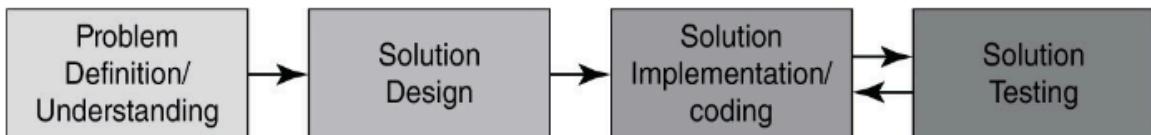


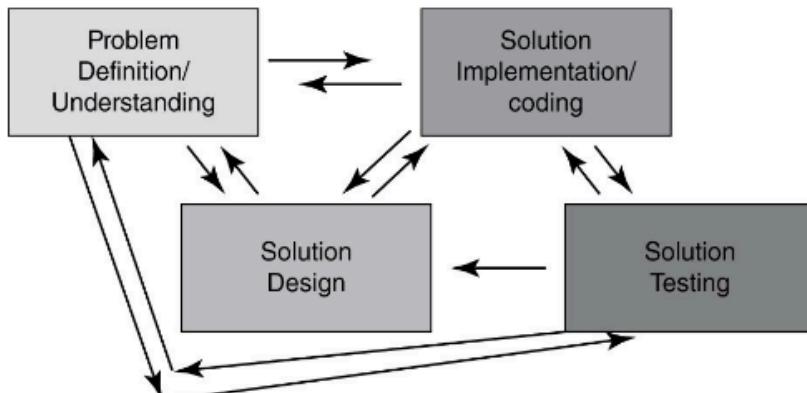
1 Processo software e Modelli di processo

Processo (di produzione) Software: regole per svolgere, coordinare e controllare le attività per raggiungere gli obiettivi del progetto software.

Il processo ideale



Quello che (spesso) succede



Un aspetto importante è la stima dell'**effort**, ossia della quantità di lavoro richiesta a completare un task assegnato.

Task	Effort (hours)	Day Start	Day End
Understand the problem			
Design the program			
Coding (sort algorithm)			
Coding (file read/write)			
Coding (user interface)			
Testing			
Total			

Un processo software definisce:

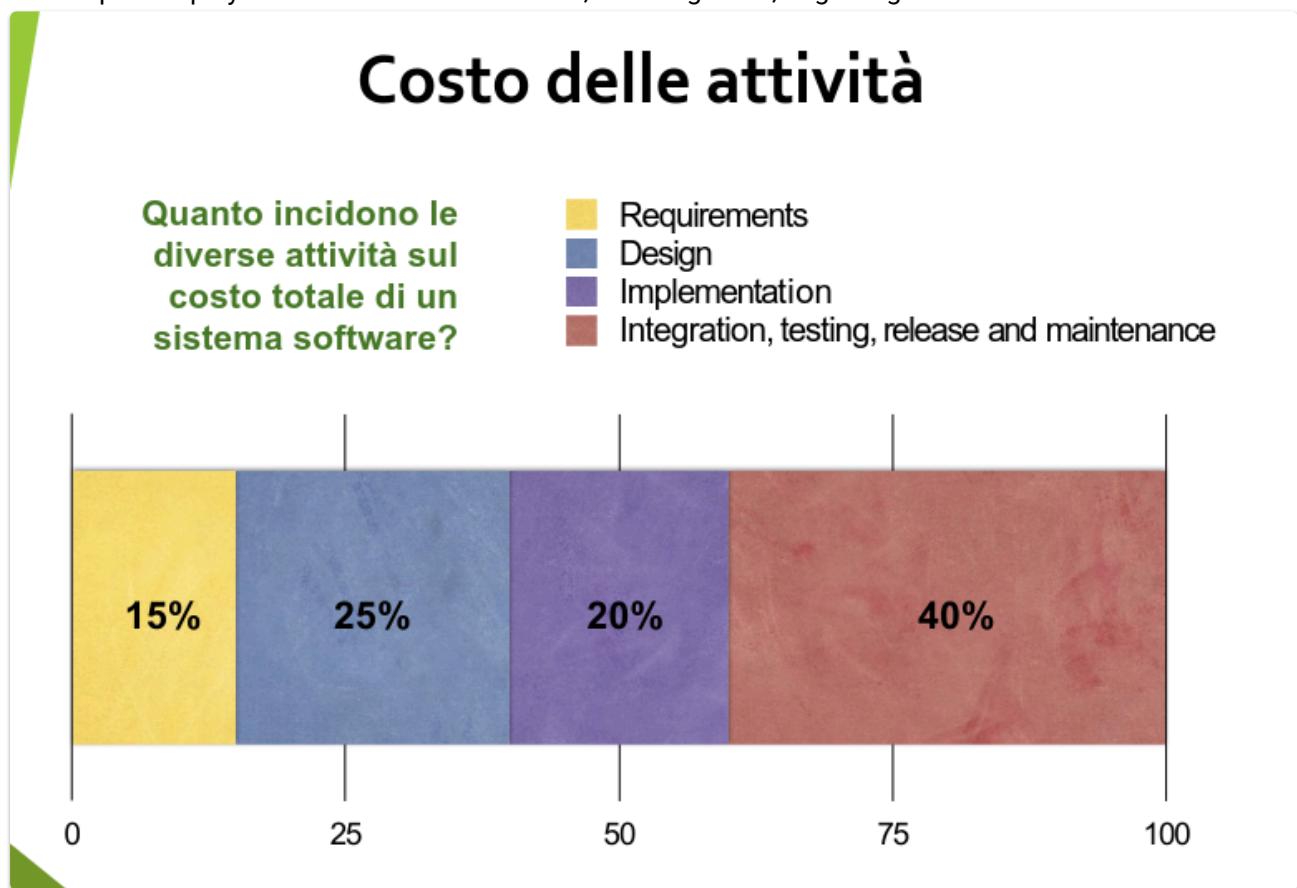
- **Ruoli:** ad un progetto software vi sono diversi partecipanti, ognuno del quale ha un determinato ruolo (project manager, software architect, dev, tester...);

- **Attività (task):** ogni ruolo ha una diversa attività da svolgere. L'attività inizia quando sono soddisfatte le “**entry conditions**” e finisce quando sono soddisfatte le “**exit conditions**”;
- **Linee guida:** indicano come le attività devono essere svolte (pratiche di programmazione, errori comuni, standard da adottare, convenzioni sul codice...);

Con **modello di processo** si intende una **descrizione** astratta del processo software, che si concentra solamente sugli **aspetti principali**.

1.1 Attività principali

1. **Analisi dei requisiti:** determinare le funzioni che deve svolgere il software e i vincoli da considerare;
2. **Progettazione (Design):** definire la struttura del software mediante un modello
3. **Implementazione:** trasformazione del design in un programma funzionante
4. **Integrazione e Testing:** si combinano i vari componenti per formare un sistema completo e si verifica che il tutto funzioni correttamente;
5. **Rilascio e manutenzione:** si effettua il deployment installazione del software nell'ambiente di esercizio) e le attività post-deployment come la manutenzione, data migration, bug fixing...



1.2 Code-and-fix

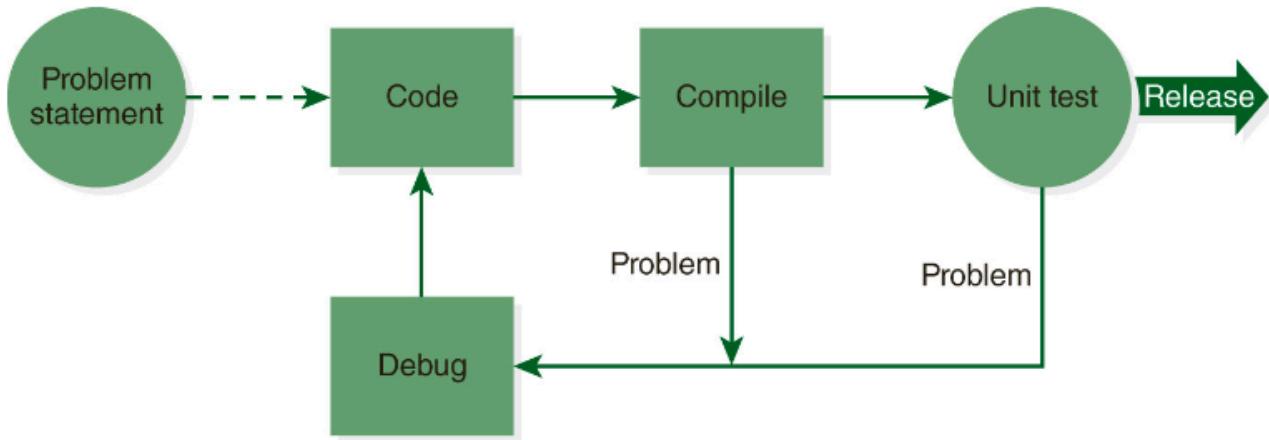


FIGURE 4.1 A simple process.

Modello più semplice, utilizzato per i primi processi software.

1.3 Waterfall

Modello a cascata (Waterfall)

Il primo modello di processo a essere formalmente definito

- Le attività si svolgono **sequenzialmente**, ciascuna riceve come input l'output della precedente
- Consente di descrivere lo **stato di avanzamento** in maniera più precisa di "quasi finito"



Le attività vengono svolte **sequenzialmente** e ciascuna riceve come input l'output di quella precedente. Si utilizza per progetti i cui requisiti rimangono invariati nel tempo.

