenta di produrre un codice sorgente ad alto livello leggibile. L'output, in generale, non sarà lo stesso del codice sorgente originale, e potrebbe anche non essere nello stesso linguaggio. Il decompilatore non fornisce le annotazioni dei programmatori originali, che forniscono informazioni molto importanti relativamente al funzionamento del software. I **disassembler** sono programmi che prendono il binario eseguibile di un programma come input e generano file di testo che contengono il codice del linguaggio assembly per l'intero programma o parti di esso.

Il **Data Reverse Engineering (DRE)** è definito come l'uso di tecniche strutturate per ricostituire i data assets di un sistema esistente. Attraverso tecniche strutturate, vengono analizzate le situazioni esistenti e vengono costruiti modelli prima di sviluppare il nuovo sistema. Il reverse engineering di un'applicazione data-oriented, inclusa la sua interfaccia utente, inizia con il DRE.

Il recupero delle specifiche, cioè dello schema concettuale del database è noto come **database reverse engineering (DBRE)**, e serve per facilitare la comprensione ed il redocumenting di database e file dell'applicazione. Un processo DBRE si basa sull'esecuzione all'indietro della fase logica e della fase fisica. Il processo è diviso in due fasi principali:

- **data structure extraction**, dove viene effettuato un recupero dello schema del sistema di gestione dei dati, insieme a vincoli e strutture
- **data structure conceptualization**, che si divide in concettualizzazione base e normalizzazione concettuale; nella prima si estraggono i concetti semantici rilevanti da uno schema logico, preparando lo schema per l'estrazione, traducendo nomi e ristrutturando componenti, mentre nella seconda lo schema concettuale viene ristrutturato per ottenere qualità desiderate come semplicità, leggibilità, minimalità, estensibilità ed espressività.

Wrapping

Con il **Wrapping** andiamo ad incapsulare il componente legacy con un nuovo livello software che fornisce una nuova interfaccia e nascondendo la complessità del vecchio componente. Il componente wrappato viene visto come un server remoto che fornisce un servizio richiesto da un client che non conosce dettagli implementativi del server.

Il wrapper si connette quindi al client, per quanto riguarda gli **input**, il wrapper accetta richieste da parte del client che usa il LIS, le ristruttura poiché devono essere in una forma che sia utilizzabile da parte del LIS e invoca i target object con i componenti strutturati. Per gli **output** wrapper cattura gli output dell'oggetto remoto e li confeziona in modo tale che possano essere usati dal client.

Abbiamo diversi tipi di wrapper:

- **database wrappers**, che forniscono un'interfaccia per interagire con un database, fornendo astrazione dalla sua implementazione e si dividono in:
  - **forward wrapper** che consiste nel tradurre le query in dati compatibili con il legacy system, quindi attuabile quando un nuovo componente deve accedere ai dati gestiti dal LIS
  - **backward wrapper** che consiste nella migrazione dei dati e l'implementazione del nuovo sistema che accederà al nuovo database, quindi quando è il LIS che deve accedere ai dati salvati sul nuovo sistema.
- **system service wrappers**: si ha un LIS che fornisce servizio, abbiamo quindi un adapter che mi permette di usare il servizio del LIS
- **application wrappers**: si va a wrappare un intero processo batch, per permettere di invocare i componenti del LIS come oggetti. Questo si fa in quanto i processi batch non hanno un'interfaccia, ma vengono lanciati da linea di comando, per produrre report o files aggiornati.
- **function wrappers**, che incapsulano funzioni o metodi esistenti, fornendo un'interfaccia alternativa o estesa per utilizzarli, garantendo l'accesso solo al metodo o ai metodi incapsulati.

Abbiamo diversi livelli di granularità, dove ogni livello offre un diverso grado di astrazione e coinvolgimento dei componenti del sistema:

- **livello di processo**: Questo livello fa riferimento all'applicazione batch. Input e output sono scritti in un file con cui il programma batch comunica. Il wrapper deve passare gli input ad un file che sono richiesti in input al processo. Quest'ultimo produce degli output che sono poi presi dal wrapper e che vengono passati all'applicazione client.
- **livello di transazione**: il wrapper va a simulare uno user terminal. Tale livello richiede la modifica del codice del programma legacy per fare in modo di rigirare tutta la I/O verso il wrapper invece che verso il terminale
- **livello di programma**: in cui devo avere delle API che permettono al wrapper di invocare e di passare i parametri al programma. Nel caso in cui però il formato dei dati sia differente dal formato che il LIS vuole, bisogna prendere i dati e convertirli. In aggiunta l'output del programma è preso dal wrapper e riformattati, facendoli tornare al loro tipo originale per poi inviarli al client.
- **livello di modulo**: i moduli legacy sono eseguiti usando la loro interfaccia standard, con il wrapper che bufferizza i valori passati. Il concetto è che il client ed il modulo legacy si trovano in spazi di indirizzamento diversi. In tal caso bisogna passare l'oggetto non per riferimento ma per valore. Come prima cosa quindi il wrapper bufferizza i valori ricevuti nel proprio address

space. Successivamente tali valori sono passati al modulo invocato. L'output del metodo invocato viene poi passato al client

- **livello di procedura:** le procedure interne del sistema legacy sono invocate con un'interfaccia dedicata e inizializzate prima della chiamata. É come se volessi estrarre un pezzo di codice dal LIS ed incapsularlo in un altro programma. Facendo ciò, devo fare data flow analysis per ricostruire l'interfaccia di quel pezzo di codice, definendo **input**, ovvero variabili usate in quel pezzo di codice ma non definite al suo interno; e **output**, ovvero variabili definite che però non vengono usate in quel pezzo di codice.

Posso inoltre adattare il programma che devo wrappare, in modo tale che il programma possa continuare a funzionare normalmente. I programmi sono adattati con tools, raramente a mano. I tipi di tool raccomandati sono: transaction wrapper, program wrapper, module wrapper e procedure wrapper.

Un sistema legacy è wrappato in tre passi:

- viene costruito il wrapper
- il programma target viene adattato
- vengono verificate le interazioni fra il programma target e il wrapper

Il wrapper prende quindi input dal client, li trasforma in input validi per il target program e li invia. Intercetta poi gli output, ritrasformando i dati e li invia al client. Se i linguaggi di programmazione sono diversi, realizzare un wrapper potrebbe risultare complesso. Concettualmente, un wrapper comprende:

- **Interfaccia esterna**, permette al client di accedere all'interfaccia del wrapper. Se il wrapper è scritto in un linguaggio diverso, rispetto al programma client, si può aggiungere un altro strato che è scritto nello stesso linguaggio del programma client, in modo tale da nascondere la tecnologia con cui è realizzato il wrapper. Il wrapper deve anche fungere da gestore della concorrenza in quanto un LIS potrebbe essere invocato da più programmi, dovendo quindi gestire le chiamate di diversi client.
- **Interfaccia interna**, visibile al server. Se il wrapper può invocare direttamente l'oggetto server per prendere i risultati, la chiamata non avviene dal componente LIS al wrapper ma al contrario. Se quindi il LIS offre un servizio, sarà il servizio ad essere invocato.
- **gestore messaggi**, essendoci concorrenza bisogna serializzare. Per serializzare si ha un gestore dei messaggi che bufferizza messaggi di input e output. Se si ha invece un componente che gestisce già la concorrenza il gestore dei messaggi non serve più a tale scopo. Dovendo wrappare un qualcosa che si trova su pc, si potrebbe usare una driving cleaning library, ovvero oggetto che gestisce la concorrenza. Invocando quindi un oggetto, quest'ultimo viene invocato mediante meccanismo che non gestisce la concorrenza, è necessario inserire il gestore dei messaggi per gestirla.
- **convertitore d'interfaccia**, si occupa di convertire i parametri dal client al LIS e viceversa.
- **emulatore I/O**, che intercetta gli input, riempendo il buffer di ingresso con i valori dei parametri dell'interfaccia interna, e gli output, copia il contenuto del buffer in input nello spazio dei parametri dell'interfaccia esterna

Altro modo per definire il wrapping di un programma, passando da text-based a GUI in cui si continua ad usare quindi un LIS mediante user interface. L'idea è di estrarre mediante reverse engineering il modello dell'interfaccia utente, analizzando tutte le screen sections. Si creava poi una nuova GUI che poteva essere rengineerizzato in una web app

Migrazione

Il wrapping viene usato quando si vuole fare migrazione. Si parte da un software esistente che non evolve più e lo si incapsula all'interno di un nuovo sistema, mantenendone le funzionalità originali. Questo processo richiede il reengineering del sistema per consentirne la migrazione verso una nuova tecnologia. La migrazione va a sostituire il sistema originariamente sviluppato con una tecnologia specifica con uno basato su una tecnologia diversa.

Durante questa operazione, è possibile migrare il database, trasferendo i dati in esso contenuti, e i programmi, adattandoli alla nuova architettura. Successivamente, si procede con la migrazione delle componenti, sostituendo gradualmente quelle legacy con componenti nuove e più moderne. In questo modo, il sistema si evolve senza perdere continuità operativa (approccio incrementale).

La migrazione comprende diversi passi principali:

- **Schema conversion**: la conversione dello schema del database legacy in una struttura di database equivalente espressa nella nuova tecnologia. La trasformazione dal database dello schema sorgente al target schema è composta da due processi:
  - il primo è chiamato DBRE, il cui obiettivo è di effettuare il recovery dello schema concettuale che esprime la semantica della source data structure
  - il secondo processo è semplice e deriva lo schema fisico di destinazione da questo schema concettuale
- **Data Conversion**: Per conversione dei dati si intende lo spostamento delle istanze di dati dal database legacy al database di destinazione. La conversione dei dati richiede tre passaggi: estrazione, trasformazione e caricamento (ETL). Innanzitutto, si estraggono i dati dall'archivio legacy. In secondo luogo, si trasformano i dati estratti in modo che le loro strutture corrispondano al formato. Inoltre, si esegue la pulizia dei dati (ovvero scrubbing o pulizia) per correggere o eliminare i dati che non si adattano al database di destinazione. Infine, si trasportano i dati trasformati nel database di destinazione
- **Program Conversion**: la modifica di un programma per accedere al database migrato anziché ai dati legacy. Il processo di conversione lascia invariate le funzionalità del programma. La conversione del programma dipende dalle regole utilizzate per trasformare lo schema legacy nello schema di destinazione

Se migro completamente il sistema legacy, posso usarlo come sistema per verificare il nuovo sistema quando faccio regression testing. Posso quindi usare l'oracolo che viene dal LIS.

Un'interruzione per passare dal vecchio al nuovo sistema è necessaria, e in tal senso abbiamo tre strategie di transizione:

- **Cut-and-Run**, con cui si spegne il sistema legacy e si accende il nuovo sistema
- **Phased Interoperability**, con cui, per ridurre i rischi, il taglio viene effettuato gradualmente in passi incrementali. Quando tutto il sistema è migrato, si spegne il vecchio sistema
- **Parallel Operation**, con cui il sistema di destinazione e il LIS funzionano contemporaneamente, e una volta che il nuovo sistema è considerato affidabile, il LIS viene tolto dal servizio

Abbiamo diversi approcci di migrazione:

- **cold turkey,** strategia big bang**,** non incrementale come se facessi reengineering e poi implementassi tutto il sistema. Tale approccio è molto rischioso e del tutto sconsigliato, a meno che il LIS non è di piccole dimensioni. Infatti, il rischio del fallimento cresce con la grandezza del sistema che deve essere migrato.

- **database first**: approccio in cui vado a migrare prima il database e gli altri componenti. Migrando prima il db si ha bisogno di un forward gateway che mi consenta di accedere al database traducendo le chiamate dell'applicazione legacy alle chiamate target. Similmente, gli output del database reingegnerizzato vengono tradotte per essere usate dal sistema legacy. L'approccio è applicabile solo ad un sistema legacy completamente scomponibile, dove esiste un'interfaccia pulita con il database legacy. *Tutto ciò viene fatto mediante un processo di database reverse engineering e successiva renginerizzazione del database*.

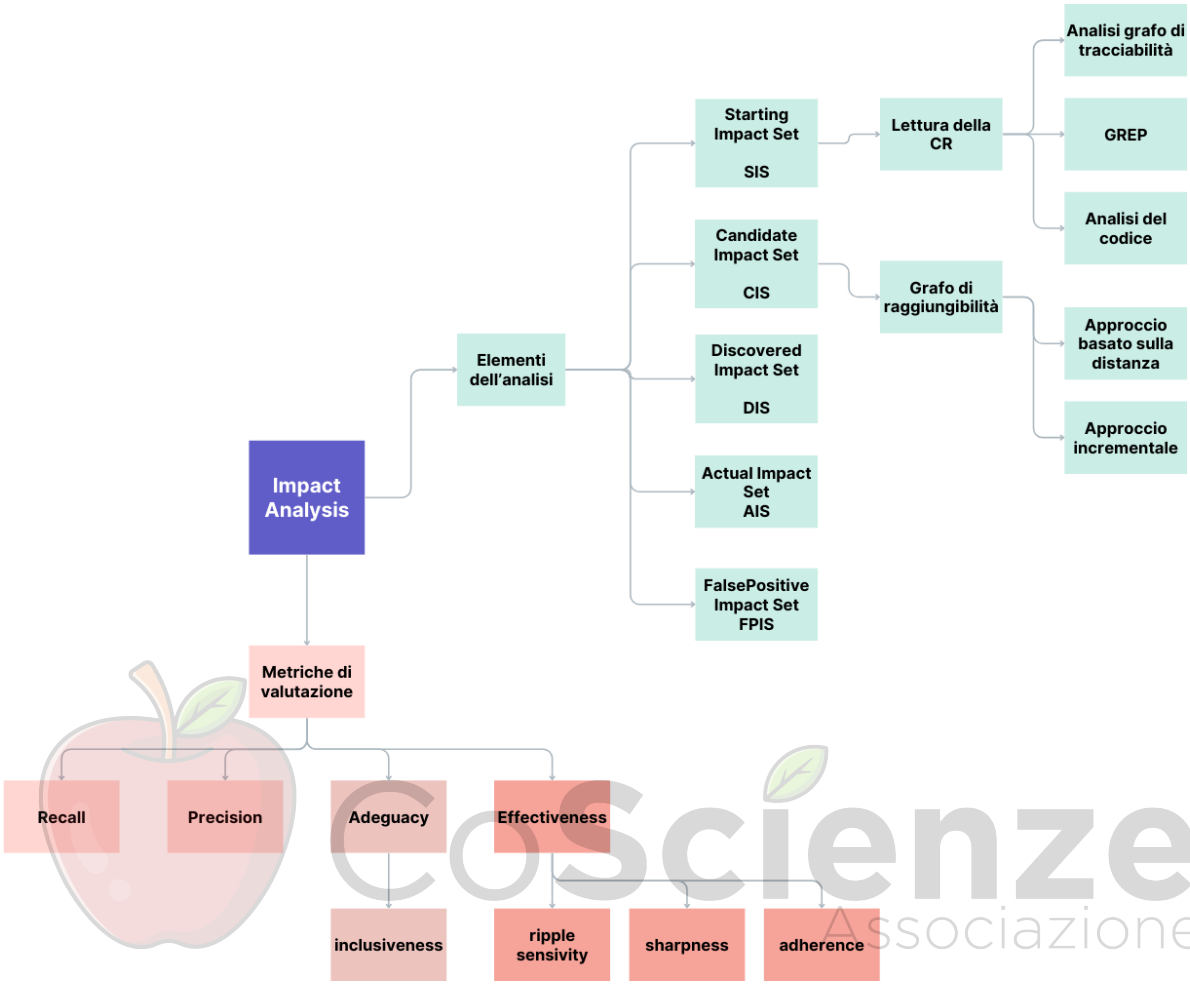
- **database last**, dove la migrazione del database avviene per ultima. Proprio per questo avremo bisogno di un reverse gateway che consentirà alla parte nuova di accedere ai legacy data. Il nuovo schema di dati non è stato ancora costruito e quindi il reverse gateway deve mappare il nuovo schema sul vecchio schema. Anche in questo caso, l'approccio è applicabile solo ad un sistema legacy completamente scomponibile, dove esiste un'interfaccia pulita con il database legacy.