Vantaggi:

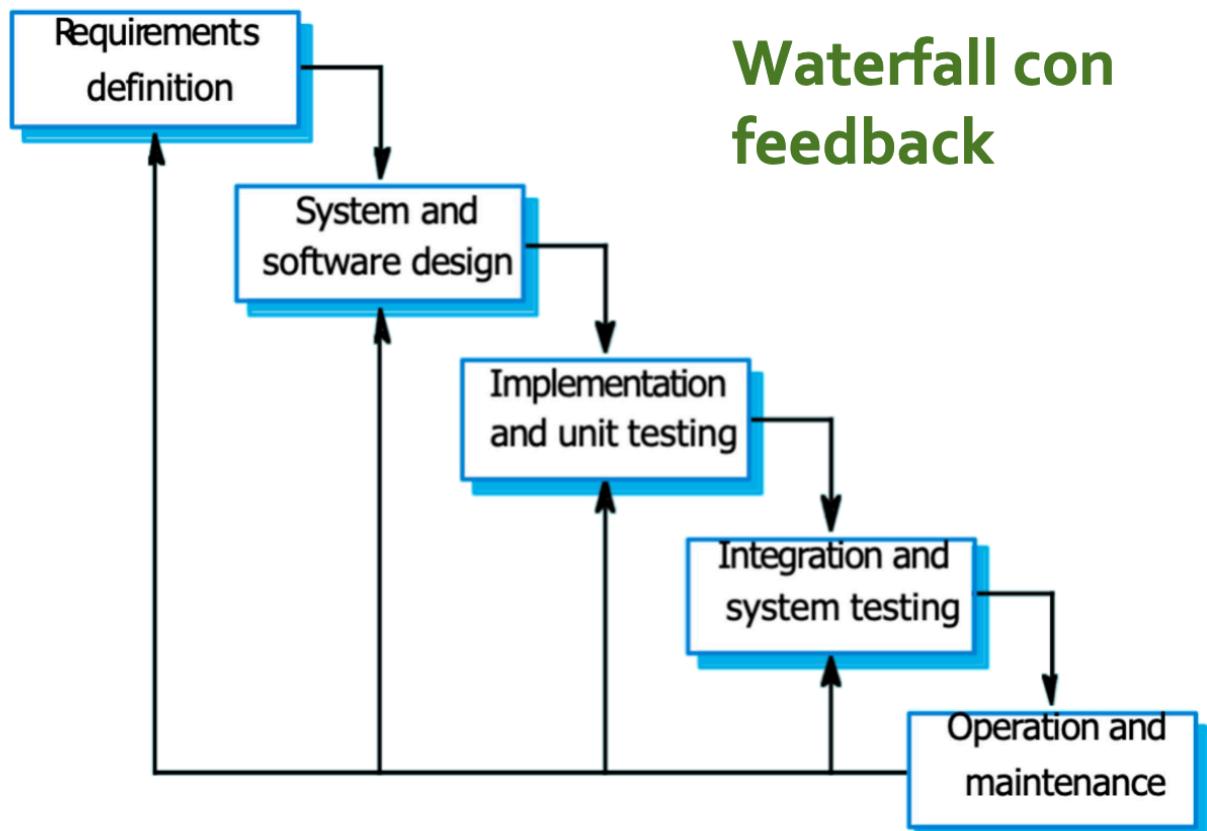
- Struttura chiara
- Divisione del lavoro, monitoraggio e gestione del progetto semplificata

Svantaggi:

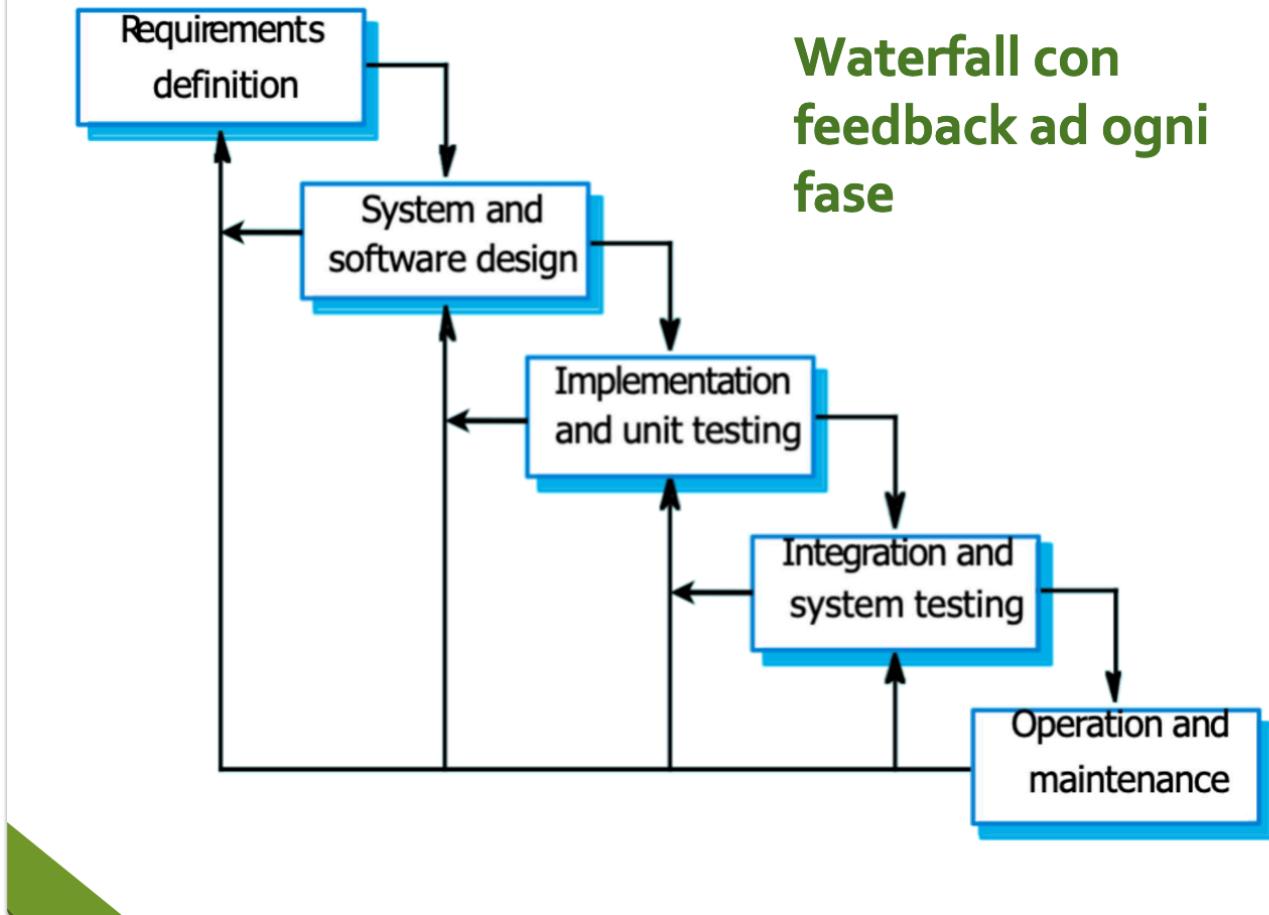
- Feedback degli utenti alla fine del processo;
- Cambiamenti molto **costosi** (soprattutto sui requisiti);
- Se il processo si interrompe durante uno step intermedio si rischia di **non avere risultati intermedi utili**;

Varianti:

Varianti del modello a cascata



Varianti del modello a cascata



1.4 Iterativi

Il sistema è sviluppato in più **iterazioni**, ognuna delle quali comprende le diverse attività (requisiti, progettazione, implementazione, integrazione, testing).

Modello ideale per progetti i cui **requisiti potrebbero cambiare** nel tempo o non sono perfettamente definiti fin dall'inizio.

Vantaggi:

- **feedback** ad ogni iterazione;
- **problemi** individuati e risolti prima;

Ne fanno parte i modelli: **incrementali**, a spirale e gli "Unified Process".

1.4.1 Modello incrementale

Il progetto è diviso in **incrementi** che vengono rilasciati in momenti successivi (release). Ogni **release** implementa una funzionalità aggiuntiva; si è pertanto soliti realizzare il "core" del progetto nella release 1 e successivamente arricchirlo negli incrementi successivi.