- **composite database**, approccio ibrido in cui il nuovo sistema accede sia al vecchio che al nuovo database durante il periodo di transizione, essendo entrambi in esecuzione, attraverso una combinazione di reverse e forward gateway. I dati nel database potrebbero essere duplicati e per questo è presente un coordinatore di transazioni.

- **chicken little**, affina la strategia composite implementando un database gateway per i sistemi decomponibili, application gateway per i sistemi semi decomponibili ed un information system gateway per i sistemi non decomponibili, al fine di garantire l'intera funzionalità del sistema legacy

- **butterfly**, il vecchio sistema rimane operativo, e prevede l'implementazione storage temporanei per la migrazione dei dati dal vecchio al nuovo sistema

Impact Analysis



Con l'impact analysis, andiamo ad indentificare le componenti impattate dalle change request per determinare quali parti del sistema verranno impattate e se ci sono porzioni critiche; per determinare le porzioni del software che devono essere sottoposte a test di regressione (se voglio risparmiare sui costi del testing posso andare a vedere quali sono i casi di test che devo rieseguire, ovvero parti che vanno ad esercitare componenti impattate da una modifica).

Gli elementi da definire durante il processo di impact analysis sono:

- **Starting Impact Set (SIS)** insieme delle componenti impattate direttamente dalla change request, che devono essere modificate per implementare la change request.
- **Candidate Impact Set (CIS),** ovvero il SIS arricchito dai componenti che devo andare a modificare come effetto della ripple analysis, ovvero le componenti da modificare per un impatto indiretto.
- **Discovered Impact Set (DIS):** i componenti che durante la modifica sono stati scoperti come da modificare e che non sono stati individuati inizialmente.
- **Actual Impact Set (AIS):** L'insieme delle componenti effettivamente modificate.
- **False Positive Impact Set (FPIS):** Insieme dei componenti che avevo previsto si modificassero, ma non sono da modificare.

L'impact analysis inizia con **l'identificazione dello Starting Impact Set (SIS)**. Abbiamo divere alternative per identificare lo Starting Impact Set:

1. Possiamo usare la metodologia di ricognizione di Wilde e Scully, che si basa sull'idea che le feature di un programma siano selezionabili perché dipendono da una specifica sequenza di input. Quindi andiamo ad eseguire il programma una volta con le feature selezionate ed una volta senza. È probabile che il codice che è stato eseguito solo la prima volta, quindi con le feature selezionate, si trovi nel codice che implementa la feature o è vicino ad essa.
2. La prima alternativa è individuare i concetti espressi nella Change Request e mapparli sul codice, ad esempio utilizzando **grep** per individuare specifici identificatori all'interno di tutto il codice.
3. La seconda alternativa è utilizzare tecniche di **information retrival**, utilizzando la change request come query ed il codice come documento.
4. La terza alternativa, applicabile nel caso in cui si abbia una **test suite** in cui si ha un mapping tra test case e features è andare ad individuare i test case che dovrebbero esercitare il codice che implementa la feature. Quindi, divido la test suite in test cases che testano le features che hanno a che fare con la CR e test cases che non hanno a che fare con features che interessano la CR, eseguo entrambe le volte il codice, posso individuare gli statement eseguiti con i test cases che implementano le features e quelli che non implementano le features.
5. La quarta alternativa è la **tecnica incrementale in cui partiamo dal main**, vediamo se si ha una chiamata a funzione e vediamo se quella funzione è impattata. Se non è impattata non si fa nulla, mentre se è impattata si entra nella funzione e si vede se a sua volta chiama altre funzioni che potrebbero essere impattate.

Per **individuare il CIS**, possiamo utilizzate la tracciabilità per individuare gli impatti:

- **Diretto**: se c'è un link di tracciabilità tra due componenti si ha un impatto diretto
- **indiretto**: se da un componente raggiungo mediante diversi link di tracciabilità un altro componente si ha un impatto indiretto

Per sapere se una componente ha un impatto su un'altra è necessario calcolare la connectivity matrix, e da essa posso calcolare la matrice di raggiungibilità, costruita considerando gli SLO e le relazioni fra essi, e il grafo di raggiungibilità che mostra le entità che possono essere influenzate da una modifica di un SLO. Se la matrice di raggiungibilità è troppo folta, vuol dire che ho come CIS quasi tutto il sistema, ovvero avrò ChangeRate troppo elevato, per cui c'è qualcosa che non va. Per risolvere il problema posso usare:

- **Distance based**, dove gli SLO più lontani di una soglia di distanza non sono considerati nell'impatto delle modifiche

- **Incremental**, dove il CIS è costruito in maniera incrementale; in particolare, per ogni SLO nel SIS si considerano tutti gli SLO che interagiscono con esso, e solo gli SLO effettivamente impattati dalla change request sono messi nel CIS

Per valutare un approccio di impact analysis, utilizziamo le metriche di:

- **recall**, frazione degli impatti effettivi contenuti nel CIS $\frac{|\text{CIS} \cap \text{AIS}|}{|\text{AIS}|}$
- **precision**, frazione degli impatti candidati che sono effettivamente impattati $\frac{|\text{CIS} \cap \text{AIS}|}{|\text{CIS}|}$

Queste metriche vengono calcolate dopo la modifica. Le metriche successive si basano sul concetto che, se l'impact analysis non tira fuori tutto l'AIS non è buono, per cui si può ragionare in termini di precisione solo se la recall è pari ad 1. Le metriche che si basano su tale concetto sono adequacy ed effectiveness.

- **adeguacy**, la capacità di un approccio di identificare tutti gli elementi da modificare, espressa in termine di inclusiveness (1 se $AIS \subseteq$ CIS, 0 altrimenti). Un approccio inadeguato è infatti inutile, in quanto fornisce all'ingegnere della manutenzione informazioni errate. Se l'adequacy è pari ad 1, ovvero se l'approccio è adeguato, posso valutarne l'efficacia.
- **effectiveness**, che misura la capacità di un approccio di generare risultati utili alla manutenzione; si divide in:
    - **ripple sensitivity**, va a valutare l'impatto del ripple effect. In pratica considera l'insieme di oggetti direttamente impattati, chiamandoli Direct Impacted Set of Objects (DISO), ovvero il SIS. Si va successivamente a vedere quali sono i componenti impattati direttamente a causa del ripple effect, tracciando gli impatti candidati, ovvero CIS che viene chiamato Indirectly Impacted Set of Objects (IISO). La cardinalità di IISO è un indicatore del ripple effect. Più IISO è grande rispetto a DISO maggiore è il ripple effect. La metrica usata per calcolare ciò viene chiamata amplification=$\frac{|\text{IISO}|}{|\text{DISO}|}$. Vorrei che si mantenesse un rapporto da 0 ed 1 per indicare che per ogni componente modificato direttamente si modifica al più un componente indirettamente.
    - **sharpness**, capacità di un approccio di evitare di dover includere nel CIS oggetti che non devono essere modificati, espressa come $ChangeRate = \frac{|\text{CIS}|}{|\text{System}|}$. Essa non mi dice quanto è buona l'impact analysis in realtà ma mi indica quanto sta diventando complesso il sistema. Più il ChangeRate è elevato, maggiore è il numero di componenti che dovranno essere modificate. Ciò però non afferisce con l'impact analysis, ma è un problema del sistema. Di per sé non mi dà una bontà della misura dell'impact analysis.
    - **adherence**, capacità dell'approccio di produrre un CIS il più vicino possibile all'AIS, ed é espressa sotto forma di S-ratio= $\frac{|\text{AIS}|}{|\text{CIS}|}$. La S-Ratio è esattamente la precision nel caso in cui AIS è contenuto nel CIS. Tale metrica viene applicata quando l'inclusiveness è pari ad 1

Inclusiveness ed adherence possono essere calcolati solo dopo aver effettuato la modifica, mentre il change rate e l'amplification possono essere calcolate prima

Le tecniche impact analysis basate sulle dipendenze identificano l'impatto dei cambiamenti analizzando le dipendenze sintattiche, perché è probabile che le dipendenze sintattiche causino dipendenze semantiche.

Ci sono due tecniche tradizionali di impact analysis: la prima si basa sul **call graph**; la seconda si basa sul **dependency graph**.

In un **call graph** un nodo presenta una funzione, un componente o un metodo mentre un arco rappresenta una relazione in cui A invoca B. Se viene effettuata una modifica su un componente, si va a propagare l'impatto sia avanti che dietro, ovvero sui nodi che chiamano quel componente e quelli chiamati dal componente. Se si ragiona così però si avrebbe un impact set impreciso. Si dovrebbe quindi considerare anche la parte dinamica, ovvero quando chiamo un nodo, chi effettivamente viene chiamato? Si va quindi a considerare una traccia di esecuzione. Mediante tale traccia possiamo individuare quali sono le procedure che sono indirettamente o direttamente invocate, ma anche le procedure che sono invocate dopo che il componente termina.

Nel **program dependency graph** (PDG): ogni semplice statement è rappresentato da un nodo e ogni predicato è espresso da un nodo

Abbiamo un qualcosa di più a grana fine. Si hanno le dipendenze. In tal caso il PDG non ha nodi regione e le dipendenze sono al contrario. Se si vuole calcolare il backward slicing le dipendenze vanno seguite secondo il loro verso e non al contrario. Tecniche che possono essere usate sono:

- static program slice: tecnica più imprecisa. Per una variabile var al nodo n, è possibile identificare tutte le definizioni che raggiungono var. Trova tutti i nodi nel PDG che sono raggiungibili da quei nodi. I nodi visitati nel processo di attraversamento costituiscono la fetta desiderata.
- dynamic program slice: più efficiente nel localizzare errori,

Un argomento relativo all'impact analysis è l'analisi del ripple effect, che si ha quando una modifica a una singola variabile può richiedere la modifica di diverse parti del sistema software. L'analisi del ripple effect rivela cosa cambia e dove si verificano i cambiamenti. Un modo per mantenere tracciabilità orizzontale e mostrare la relazione fra work product é l'analisi del grafo di tracciabilità, dove ci sono diversi rettangoli in cui é rappresentata la **tracciabilità verticale** (rappresentata tramite frecce al suo interno) del prodotto rappresentato, mentre le frecce tra rettangoli rappresentano la **tracciabilità orizzontale**. Per un nodo i in un grafo, il suo grado in entrata in(i) rappresenta il numero di archi entranti e quindi il numero di nodi che hanno un impatto diretto su i. Similarmente, il grado in uscita out(i), è il numero di archi uscenti da i.

Misura e Scala

Una misura è l'attribuzione di un numero (o di un simbolo) ad un attributo di una entità per caratterizzarlo. La misura diretta di un attributo non dipende da altri attributi, altrimenti parliamo di misure indirette. Le misure rappresentano un aspetto cruciale in tutte le branche dell'ingegneria tranne che nell'ingegneria del software, in quanto si parla di proprietà come usabilità, affidabilità, e manutenibilità senza spiegare come possano essere misurate, e le poche misure che vengono fatte lo sono in maniera inconsistente.

La teoria rappresentazionale della misura fornisce un quadro di riferimento rigoroso che permette di valutare la bontà di una misura, in particolare definendo i concetti di misura, scala, trasformazione di scala.

Un **sistema di relazioni empiriche** è costituito dalle entità (E) ed un insieme di relazioni (Re) che legano tra loro le entità.

Un **sistema di relazioni numeriche** è composto da un insieme di simboli (N) e da un insieme di relazioni (Rn) che legano tra loro tali simboli.

Dato un sistema di relazioni empiriche e un sistema di relazioni numeriche, si definisce **misura** una relazione M tra i due sistemi tale che ad ogni entità è associato un numero, ad ogni relazione tra entità corrisponde una relazione tra numeri. L'esistenza di una relazione tra istanze di entità implica l'esistenza della corrispondente relazione tra i numeri (misure) associate a quelle istanze di entità.

Dato un sistema di relazioni empiriche Σ, un sistema di relazioni numeriche Γ, e una misura M, si definisce scala la terna (Σ, Γ, M).

Definiamo trasformazione di scala ammissibile una funzione che trasforma una qualsiasi scala in un'altra, e il tipo di una scala è determinato dall'insieme di trasformazioni ammissibili.

- **Scala nominale (M' = F(M))**: (es. label, classificazioni) Ogni attributo è classificabile mediante un insieme di possibili valori**.** Non c'è alcuna relazione tra gli elementi della scala, per cui l'unica trasformazione che posso fare è una trasformazione biettiva (es. cambiare la label)
- **Scala ordinale (M' = F(M))**: (es. preferenza, test QI). Ogni attributo caratterizzato da un ordinamento lineare dei valori e l'unica trasformazione ammessa è una funzione monotona, ovvero una trasformazione che mi mantiene l'ordinamento tra i valori
- **Scala intervallo (M' = aM+b):** (es. temperature). abbiamo un ordinamento dei valori in cui sappiamo anche quanto sia la distanza relativa tra i valori. Distanza relativa in quanto la trasformazione ammissibile (traslazione) può cambiare la distanza, ma lascia inalterata il rapporto tra le distanze.
- **Scale rateo (M' = aM):** (es. durate, lunghezze). Se esiste anche un elemento "zero", considerato come zero assoluto e dove non possiamo avere valori negativi. In tal caso si può parlare di rapporti tra misure e tutte le trasformazioni tra scala fanno in modo che il rapporto tra le misure sia invariabile.
- **Le scale assolute** derivano da attributi che danno luogo ad un semplice conteggio di entità e non possiamo fare alcuna trasformazione in quanto il conteggio è quello e non posso cambiarlo

Il tipo di scala di una misura determina l'insieme di operazioni ammesse. Scale nominali e ordinali ammettono test statistici non parametrici: percentile e mediana; Scale intervallo e rateo ammettono test statistici parametrici: media, deviazione standard e media geometrica.