Vantaggi:

6. Il cliente ottiene immediatamente un **sottoinsieme funzionante del sistema** (una versione minimale del

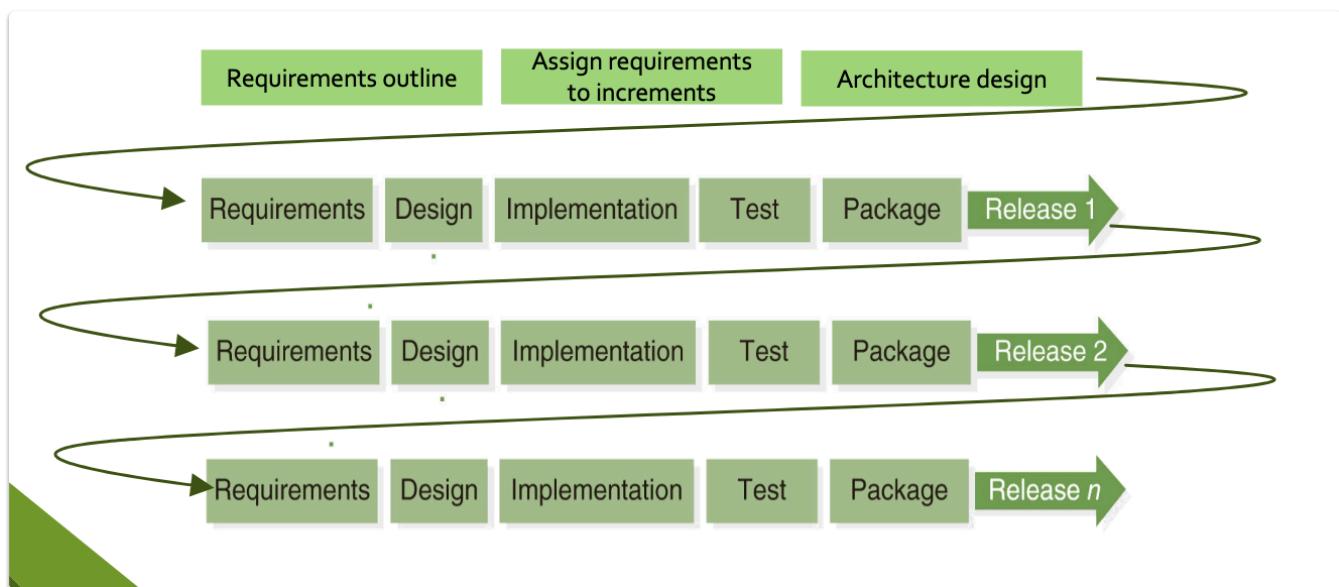
software);

7. Le prime release aiutano a capire i requisiti di quelle successive;
8. Il rischio di **fallimento completo é ridotto**;
9. Le funzionalità più importanti (core) sono realizzate per prime e quindi testate per più tempo.
10. Possibilità di gestire i requisiti che cambiano nel tempo.

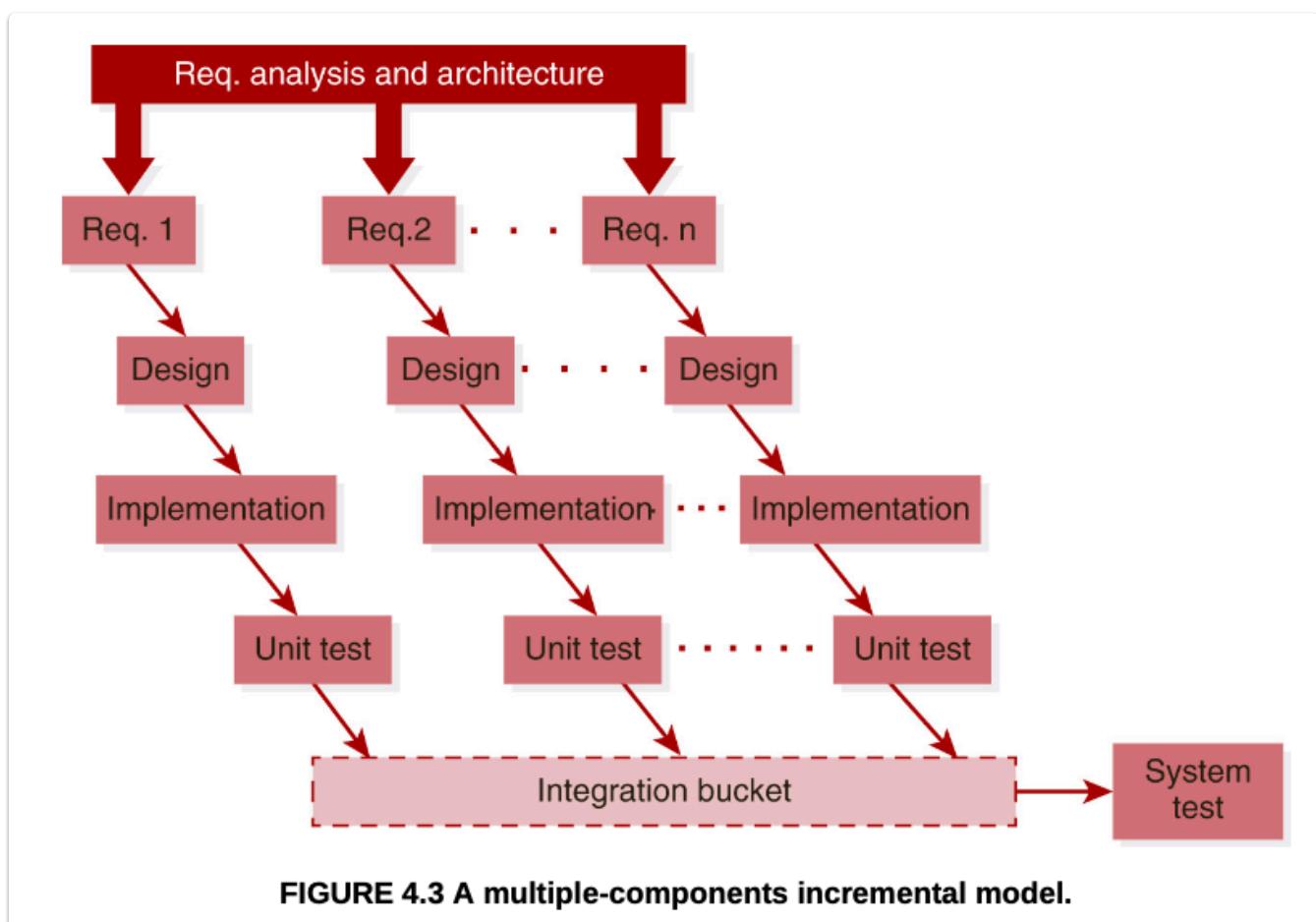
Svantaggi:

11. Lavoro e assegnazione dei task più complesso;
12. Difficile sfruttare il parallelismo in team;

1.4.2 Multiple release incremental model



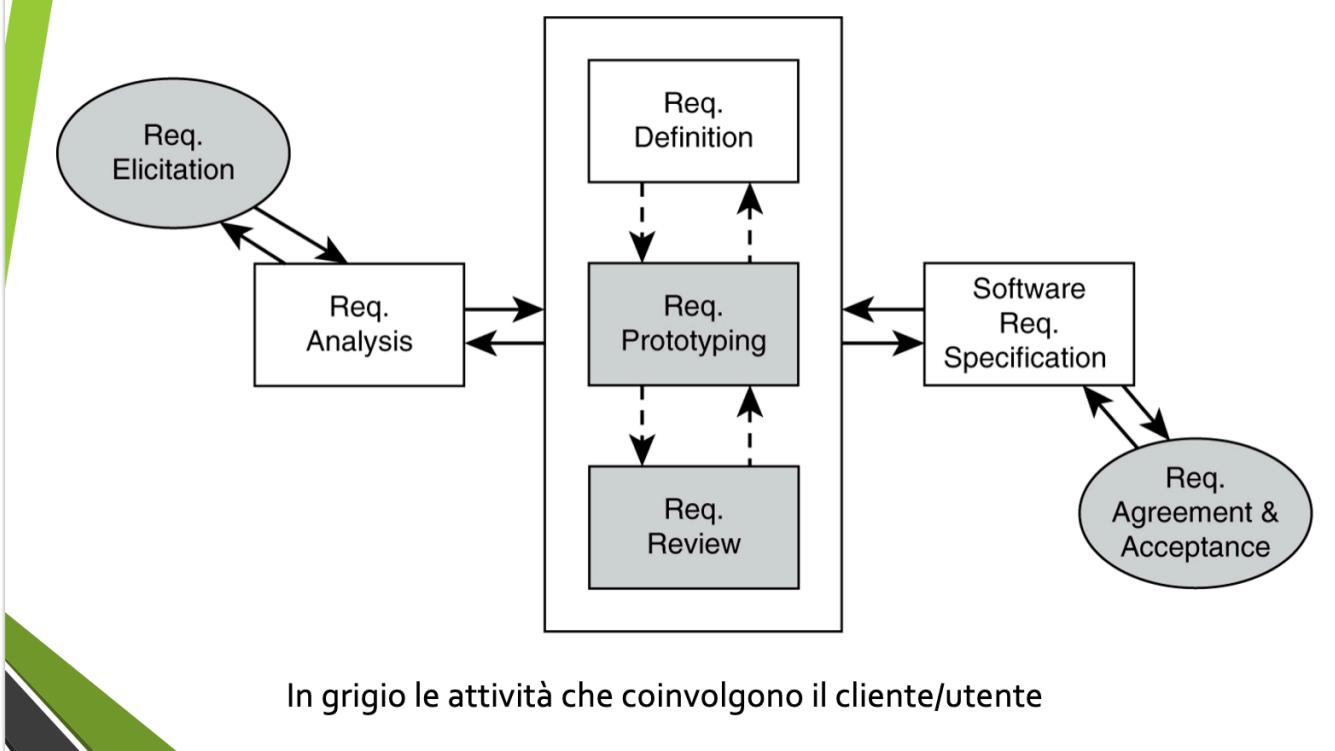
1.4.3 Multiple components incremental model



2 Ingegneria dei requisiti

Ingegneria dei requisiti: insieme delle attività necessarie alla **comprendere** delle funzionalità che richiede il cliente e **definizione** dei vincoli sotto i quali il sistema viene sviluppato.

Attività di Ingegneria dei Requisiti



Requisiti: descrizione delle funzionalità e dei vincoli del sistema. L'ingegneria dei requisiti si compone di 5 fasi diverse:

1. Elicitazione dei requisiti;
2. Analisi dei requisiti
3. Definizione dei requisiti
4. Revisione dei requisiti;
5. Specifica ed accettazione dei requisiti

2.1 Elicitazione

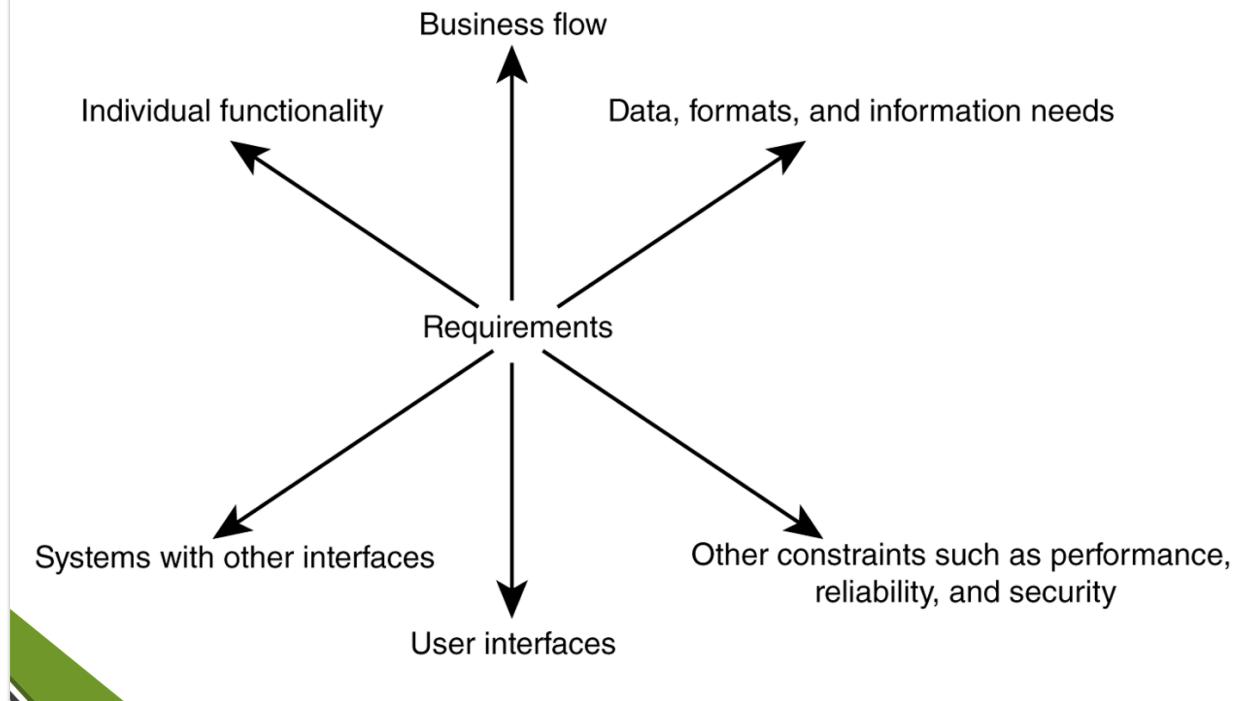
Dato che gli utenti/clienti spesso comprendono solamente i requisiti relativi alle proprie mansioni lavorative, è necessario che sia l'ingegnere del software ad **ottenere informazioni** dai soggetti interessati:

- **Requisiti di alto livello:** obiettivi dell'azienda e della direzione (l'elicitazione viene svolta dagli **analisti di business**);

- **Requisiti di dettaglio:** esigenze specifiche degli utenti;

Requisiti di dettaglio

Si elicotano le **esigenze specifiche degli utenti**



2.1.1 Metodi di elicitatione

2.1.2 Interviste

Utili per ottenere una comprensione generale di ciò che fanno gli stakeholder (soggetti a vario titolo interessati nella realizzazione del sistema software) e di come potrebbero interagire con il sistema;

- **Aperte:** si esplorano varie questioni con gli stakeholder;
- **Chiuse:** si basano su un elenco predeterminato di domande;

2.1.3 Analisi etnografica

Le persone non devono spiegare il loro lavoro e i loro requisiti di dettaglio: è presente un esperto che **osserva ed analizza** il modo in cui le persone lavorano.

2.2 Analisi dei requisiti

Dopo aver elicitato i requisiti, è necessario **organizzare i dati raccolti**.

2.2.1 Categorizzazione dei requisiti

- **Requisiti funzionali:** Funzionalità che il sistema dovrebbe **fornire**, come il sistema dovrebbe comportarsi in particolari occasioni...
- **Requisiti non funzionali:** **Vincoli** sulle funzionalità offerte dal sistema (vincoli temporali, sugli standard da adottare,...).

2.2.2 Categorizzazione di dettaglio

- 1. Funzionalità individuale
 - 2. Business flow (scenari)
 - 3. Esigenze di dati e informazioni
 - 4. Interfacce utente
 - 5. Altre interfacce con sistemi/piattaforme esterne
 - 6. Ulteriori vincoli come prestazioni, sicurezza
affidabilità, ecc.
-
- Funzionale**
- Non funzionale**

6. **Funzionalità individuali:** descrivono i servizi individuali che deve offrire il sistema (es: funzionalità di accesso, gestione conto, ecc...);
7. **Business flow:** descrive le interazioni tra uno o più utenti e il sistema per la realizzazione di uno specifico processo aziendale (es: immatricolazione studente, pagamento fattura, ecc...);
8. **Esigenze di dati ed informazioni:** descrivono i dati che il sistema deve gestire (es: dati profilo utente, dati transazioni, registri di audit, ecc...);
9. **Interfacce utente:** si focalizzano sulla progettazione e sull'usabilità delle interfacce utente del sistema (es: dashboard, design reattivo, messaggi di errore, ecc....);
10. **Interfacce con sistemi esterni:** definiscono il modo in cui il sistema interagisce con altri sistemi esterni (es: integrazione con gateway di pagamento, uso di provider di autenticazione esterni, API, ecc....);
11. **Requisiti non funzionali:** impongono vincoli al funzionamento del sistema (es: prestazioni, sicurezza, affidabilità, scalabilità);

Uno schema di categorizzazione può essere applicato utilizzando un **prefisso e un **numero****

Area dei requisiti	Prefisso	Numerazione delle dichiarazioni dei requisiti
Funzionalità individuale	IF	IF-1.1, IF-1.2, IF-1.3, IF-2.1
Business flow	BF	BF-1, BF-2
Dati e formato dei dati	DF	DF-1.1, DF-1.2, DF-2.1
Interfaccia utente	UI	UI-1.1, UI-1.2, UI-2.1
Interfaccia con i sistemi	IS	IS-1
Ulteriori vincoli	FC	FC-1

2.2.3 Definizione delle priorità

Alcune limitazioni (spesso temporali) rendono impossibile lo sviluppo parallelo di tutti i requisiti, è pertanto necessario dare delle **priorità** per capire quale caratteristiche sviluppare prima.

Priorità in base al **business value** ossia al valore per l'azienda:

- **Alto** (must have);
- **Medio** (should have);
- **Basso** (nice to have).

Priorità in base al **rischio tecnico**:

- **Alto**: difficile prevedere se si riuscirà a realizzare il requisito;
- **Medio**: requisito realizzabile ma sono presenti alcuni fattori di rischio che non possono essere controllati;
- **Basso**: requisito realizzabile;

Può tornare utile definire un **elenco delle priorità**:

Requirement Number	Brief Requirement Description	Requirement Source	Requirement Priority*	Requirement Status
1	One-page query must respond in less than 1 second	A major account marketing representative	Priority 1	Accepted for this release
2	Help text must be field sensitive	Large account users	Priority 2	Postponed for next release

*Priority may be 1, 2, 3, or 4, with 1 being the highest.

Le priorità possono essere definite anche in base ad altri criteri come per esempio le attuali **esigenze dei clienti**, la concorrenza e le condizioni di mercato, le esigenze future.... i criteri sono molto soggettivi e pertanto devono coinvolgere molti **stakeholder** (punti di vista differenti).

2.2.4 Ricerca di coerenza e completezza

L'analisi dei requisiti deve garantire:

- **Completezza**: i requisiti devono includere la descrizione di tutte le funzionalità necessarie di un sistema;
 - **Coerenza**: non devono essere presenti contraddizioni o conflitti nelle descrizioni delle funzionalità del sistema;
- Non sempre, a causa della complessità del sistema, si riesce a garantire completezza e coerenza.

2.3 Definizione dei requisiti

Stesura formale (codifica) dei requisiti. Si può usare:

- Linguaggio naturale;
- Linguaggio naturale strutturato;
- Casi d'uso;
- Diagrammi.

2.3.1 Linguaggio naturale

Elenco di **descrizioni testuali**.

Vantaggi

- Espressivo, intuitivo e universale
- Requisiti compresi da utenti e clienti

Svantaggi

- Mancanza di chiarezza (poca precisione)
- Accorpamento (involontario) dei requisiti

2.3.2 Linguaggio naturale strutturato

Requisiti scritti seguendo una struttura fissa, un [template](#).

Tabelle Input-Process-Output (IPO)

Esempio:

Requirement Number	Input	Process	Output
12: Customer order	<ul style="list-style-type: none">• Items by type and quantity• Submit request	<ul style="list-style-type: none">• Accept the items and respective quantities	<ul style="list-style-type: none">• Display acceptance message• Ask for confirmation message

2.3.3 Casi d'uso

Caso d'uso: insieme di scenari di funzionamento del sistema collegati tra loro, finalizzati al raggiungimento di uno specifico obiettivo dell'utente. Rappresentano il [comportamento esterno](#) di un sistema.