Testing

L'IEEE 829-2008 definisce il testing un'attività in cui un sistema o un componente è eseguito sotto specificate condizioni per osservare, memorizzare i risultati e fare delle valutazioni su alcuni aspetti del sistema o del componente.

Un test è formato da un insieme di casi di test, definito come test suite. Un test ha successo se rivela uno o più malfunzionamenti di un programma. La condizione necessaria per effettuare un test è conoscere il comportamento atteso per poterlo confrontare con quello osservato. L'oracolo conosce il comportamento atteso per ogni caso di test e può essere:

- **umano**: che si basa sulle specifiche o sul giudizio, ad esempio quando si inseriscono le assert in un programma, l'oracolo è comunque umano in quanto l'assert è scritto da noi, ma è automatico il confronto con l'oracolo. Si potrebbe avere però un'automatizzazione nella generazione dell'oracolo e relativo controllo
- **automatico**: generato dalle specifiche formali, dalla versione precedente o dal software stesso ma sviluppato da altri. É buona norma avere un oracolo memorizzato per i dati di test, riuscendo ad automatizzare i casi di test. Avere degli oracoli automatici ai fini dell'esecuzione è ottimale per risparmiare tempo.

Il testing serve ad aumentare la fiducia nel programma. Mi fermo quando il livello di fiducia è tale per quelli che sono gli obiettivi del programma. Un programma è testato a sufficienza seguendo:

- **criterio temporale**
- **criterio di costo**;
- **criterio di copertura**, nel caso in cui stiamo facendo test sistematico.
- **criterio statistico**: ad esempio prendo il tempo medio tra le failure predefinito e lo confronto con un modello di affidabilità esistente. Voglio che questo tempo aumenti in quanto più aumenta più è affidabile il sistema.

Un **test è ideale** se l'insuccesso del test implica la correttezza del programma. Poniamoci di trovare il test ideale, ovvero un insieme di casi di test tale che, se tutti falliscono, per cui non rilevano malfunzionamenti, io posso dire che il programma è corretto.

Un **test esaustivo** contiene tutti i casi di test di un programma, per cui è un test ideale. Esso però è impraticabile e quasi mai fattibile. Bisogna quindi cercare di approssimare un test ideale, cercando di massimizzare il numero di malfunzionamenti rilevati e minimizzando il numero di casi di test tramite un criterio di selezione dei test.

Un **criterio di selezione è affidabile** se per ogni coppia di test selezionati, T1 e T2, dove entrambi sono insiemi di casi di test, se T1 ha successo anche T2 ha successo e viceversa. In pratica un criterio di selezione di casi di test è affidabile o se tutti i casi di test generati riescono a rilevare malfunzionamenti o nessuno riesce.

Un **criterio di selezione è valido** per un programma se, qualora il programma non sia corretto, esiste almeno un test selezionabile mediante il criterio di selezione che consente di rilevare malfunzionamento.

Alcune tecniche di testing sono: testing sistematico, testing statistico e analisi mutazionale.

Analisi mutazionale: tecnica che deriva dal testing dell'hardware e serve a generare programmi alternativi, chiamati mutanti. Si fa testing ma non si rilevano malfunzionamenti. Per valutare se la test suite è buona si vanno ad inserire difetti nel codice. Se la test suite non è in grado di rilevare malfunzionamenti iniettati, allora la test suite non è buona, per cui deve essere arricchita con casi di test che vanno a rilevare quel malfunzionamento. Alternativamente, la test suite è in grado di uccidere il mutante.

Il testing statistico è un testing quasi random. Non è completamente random poiché predilige i casi di test che vengono eseguiti più frequentemente, in base alla frequenza di utilizzo di certi dati da parte di un utente. C'è quindi bisogno di costruire un modello dell'utilizzo del sistema da parte dell'utente per generare questi casi di test.

Il testing random (o uniforme) è un approccio casuale, in quanto non tiene conto della frequenza o dell'importanza di certi dati di input. Ogni dato di test ha la stessa probabilità di essere selezionato, il che potrebbe risultare appropriato solo se tutti gli input avessero uguale rilevanza. Tuttavia, nella realtà non è così in quanto tutti gli input non hanno la stessa probabilità di causare malfunzionamenti, ed i casi di test dovrebbero essere selezionati in modo da massimizzare le probabilità di rilevare errori

Quando non è possibile determinare quali input siano più probabili nel rilevare malfunzionamenti, si adotta un **approccio sistematico** basato sul partizionamento dello spazio di input. Gli input vengono suddivisi in classi (o cluster), in cui il programma si comporta in modo simile. All'interno di ciascuna classe, tutti gli input hanno la stessa probabilità di individuare un malfunzionamento (e varie tra le classi). Scegliendo un rappresentante per ogni classe, si ottiene un criterio di testing più valido ed efficace.

Questo approccio risolve il limite del testing casuale, che può non rilevare errori nascosti in specifiche regioni dello spazio di input. Ad esempio, se un sistema fallisce solo con valori estremi, un test casuale potrebbe non includere abbastanza campioni rappresentativi. L'obiettivo è distribuire in modo efficace i casi di test, garantendo una buona copertura ed avremo:

- **Partizionamento completo**: i cluster sono disgiunti e devono essere tutti testati, ma il costo è elevato.
- **Quasi-partizionamento**: i cluster possono sovrapporsi, permettendo di scegliere meno casi di test.

Il criterio di partizionamento dipende dal tipo di testing:

- **Black box testing**: il partizionamento si basa sulla specifica del sistema. È utile per identificare funzionalità non implementate ma presenti nella specifica.
- **White box testing**: il partizionamento si basa sul codice sorgente. Permette di individuare parti implementate ma non specificate.

L'obiettivo finale è definire un criterio di copertura che assicuri test rappresentativi per ogni cluster, ottimizzando il rapporto tra costi e affidabilità.

Blackbox

Nel testing funzionale la definizione dei casi di test e dell'oracolo è fondata sulla base della sola conoscenza dei requisiti del sistema e dei suoi comportamenti. Quindi, a livello di sistema userò la conoscenza dei requisiti funzionali o dei casi d'uso, mentre a livello di unità userò la conoscenza o la specifica dei moduli, come precondizioni e postcondizioni. Le stesse tecniche black box a livello di unità possono essere applicate a livello di sistema.

Per i criteri di copertura per il testing funzionale bisogna:

- **eseguire almeno una volta ogni funzionalità**;
- **per ogni funzionalità decidere quanti casi di test selezionare**. Ciò può essere dedotto dalla specifica, dai dati di ingresso, precondizioni e postcondizioni.

Suddivisione in classi di equivalenza

Sia nel testing **black box** che in quello **white box**, l'obiettivo è partizionare lo spazio di input in cluster in cui tutti gli input, o nessuno, rilevino un malfunzionamento. Poiché definire un criterio di selezione completamente affidabile e valido è indecidibile, si adotta una soluzione approssimata.

Nel metodo delle classi di equivalenza, la specifica del programma viene utilizzata per suddividere il dominio degli input in classi, ciascuna rappresentante un insieme di valori con comportamento uniforme. Se il programma si comporta correttamente per un caso di test di una classe, si presume che funzioni per tutti i casi in quella classe.