Praticamente sono una [tecnica](#) utilizzata per [identificare i requisiti funzionali](#) di un sistema che si basa sulla descrizione delle interazioni tipiche tra gli utenti ed il sistema. Con [scenario](#) si intende una sequenza di passi che caratterizzano una particolare interazione tra utente e sistema.

Identificano:

- Funzionalità principali;
- Precondizioni e post-condizioni delle funzionalità;
- Flusso di attività dei componenti;
- Condizioni di errori e flussi alternativi;

Descrizione caso d'uso:

- Nome univoco
- Attori partecipanti
- Pre/post condizioni
- Flusso di eventi normale
- Flusso di eventi alternativi

Descrizione del caso d'uso: Esempio

Flusso di eventi:

1. Passeggero seleziona il numero di zone da percorrere
2. Il distributore visualizza l'importo dovuto
3. Passeggero inserisce una somma di denaro, pari almeno all'importo dovuto
4. Il distributore restituisce il resto
5. Il distributore emette un biglietto

Flusso di eventi alternativi:

- 3a. Passeggero annulla l'operazione
 - 3a.1 L'esecuzione riprende dal passo 1.
- 3b. Trascorrono 60 secondi senza che Passeggero inserisca alcuna moneta
 - 3b.1 Il distributore restituisce l'eventuale denaro già inserito
 - 3b.2 L'esecuzione riprende dal passo 2.

2.3.3.1 Attori

Attore: tipologia di utente (non necessariamente umano! Un sistema può utilizzare servizi offerti da altri sistemi) che interagisce con il sistema (es: sistema informativo dell'università - studenti, docenti...). Un attore può partecipare a più casi d'uso contemporaneamente.

2.4 Revisione dei requisiti

Checklist di revisione:

1. **Verificabilità:** il requisito è realisticamente testabile?
2. **Comprensibilità:** il requisito è stato compreso correttamente?
3. **Tracciabilità:** l'origine del requisito è chiaramente indicata?
4. **Adattabilità:** il requisito può essere modificato senza un forte impatto su altri requisiti?

2.5 Specifica ed accettazione dei requisiti

Una volta elicitati, analizzati, definiti e revisionati, i requisiti devono essere inseriti in un documento di **Specifiche dei Requisiti del Software (SRS)**, il quale deve essere accettato dal cliente e costituisce un accordo formale cliente-fornitore.

2.5.1 Tracciabilità dei requisiti

Durante la fase di sviluppo può tornare utile **tener traccia** dei vari requisiti per identificare e suddividere quelli che già sono stati sviluppati e testati da quelli che ancora devono essere implementati. Si può usare una

matrice di tracciabilità:

Requisiti	Design	Codice	Test	Requisiti correlati
1	Componente X	Modulo 3	Caso di test 32	2, 3
2	Componente Y	Modulo 5	Caso di test 16	1
3	Componente X	Modulo 2	Caso di test 27	1
4

3 Specifica di un componente software

Componente software: unità software che può essere sviluppata e testata singolarmente ed usata da altre parti del sistema software.

- **Modularità**: aiuta a dividere un sistema complesso in parti più semplici;
- **Riusabilità**: un componente può essere riutilizzato in più sistemi software diversi;
- Il componente può essere realizzato da un'azienda diversa.

Componenti software

Esempi

- Una **funzione C** che realizza uno specifico algoritmo
- Un **modulo C** (un file .c + il corrispondente file .h) che contiene la definizione di una struttura dati e un insieme di funzioni che manipolano la struttura dati
- Una **libreria C** che contiene un insieme di moduli collegati tra loro
- Una **classe Java** che risolve uno specifico problema
- Un **package Java** che contiene un insieme di classi collegate che, insieme, risolvono uno specifico problema
- Una **libreria Java** che contiene un insieme di package collegati tra loro
- ...

La specifica di un componente, rispetto ai requisiti di un sistema software, è più dettagliata, tecnica e precisa.

La **specificità** del componente determina **cosa quest'ultimo deve fare**. Quali informazioni inserire nella specifica?

- tutte le informazioni necessarie agli sviluppatori che dovranno usare il componente;
- nessuna delle informazioni che servono soltanto agli sviluppatori che dovranno realizzare il componente;

Se un'informazione serve solo a chi realizza il componente, inserirla nella specifica potrebbe impedire ai realizzatori di trovare una soluzione al problema migliore di quella che ha in mente chi ha scritto la specifica...

3.1 Contratti

Contratto: accordo **cliente-fornitore** dove il **cliente** richiede un **servizio** e il **fornitore** fornisce il servizio richiesto. Funge da **protezione** da entrambe le parti:

- **Protezione del cliente:**
 - definisce cosa deve dare il cliente per ricevere il servizio concordato: il fornitore non può richiedergli di più di quanto concordato e scritto nel contratto;
 - definisce il servizio che il cliente deve ricevere: il fornitore non può dargli meno di quello che è previsto;
- **Protezione del fornitore:**
 - definisce il servizio considerato accettabile: il cliente non può richiedere più di quello previsto dal contratto;
 - definisce il compenso che il fornitore deve ricevere: il cliente non può dargli di meno di quello previsto dal contratto;

Se una delle due parti non rispetta i propri obblighi, perde il diritto ai relativi benefici.

Esempio: il contratto tra l'inquilino e il proprietario di un appartamento

Parte	Obblighi	Benefici
Inquilino	Pagare l'affitto all'inizio del mese	Soggiornare in appartamento
Proprietario	Consentire all'inquilino di soggiornare nell'appartamento	Ottenere il pagamento dell'affitto ogni mese

3.1.1 Design By Contract

Il **Design By Contract (DbC)** è un metodo usato per la **specificazione dei componenti software**: le caratteristiche dei componenti sono descritte immaginando di definire un "contratto" che regolamenta la collaborazione tra le due parti.

- **Client** (cliente): chi usa i servizi offerti da un componente software (eg. una funzione, il chiamante);
- **Provider** (fornitore): componente software che offre dei servizi che sono descritti nella specifica;

[Client e Provider rappresentano parti di codice, non persone o aziende!](#)

La progettazione per contratto si basa sul concetto di **asserzione**, ossia un'espressione booleana che non può

essere falsificata: può risultare falsa solamente in presenza di un errore di programmazione. Le tre asserzioni usate sono: pre-condizioni, post-condizioni ed invarianti.

3.1.2 Pre-condizioni, Post-condizioni ed Invarianti

La specifica di un componente prevede una o più operazioni che il client può richiedere al componente (i "servizi" nella metafora del contratto)

- **Pre-condizioni:** condizioni che devono essere verificate nel momento in cui viene effettuata la richiesta di un'operazione al componente software:
 - **Obbligazioni per il client:** il client ha la responsabilità di garantire che esse siano soddisfatte;
 - **Benefici per il provider:** il provider può sfruttarle nella sua implementazione per realizzare la funzionalità richiesta.
Se non sono soddisfatte il **client** è in **difetto**.
- **Post-condizioni:** condizioni che (a patto che le precondizioni siano soddisfatte), devono essere verificate nel momento in cui il componente completa l'operazione richiesta:
 - **Obbligazioni per il provider:** il provider ha la responsabilità di garantire che siano soddisfatte;
 - **Benefici per il client:** il client può sfruttarle per raggiungere i suoi scopi (client richiede l'operazione proprio perché ha bisogno che sia realizzata qualcuna delle sue post-condizioni);
Se non sono soddisfatte, il provider è in difetto.
- **Invarianti:** se il componente software contiene **strutture dati**, al momento della richiesta esse devono trovarsi in uno stato "coerente"... tale stato deve essere mantenuto dopo ogni operazione effettuata. Possono essere considerate allo stesso tempo precondizioni e/o postcondizioni. Un invariante è sostanzialmente un'**asserzione** che riguarda una **classe**. L'inizializzazione di una struttura dati è un'operazione speciale in quanto ha le invarianti solamente come post-condizioni:
 - Obblighi per client e provider;
 - Devono essere **soddisfatte all'inizio e alla fine di un'operazione**... potrebbero non essere verificate durante l'esecuzione di un'operazione;
Se all'inizio di un'operazione un'invariante non è soddisfatta, il client è in difetto. Se all'inizio di un'operazione invarianti e pre-condizioni sono soddisfatte ma, al termine dell'operazione, un'invariante non è soddisfatta, il provider è in difetto.

Da ciò si può dare una definizione di eccezione: si tratta di un **evento** che si verifica quando un'operazione viene invocata nel **rispetto delle sue pre-condizioni** ma **non è in grado di terminare l'esecuzione nel rispetto delle post-condizioni**.

3.1.3 Asserzioni

Espressione booleana che il programmatore si aspetta sia vera in un punto specifico del programma. Se l'asserzione fallisce, viene lanciata un'eccezione. Più nello specifico si tratta di un'espressione booleana che non può essere **falsificata**: può risultare falsa solamente in presenza di un errore di programmazione.

Esempio:

```
#include <assert.h>

void divide(int numerator, int denominator) {
    assert(denominator != 0);
    printf("Result: %d\n", numerator / denominator);
}
```

Aiutano nella fase di debug e facilitano la comprensione del comportamento atteso del codice. Le asserzioni possono essere usate per **controllare** se il "contratto" viene rispettato:

- All'inizio di un'operazione: si **asseriscono le precondizioni e le invarianti** per controllare che il client abbia svolto correttamente la sua parte del contratto;
 - Al termine di un'operazione: si **asseriscono le post-condizioni** e le invarianti per controllare che il provider abbia svolto correttamente la sua parte di contratto
- Non sempre è possibile** validare pre-condizioni, post-condizioni ed invarianti mediante le **asserzioni!** Può capitare che la verifica sia troppo dispendiosa computazionalmente e pertanto, solitamente, si asseriscono solamente proprietà semplici da verificare.

Tuttavia... Non sempre precondizioni, postcondizioni e invarianti possono essere verificate mediante asserzioni!

- L'espressione booleana corrispondente a una precondizione, postcondizione o invariante potrebbe avere un **costo computazionale significativo**

Esempio: l'operazione `is_empty` dello stack ha come post condizione che lo stack non sia modificato. La verifica richiederebbe di creare una copia dello stack all'inizio dell'operazione, e confrontare tutti gli elementi dello stack finale con quelli dello stack iniziale!

Nella pratica si inseriscono solo le asserzioni corrispondenti a proprietà semplici da verificare!

3.1.4 Contratti nella programmazione OOP

Metodi e strutture dati private fanno parte del contratto della classe? Dato che il **contratto regolamenta** i rapporti tra la **classe** e i suoi **client** e, i client non possono accedere ai membri **privati** della classe, questi non fanno parte del contratto. Tuttavia, la metafora dei contratti può essere utilizzata per garantire la correttezza della realizzazione di una classe che, deve rispettare 2 diversi contratti:

- **contratto pubblico**, verso i client della classe. Si definisce al momento della specifica della classe;
- **contratto privato**: garantisce che le varie parti della classe lavorino correttamente tra loro. Si definisce al momento della progettazione di **dettaglio** della classe. Il contratto privato di una classe può essere documentato in una documentazione interna disponibile agli sviluppatori che devono usare la classe.

Le **invarianti** solitamente riguardano solamente la parte **privata** della classe e pertanto il meccanismo dell'**incapsulamento** aiuta a garantire che un client non possa erroneamente violare le invarianti.

3.2 Documentazione della specifica

La specifica di un componente software deve essere documentata e la documentazione deve essere accessibile a:

5. Sviluppatori che **usano il componente** (non è detto che abbiano accesso al codice sorgente);
6. Sviluppatori che **realizzano il componente**: devono garantire che la progettazione e l'implementazione del componente rispetti la specifica;
7. Sviluppatori che devono **manutenere il componente**: è importante che documentazione e codice sorgente siano allineati

Per favorire l'**allineamento** tra codice sorgente e documentazione è buona norma inserire la descrizione della specifica direttamente nel codice sorgente, sotto forma di commenti, e successivamente utilizzare un

software (come [Doxygen](#)), per la generazione automatica della documentazione a partire dai [commenti di documentazione](#).

Esempio

```
/* Calcola la somma di un array di interi
 * Parametri
 *   array puntatore al primo elemento di un
 *   array di interi
 *   n   numero di elementi dell'array (>= 0)
 *   somma puntatore a una variabile intera che
 *         riceverà il risultato; la variabile
 *         non ha bisogno di essere inizializzata
 * Risultato
 *   Al termine della chiamata, la variabile
 *   *somma conterrà la somma degli elementi
 *   dell'array. La funzione non modifica l'array.
 */
void calcola_somma(int *array, int n, int *somma);
```

Cosa garantisce la funzione alla fine della chiamata (postcondizioni)

È buona norma definire anche la "direzione" in cui viaggiano le informazioni descritte da un determinato parametro. Nell'immagine sovrastante per esempio, il puntatore `*somma` viene utilizzata come parametro di [ritorno](#), sfruttando il passaggio di parametri per riferimento... mentre il parametro `array` è un parametro di input. Nei commenti di documentazione è buona norma specificare questa distinzione:

Modo	Direzione informazioni	Il chiamante...	La funzione...
Ingresso (In, Input)	chiamante -> funzione	Deve definire l'informazione della chiamata. Può assumere che l'informazione non sia modificata dalla chiamata.	Può usare il valore dell'informazione. Non deve modificare il valore dell'informazione (a meno che non lavori su una copia).
Uscita (Out, Output)	funzione -> chiamante	Può usare il valore dell'informazione dopo la chiamata.	Deve definire il valore dell'informazione. Non deve usare il valore dell'informazione prima di averlo definito.
Ingresso/Uscita (In Out, Input Output)	chiamante -> funzione funzione -> chiamante	Deve definire l'informazione della chiamata. Può usare il valore dell'informazione dopo la chiamata, ma deve assumere che può essere diverso da quello iniziale.	Può usare il valore dell'informazione. Può modificare il valore dell'informazione.

```
/* Calcola la somma di un array di interi
 * Parametri di ingresso
 *   array puntatore al primo elemento di un
 *   array di interi
```

```

* n numero di elementi dell'array (>= 0)
*
* Parametri di uscita
* somma la somma calcolata
*/
void calcola_somma(int *array, int n, int *somma);

```

4 Progettazione Software

Lo scopo della progettazione è prendere un insieme di **decisioni** su come verrà costruito il sistema (come scomporre il sistema in componenti, come le componenti si **interfacciano** ed **interagiscono** tra loro, come funziona ogni componente).

La progettazione aiuta a garantire il **soddisfacimento dei requisiti**, funzionali e non.

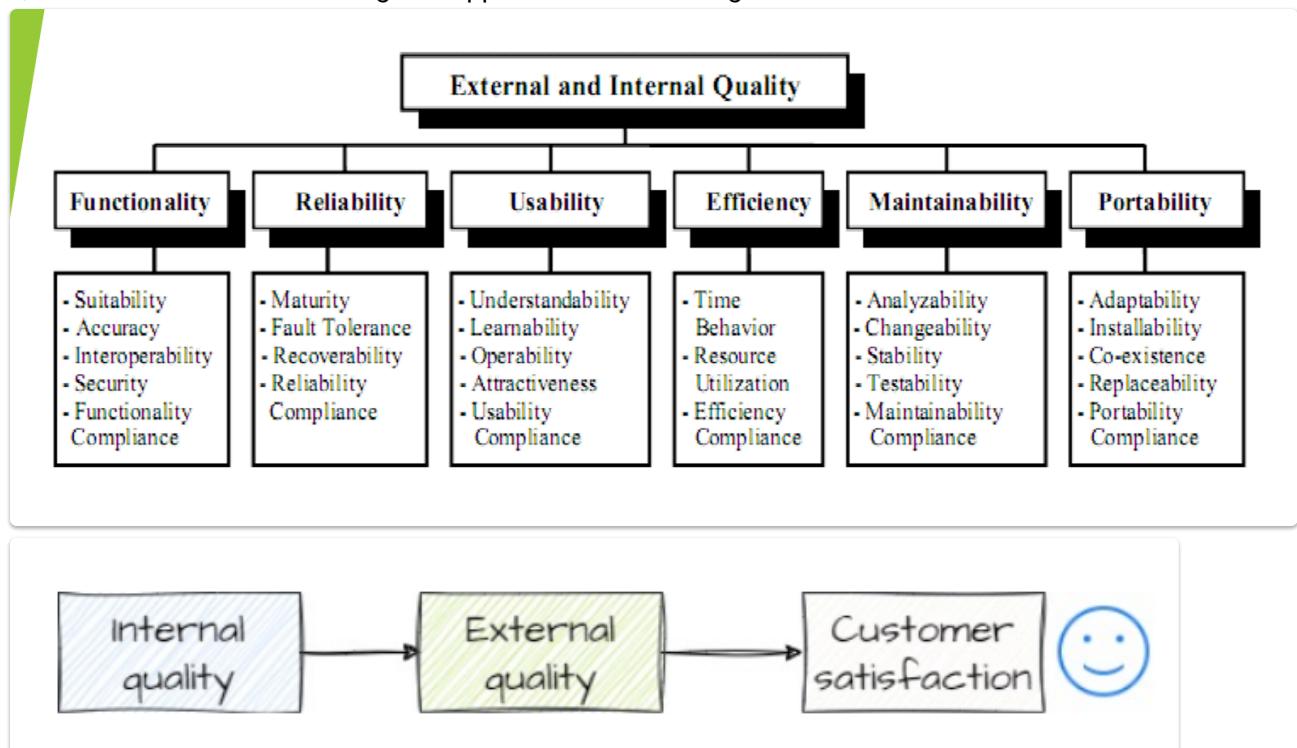
Le decisioni prese in fase di progettazione possono essere rappresentate mediante un insieme di modelli del sistema:

- **statici**: descrivono l'organizzazione del sistema o di una sua parte;
- **dinamici**: descrivono il comportamento del sistema o di una sua parte durante l'esecuzione.

4.1 Attributi di qualità (QA)

QA : proprietà misurabile o testabile di un sistema che ha un impatto sulla capacità del sistema di soddisfare le esigenze degli stakeholder:

- **QA esterni**: visibili agli utilizzatori del sistema (spesso diventano requisiti non funzionali). eg: Performance.
- **QA interni**: visibili solamente agli sviluppatori del sistema. Eg. manutenibilità.



QA interni

- La **manutenibilità** misura la facilità con cui il sistema può ricevere modifiche e aggiornamenti.
- La **modularità** misura il fatto che modifiche a un componente del sistema abbiano un impatto minimo sugli altri componenti.
- La **riusabilità** misura la facilità con cui componenti del sistema possano essere riusati per realizzare un sistema diverso.
- La **portabilità** misura la facilità con cui il sistema può essere trasferito con successo su ambienti/piattaforme diversi.
- La **testabilità** misura quanto sia facile testare in modo efficace e completo il sistema o le sue parti.