Ogni combinazione di classi di equivalenza rappresenta una partizione dello spazio di input. Tuttavia, per garantire una copertura completa, non basta analizzare i parametri singolarmente: è necessario considerare le combinazioni delle classi di equivalenza di tutte le variabili.

Criteri di selezione

Le classi di equivalenza non rappresentano una partizione completa dello spazio di input, a meno che non ci sia un'unica variabile. Quando lo spazio di input è formato da più variabili, ogni classe combina valori per ciascuna variabile, rendendo necessario incrociare le classi per coprire tutte le possibili interazioni. Esistono due principali criteri di selezione dei casi di test basati sulle classi di equivalenza:

1- Weak Equivalence Class Testing

Nel Weak si seleziona un valore rappresentativo per ogni classe di equivalenza di ciascuna variabile e li combina in un unico caso di test. In questo modo, un singolo caso può coprire più classi di equivalenza contemporaneamente.

Il numero minimo di casi di test richiesti corrisponde al numero massimo di classi di equivalenza in cui il dominio è suddiviso. Sebbene si possano utilizzare più casi di test, l'obiettivo è minimizzarne il numero, garantendo che ogni classe sia coperta almeno una volta.

Nel Weak si parla di **quasi partizionamento**, poiché alcuni input possono appartenere a più cluster. Questi input vengono trattati con particolare attenzione, ma il criterio punta, comunque, a una copertura efficiente, sfruttando combinazioni che includano contemporaneamente più classi.

2- Strong Equivalence Class Testing

Questo criterio copre tutte le possibili combinazioni tra le classi di equivalenza, generando un **partizionamento completo** dello spazio di input. In questo caso, un input che appartiene a un cluster non può appartenere ad altri.

Il SECT richiede un numero di casi di test molto elevato, calcolato come il prodotto delle cardinalità delle classi di equivalenza di tutte le variabili. Sebbene questo approccio garantisca una copertura completa, risulta oneroso in termini di complessità e numero di test da eseguire.

Più preciso è il partizionamento dello spazio di input, maggiore sarà il numero di casi di test richiesti. Un partizionamento competo, come quello ottenuto con il criterio forte, richiede più test rispetto a un quasi partizionamento, come nel caso del criterio debole.

Quando le condizioni di errore sono prioritarie, il SECT deve includere anche le classi di input non valide, un aspetto cruciale nei domini discreti, negli intervalli o negli insiemi di valori, dove il metodo delle classi di equivalenza è particolarmente efficace. Tuttavia, il SECT assume che le variabili siano indipendenti: se esistono dipendenze tra le variabili, possono emergere casi di test non validi ("error" test case), in cui una combinazione valida di classi per singole variabili risulta complessivamente non valida, violando le precondizioni.

Infine, il metodo delle classi di equivalenza non distingue nettamente tra test di robustezza e test di correttezza, poiché alcuni test di robustezza contribuiscono anche alla verifica della correttezza. Questo riflette la complessità di garantire una copertura completa senza sacrificare la qualità dell'analisi.

Selezione delle classi di equivalenza

Il metodo delle classi di equivalenza è appropriato quando i dati di input sono definiti in termini di intervalli e insiemi di valori discreti. Per la selezione delle classi di equivalenza, se la condizione sulle variabili d'ingresso specifica:

- **un intervallo di valori**, allora si avrà almeno una classe valida per valori interni all'intervallo, inferiori al minimo, e i superiori al massimo
- **un elemento di un insieme discreto**, allora si avrà una classe valida per ogni elemento dell'insieme e una non valida per un elemento non appartenente;

Le classi di equivalenza individuate devono essere utilizzate per identificare casi di test che minimizzino il numero complessivo di test e risultino significativi (affidabili). Si devono individuare tanti casi di test da coprire tutte le classi di equivalenza valide, con il vincolo che ciascun caso di test comprenda il maggior numero possibile di classi valide ancora non coperte. Analogamente si devono individuare tanti casi di test da coprire tutte le classi di equivalenza non valide, con il vincolo che ciascun caso di test copra una ed una sola delle classi non valide.

Testing dei valori limite

Dei tipici errori di programmazione capitano al limite tra classi diverse, e a tal proposito il testing dei valori limite si focalizza su questo aspetto, che è più semplice e serve a complementare la tecnica precedente. L'idea di base è di verificare i valori della variabile di input al minimo, immediatamente sopra il minimo, un valore intermedio (nominale), immediatamente sotto il massimo, e al massimo (per convenzione min, min+, nom, max-, max). In questo modo, una funzione con n variabili richiede 4n + 1 casi di test.

Metodi per ridurre i casi di test

Il Weak Equivalence Class Testing (WECT) rappresenta il caso limite con il minor numero di casi di test. Tuttavia, questo approccio presuppone l'assenza di interazioni tra le classi di equivalenza, il che può risultare irrealistico in molti contesti.

Dall'altra parte, lo Strong Equivalence Class Testing (SECT) considera che gli errori possano derivare da interazioni tra le classi, testando tutte le possibili combinazioni. Questo approccio, però, assume anche che i valori non validi siano indipendenti dalle interazioni tra classi, una condizione che spesso non è vera.

L'obiettivo è quindi ridurre il numero di combinazioni, concentrandosi sui casi di test più significativi. In alcuni scenari, il valore di una singola variabile può determinare l'esito del test, ma con il SECT tale valore verrebbe comunque combinato con tutti gli altri, aumentando inutilmente il numero di test. Diventa quindi necessario un metodo che permetta: di eseguire una determinata classe solo una volta; oppure di scegliere con quali altre classi testarla. Tra le tecniche utili per ridurre i casi di test si includono: **Category Partition**, Tabelle delle decisioni, Grafi causa-effetto, Pairwise Combinational.

Category Partition

Il Category Partition ci consente di individuare le funzioni da testare e per ciascuna funzionalità, vengono identificati i parametri e gli oggetti dell'ambiente con cui la funzione interagisce, poiché insieme ai parametri di input determinano l'output. Quindi un vantaggio del Category Partition è che prende in considerazione anche gli oggetti dell'ambiente, cosa che l'Equivalence Class Testing non fa. Inoltre, consente di individuare vincoli tra le scelte tramite proprietà e selettori.

Rispetto all'Equivalence Class Testing, che si limita a partizionare il dominio in classi di equivalenza per valori scalari (ad esempio, range o enumerazioni), Category Partition permette di analizzare domini più complessi, inclusi quelli strutturali. La tecnica si concentra sulle caratteristiche dei parametri e crea categorie che descrivono il comportamento degli input. Le categorie vengono poi suddivise in scelte, che rappresentano valori normali, speciali o limite.

Gli step da fare per il category partition sono:

1. **Individuazione delle categorie:** si identificano le categorie basandosi sulle specifiche del sistema e sulle precondizioni. Ogni configurazione di input può essere vista come un sottoinsieme con caratteristiche comuni, definite come categorie.
2. **Individuazione delle scelte**: per ogni categoria, si identificano le classi rappresentative di valori, ignorando inizialmente le interazioni tra le categorie. Le scelte possono includere valori normali, speciali o limite.
3. **Introduzione dei vincoli**: dopo aver individuato le combinazioni di scelte, si introducono vincoli per eliminare combinazioni impossibili e ridurre il numero di casi di test. Questi vincoli includono:
    a. **Error constraint**: elimina combinazioni con più di una proprietà ERROR. Una scelta con proprietà ERROR indica un valore invalido e può interrompere l'esecuzione del programma senza richiedere ulteriori parametri. Tuttavia, viene spesso combinata con scelte valide per verificare se il sistema riconosce l'errore. Una scelta ERROR deve essere combinata con una sola classe valida.
    b. **Property constraint**: raggruppa valori di uno stesso parametro con proprietà comuni.
    c. **IF-property**: Limitano le scelte di una categoria che possono essere combinate con un valore specifico di un'altra categoria.
    d. **Single constraint**: Garantisce che una scelta venga testata una sola volta, indipendentemente dalla classe con cui è combinata.