QA esterni

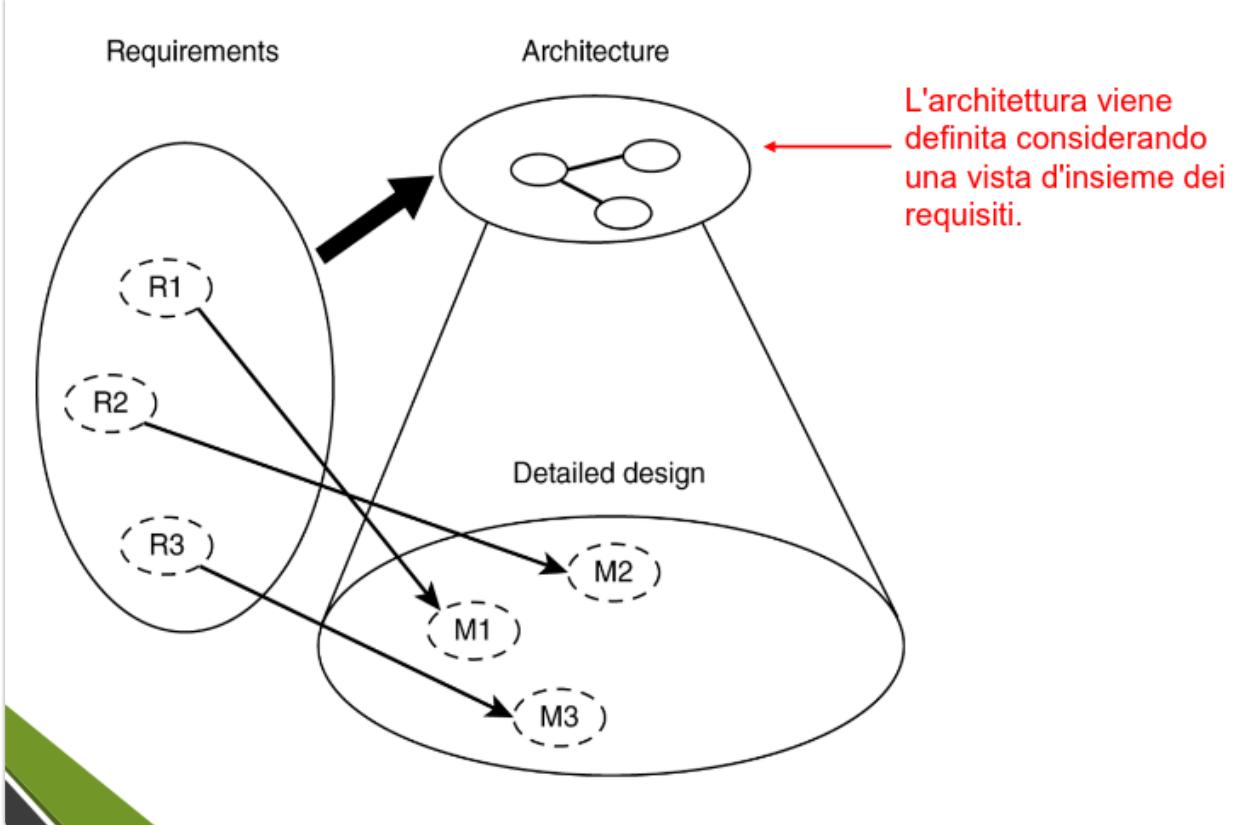
- La **disponibilità** misura il grado in cui un sistema o un servizio è accessibile e operativo
- L' **efficienza** misura la velocità, la reattività e la quantità di risorse utilizzate (es. memoria) di un sistema nell'esecuzione dei compiti.
- La **sicurezza (security)** misura la protezione di un sistema da accessi non autorizzati, violazioni e vulnerabilità.
- La **sicurezza (safety)** misura la garanzia di non causare danni a persone, cose o informazioni.
- La **scalabilità** misura la facilità con cui si può adattare il sistema a gestire un carico di lavoro maggiore (es. un numero maggiore di utenti simultanei) aggiungendo risorse hardware.
- L'**usabilità** misura la facilità con cui gli utenti possono interagire con un sistema per raggiungere i loro obiettivi.

4.2 Fasi della progettazione

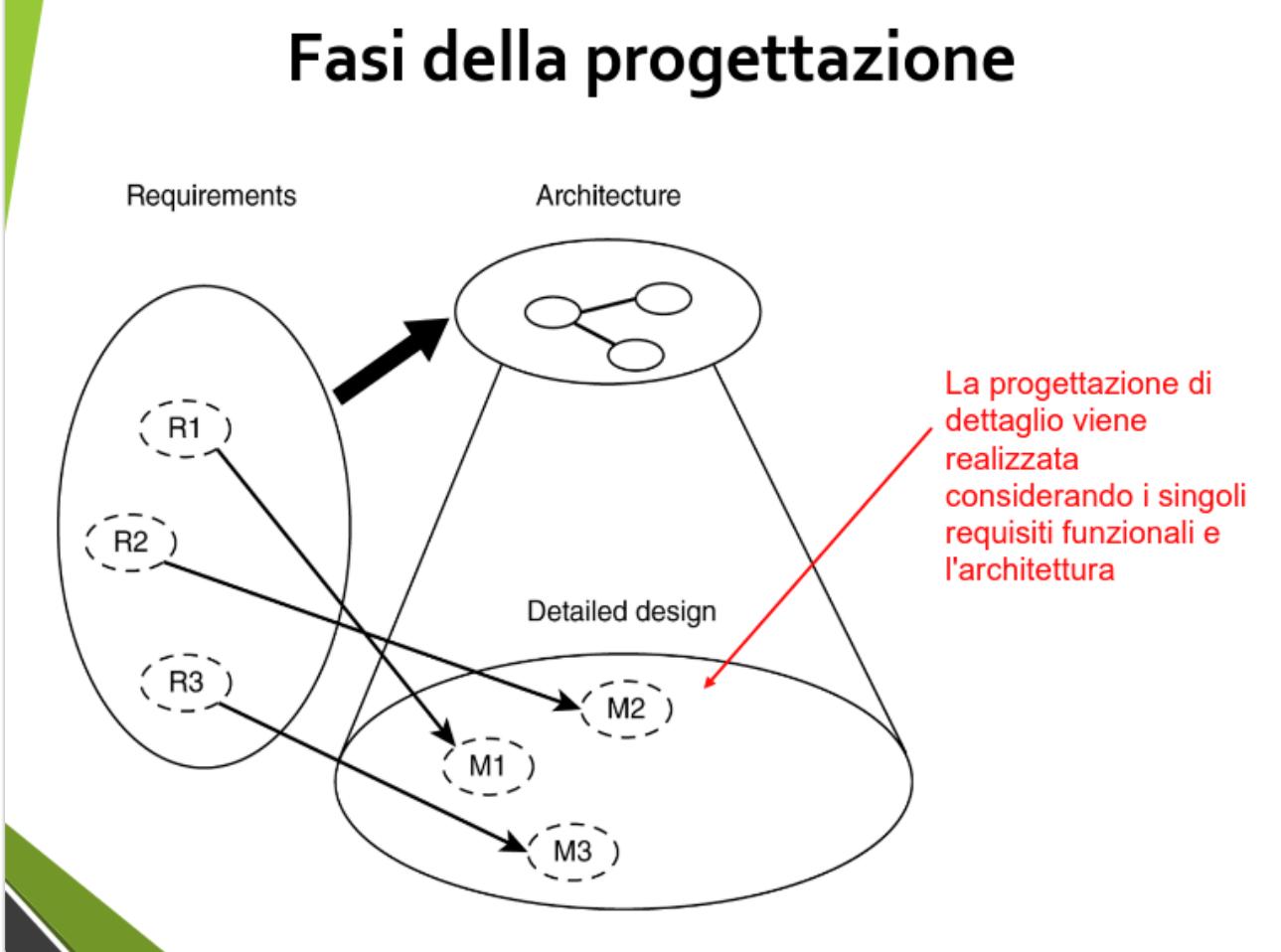
- Progettazione **architetturale**: vista ad alto livello del sistema, sulla base dei requisiti individuati;
- Progettazione **dettagliata**: componenti scomposti a un livello più fine (maggiore **dettaglio**).

4.2.1 Progettazione architetturale

Si definisce a grandi linee una vista ad **alto livello** del sistema, andando ad evidenziare i **componenti principali** e le loro relazioni. Questa fase viene guidata dai requisiti (principalmente quelli non funzionali) e dai QA.



Fasi della progettazione



4.2.1.1 Architettura software

Architettura software: insieme di **modelli** che descrivono il software da sviluppare ossia i suoi principali elementi software (moduli, componenti), le loro proprietà visibili all'esterno e le loro interazioni.

Diverse architetture hanno caratteristiche comuni, pertanto sono stati definiti:

- Stili o **pattern architetturali**
 - Tattiche architetturali
- Utilizzabili come punto di partenza per la definizione dell'architettura del sistema.

4.2.1.2 Pattern architetturali

I **pattern** sono applicabili a contesti generici pertanto è necessario specializzarli a secondo del caso che si sta esaminando.