Quando proprietà e selettori vengono utilizzati per eliminare scelte incompatibili, si ottiene un partizionamento perfetto (perché testiamo tutti gli scenari) dello spazio di input, a condizione che siano rispettate le precondizioni. Se, invece, vengono usati per ridurre il numero di casi di test, si parla di un quasi partizionamento (non testiamo qualche scenario).

Vincoli e Test Frame

I vincoli sono essenziali per definire combinazioni valide di scelte. Ad esempio, se una scelta di una categoria può essere combinata solo con determinate scelte di un'altra, si applicano selettori e proprietà per limitare le combinazioni possibili. Questo riduce lo spazio di test, ottimizzando la progettazione. Le combinazioni valide, chiamate *test frame*, rappresentano gli input per i test.

Conclusioni

L'individuazione di parametri e categorie nel Category Partition dipende dall'esperienza del tester rendendo esplicite le decisioni di testing, che possono essere riviste. Una volta completato il primo passo, la tecnica diventa relativamente semplice e può essere automatizzata, rendendo il testing sistematico più praticabile e riducendo il numero di casi di test. L'obiettivo del Category Partition è identificare insiemi di input con comportamenti simili. Tuttavia, non è sufficiente partizionare direttamente il dominio degli input; bisogna anche considerare le interazioni tra i parametri. Pertanto, è fondamentale individuare le categorie che definiscono sottoinsiemi di input con comportamento simile. Queste categorie, e le relative scelte, permettono di partizionare adeguatamente lo spazio di input. Il Category Partition consente di progettare test con flessibilità, modificando vincoli, proprietà e selettori per ottimizzare il numero di casi da testare. La scelta di combinare o meno determinate scelte è una decisione progettuale, che consente di ridurre ulteriormente i casi di test o di aumentarne la copertura.

Whitebox

Le tecniche di testing strutturale, basate sulla conoscenza della struttura del software ed in particolare del codice, sono fondate sull'adozione di criteri di copertura degli oggetti che compongono la struttura dei programmi. Con copertura si intende la definizione di un insieme di casi di test in modo tale che gli oggetti di una classe siano attivati almeno una volta nell'esecuzione dei casi di test. Un criterio di copertura è un insieme di test case che assicuri l'attraversamento almeno una volta di almeno un cammino per ogni classe. Definiamo con **Test Effectiveness Ratio (TER)** il rapporto fra oggetti coperti e oggetti totali. Costituisce una metrica di copertura, calcolata rispetto a line coverage, branch coverage, condition coverage e path coverage.

Una strategia per effettuare testing whitebox, che risulta essere troppo costosa, è dove ogni caso di test corrisponde all'esecuzione di un particolare cammino sul CFG di un programma P. Quindi bisogna individuare i cammini che ci garantiscono il livello di copertura desiderato. Viene poi effettuata l'esecuzione simbolica di ogni cammino e vengono individuate degli input data che soddisfano la path condition.

L'alternativa realizzabile è individuare i casi di test in accordo al criterio di copertura funzionale e lo si esegue, andando a misurare la copertura strutturale. Se abbiamo raggiunto la soglia, ovvero la **test effectiveness ratio**, ci fermiamo, altrimenti è necessario trovare altri casi di test che ci consentono di raggiungere la copertura desiderata, applicando tecniche white box e vedendo quali sono le parti di codice che non sono state coperte, trovando un cammino che esegue quelle parti di codice e quindi trovare un input che esegue quel cammino, facendo esecuzione simbolica.

Statement coverage (copertura degli statement)

Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i **nodi** del CFG, ovvero l'esecuzione di tutte le **istruzioni di input**. Statement coverage e node coverage sono quindi la stessa cosa. Per eseguire tutti gli statement potrebbe bastare un solo cammino, in base al tipo di problema. $TER = \frac{\text{numero statement eseguiti}}{\text{numero statement totali}}$

Branch coverage (copertura delle decisioni)

Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i **rami** del CFG, ovvero l'esecuzione di tutte le **decisioni**. Quindi per ogni decisione eseguo il ramo vero ed il ramo falso. Per ogni decisione sono necessari due casi di test, uno che fa diventare vera la condizione e una che la fa diventare falsa. Se copro tutti i branch compro tutti i nodi, per cui **branch coverage implica statement coverage**. Non è garantito trovare la failure poiché le decisioni sono composte da più condizioni in AND o OR e in alcuni casi branch coverage non è sufficiente. Possono esserci più modi diversi per coprire le decisioni.

Esso infatti permette un quasi-perfezionamento. $TER = \frac{\text{numero branch eseguiti}}{\text{numero branch totali}}$

Copertura delle decisioni

Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'esecuzione di tutte le condizioni (valori vero e falso delle componenti relazionali dei predicati) caratterizzanti le decisioni di P. Significa che per ogni condizione vogliamo che ci siano almeno due casi di test, una che la fa diventare vera ed una falsa. Non si coprono le decisioni, ovvero il predicato della struttura di controllo, ma le singole condizioni che si trovano all'interno del predicato. La copertura delle condizioni non implica la copertura delle decisioni.

Copertura di decisioni e condizioni

Dato un programma P, si definisce un insieme di test case che garantisce l'esecuzione di tutte le decisioni e delle relative condizioni. Esistono due tipi principali:

- **multiple condition coverage (MCC)**: ogni decisione viene testata considerando tutte le possibili combinazioni delle sue condizioni. Il numero totale di combinazioni è 2^n, dove *n* è il numero di condizioni.
- **modified condition decision coverage (MCDC):** si considerano solo le combinazioni di valori delle condizioni in cui il risultato della decisione viene influenzata da una singola condizione, mentre le altre rimangono costanti. Per farlo, si costruisce la tabella di verità delle condizioni e, per ciascuna di esse, si individuano due combinazioni in cui il cambiamento del valore della condizione modifica il valore della decisione, lasciando invariati gli altri fattori. Questo permette di identificare i test case che rendono una condizione vera o falsa e che, di conseguenza, determinano direttamente il risultato della decisione. Questo processo viene applicato sistematicamente a tutte le condizioni presenti.

Il numero di test case richiesti per l'MCDC è inferiore rispetto al MCC, ma garantisce comunque una copertura significativa.

Path coverage (copertura dei cammini)

La **path coverage** (copertura dei cammini) si basa sulla definizione di un insieme di test case che garantiscano l'esecuzione di tutti i cammini possibili nel grafo di flusso di controllo (CFG) di un programma PPP. Tuttavia, gestire i cammini in presenza di cicli può essere complesso, poiché ogni variazione nel numero di iterazioni di un ciclo genera un cammino differente. Questo può portare a un numero molto elevato, o addirittura infinito, di cammini possibili.

Un problema comune è la presenza di *infeasible paths*, ossia cammini per i quali non esiste alcun input che permetta di percorrerli. Inoltre, se il numero di cammini è infinito, ciò è tipicamente dovuto alla presenza di circuiti o iterazioni nel programma.

Per affrontare queste difficoltà, si utilizzano tecniche che limitano il numero di cammini da considerare. Tra queste, il criterio di **n-copertura dei cicli**, i metodi basati sugli **exemplar-paths**, i cammini **linearmente indipendenti** (definiti dall'indice di McCabe) e le tecniche basate sul **data-flow**.

La path coverage è considerata una tecnica di analisi molto dettagliata, poiché fornisce un partizionamento perfetto del programma, ma risulta complessa da applicare su scala elevata. La misura di efficacia dei test si calcola in $TER = \frac{\text{numero cammini eseguiti}}{\text{numero cammini totali}}$

Si riesce ad ottenere un partizionamento perfetto dello spazio di input mediante il path coverage, poiché i casi di test che coprono un cammino non possono coprire un altro cammino. In generale, per le altre coperture, l'insieme di cammini che copre lo statement si sovrappone all'insieme di cammini che copre altri statement. Un criterio che mi chiede più casi di test è quello che partiziona di più, mentre il criterio che mi chiede meno caso di test è quello che partiziona meno lo spazio di input, in quanto va a prendere casi di test che mi coprono più elementi contemporaneamente.

Criterio di copertura n cicli

Il **criterio di n-copertura dei cicli** prevede la selezione di test case per garantire che ogni ciclo deve essere eseguito da 0 a n volte. Solitamente si sceglie n=2, quindi il ciclo viene eseguito 0, 1 e 2 volte.

Quando n=1 si ottiene implicitamente anche il criterio di branch coverage. Questo perché, per soddisfare la 1-copertura, si devono generare test case in cui si entra nel ciclo una volta e altri in cui non si entra affatto. Tuttavia, la 1-copertura va oltre la branch coverage, in quanto, oltre a verificare l'accesso o meno al ciclo, si concentra anche sull'esecuzione effettiva di una singola iterazione.

Se il ciclo viene eseguito una sola volta, alternando uno stato "vero" e uno "falso", potrebbe accadere che non tutte le ramificazioni siano coperte. Per garantire la copertura del branch, occorre testare esplicitamente sia i casi in cui non si entra nel ciclo, sia quelli in cui il ciclo viene eseguito almeno una volta.

Nei sistemi complessi, il criterio di 2-copertura può non essere sufficiente. Questo perché le relazioni tra i dati o le condizioni potrebbero non essere contenute all'interno di cicli consecutivi, ma influenzarsi tra iterazioni non adiacenti. In questi casi, è necessario aumentare n per garantire una copertura più approfondita delle possibili combinazioni e relazioni.

Metodo degli exemplar path

L'insieme dei cammini del grafo di controllo viene partizionato in un numero finito di classi di equivalenza. Per il criterio di copertura bisogna trovare l'insieme di casi di test che assicuri l'attraversamento almeno una volta di almeno un cammino per ogni classe.

Trovo quindi gli exemplar path, ovvero i rappresentati delle varie classi. Due cammini sono assegnati alla stessa classe se essi differiscono univocamente nel numero di volte per il quale un ciclo sul cammino viene percorso.

Metodo di McCabe

Il metodo di McCabe si concentra sulla determinazione del **numero massimo di casi di test (upper bound) necessari per garantire la copertura dei branch**. Questa tecnica si basa sull'identificazione di un insieme di cammini linearmente indipendenti all'interno del Control Flow Graph.

Funzionamento

Il metodo di McCabe utilizza una rappresentazione vettoriale per i cammini. Ogni cammino può essere rappresentato da un vettore di n elementi, in cui ogni elemento corrisponde a un arco del grafo. Il valore di ciascun elemento indica quante volte quell'arco viene attraversato nel cammino: se un arco non è percorso, il valore è 0; se è percorso k volte, il valore è k. Questa rappresentazione permette di descrivere i cammini in modo dettagliato, includendo sia la presenza sia il numero di attraversamenti di ciascun arco, superando quindi i limiti di un semplice vettore booleano.

Per confrontare i cammini, è necessario rappresentarli rispetto all'intero set di archi del grafo, anche se alcuni cammini non attraversano tutti gli archi. Per questo motivo, i vettori associati includono tutti gli archi del grafo, assegnando valore 0 agli archi non percorsi. La **linearità indipendente** di un cammino si valuta considerando se esso può essere ottenuto come combinazione lineare di altri cammini. Se un cammino attraversa almeno un arco non percorso dagli altri, è indipendente, poiché il valore per quell'arco sarà diverso da 0 solo nel suo vettore, rendendolo impossibile da generare combinando vettori esistenti. Questa condizione, però, è sufficiente ma non necessaria: un cammino può essere indipendente anche se attraversa solo archi già percorsi, a seconda delle combinazioni possibili. Tale rappresentazione consente di identificare in modo sistematico i cammini linearmente indipendenti necessari per garantire la copertura richiesta.

Insieme dei cammini base

McCabe introduce inoltre il concetto di **cammini base**, che costituiscono un insieme massimo di cammini linearmente indipendenti. L'obiettivo del metodo è garantire che l'esecuzione di test case corrispondenti ai cammini base copra tutti i branch. Questo implica che, se un branch non fosse coperto, si potrebbe aggiungere un nuovo cammino al set di base, dimostrando che l'insieme iniziale non era completo.

Teoria ciclomatica dei grafi e calcolo del numero massimo di cammini

McCabe ha applicato la teoria ciclomatica dei grafi al grafo del flusso di controllo (CFG) per calcolare il numero massimo di cammini linearmente indipendenti, noto come numero ciclomatico o **complessità ciclomatica**. Per fare ciò, ha trasformato il CFG in un grafo fortemente connesso aggiungendo un arco fittizio che collega il nodo iniziale con il nodo finale, rendendo così possibile il raggiungimento di ogni nodo a partire da qualsiasi altro. Questo approccio consente di applicare la teoria ciclomatica, che è direttamente collegata alla presenza di cicli nel grafo.

La complessità ciclomatica può essere calcolata in diversi modi equivalenti:

- numero di regioni chiuse (derivate dalla presenza di cicli) + 1;
- numero di nodi decisionali (nodi predicato) + 1;
- numero di archi – numero di nodi + 2

Se il grafo non contiene decisioni (nodi predicato), la complessità ciclomatica senza l'arco fittizio è pari a 0, ma con l'arco fittizio diventa 1. Al contrario, quando sono presenti nodi decisionali, ogni predicato aggiunge una nuova regione chiusa, aumentando il valore della complessità ciclomatica.

Questa misura fornisce una metrica per il numero massimo di cammini linearmente indipendenti all'interno del CFG e serve come base per la definizione del criterio di copertura di McCabe. La presenza di cicli o di nodi decisionali incide sul numero di cammini necessari per coprire completamente il grafo.

Ricerca dei cammini indipendenti

La tecnica per individuare i cammini linearmente indipendenti si basa sulla costruzione incrementale dei cammini, detta anche **tecnica delle baseline successive**. Partendo da un cammino iniziale, si aggiungono progressivamente nuovi cammini che coprono archi non ancora attraversati. Per ogni nuovo cammino, si verifica se è linearmente indipendente rispetto agli altri. In caso affermativo, viene aggiunto all'insieme base. Questo processo termina quando tutti i branch sono coperti e non esistono ulteriori cammini linearmente indipendenti.

Implicazioni del criterio di McCabe

Il criterio di McCabe implica la copertura dei branch, ma non vale il contrario: un insieme di test che copre i branch potrebbe non garantire la linearità indipendente dei cammini. Di conseguenza, il metodo di McCabe è più rigoroso e fornisce una maggiore garanzia di copertura rispetto al semplice branch coverage.