Esempio: Pipeline

Architettura incentrata sui dati, strutturata sul **modo in cui i dati fluiscono attraverso l'applicazione**.

- L'applicazione riceve i dati in ingresso
- Una serie di **trasformazioni** sui dati viene applicata in modo sequenziale
- L'applicazione restituisce i dati elaborati come output

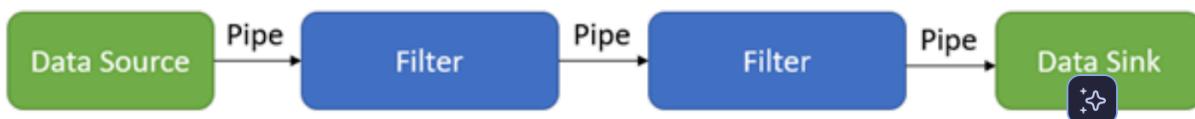
È costituito da una **serie di processi** collegati da «**pipes**»

- L'output di un processo è l'input del processo successivo.
- Ampiamente utilizzato nelle shell dei sistemi operativi
`cat books.txt | sort | uniq`

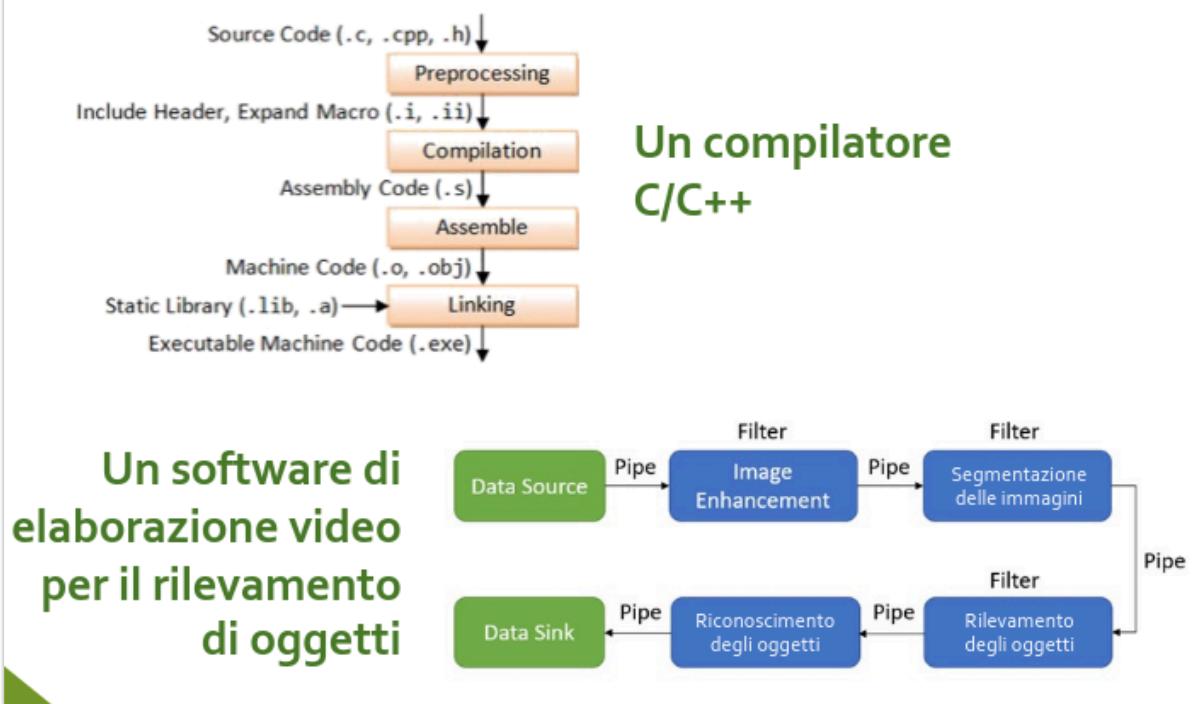
Esempio: Pipeline

Componenti:

- **Data source**: componente responsabile della ricezione dei dati di input e della loro distribuzione lungo la pipeline.
- **Pipe**: meccanismo usato per trasferire e bufferizzare i dati che viaggiano da un componente a un altro
- **Filtro**: componente di elaborazione dei dati che esegue una funzione di trasformazione.
- **Data Sink**: componente che riceve i dati elaborati alla fine della pipeline e li serve come output dell'applicazione.



Esempi di software che usano lo stile architetturale Pipeline



Altri stili architetturali:

8. Event-Driven
9. Client-Server
10. Model-View-Controller (MVC)
11. Stratificato
12. Orientato ai servizi (SOA)

4.2.1.3 Tattiche architetturali

Le **"tattiche architetturali"** risolvono problemi specifici che riguardano più componenti, senza influire sulla struttura complessiva.

Esempio:

- In un sistema vogliamo evitare che **un componente possa bloccarsi** senza che il sistema lo rilevi.
- Introduciamo un nuovo componente responsabile del **rilevamento dei guasti**, e definiamo un meccanismo "condiviso" dagli altri componenti per indicargli di non essere bloccati

Tattica 1 (Heartbeat): Ogni componente invia un messaggio al rilevatore di guasti a intervalli prestabiliti.

Tattica 2 (Ping/Echo): Il rilevatore di guasti invia a intervalli prestabiliti un messaggio a tutti gli altri componenti e attende una risposta.

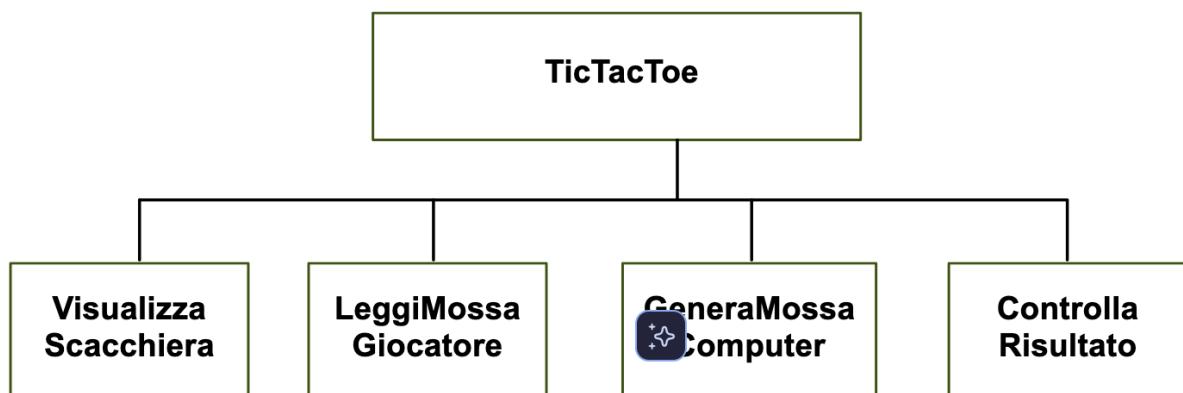
4.2.1.4 Scomposizione in moduli

Nella **progettazione architetturale**, il sistema da realizzare è suddiviso in **componenti**, i quali potranno essere ulteriormente suddivisi nella progettazione di dettaglio.

- **Scomposizione funzionale:** basato sulle **funzionalità** da realizzare, ad ogni livello di scomposizione, operazioni più complesse sono suddivise in operazioni più semplici;

Scomposizione in moduli

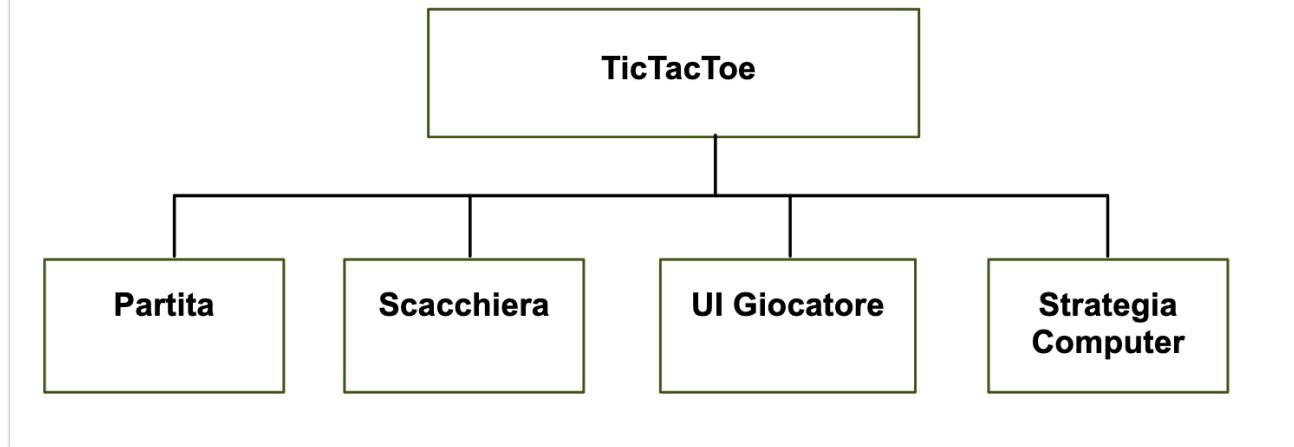
Esempio: supponiamo di voler realizzare un programma che consente di giocare al gioco **Tic-Tac-Toe** ("Tris")



- **Scomposizione object-oriented:** basato sulla suddivisione in **oggetti** (o in classi) che descrivono una parte del dominio del problema (includendo sia dati che operazioni); ad ogni livello di scomposizione, gli oggetti/classi del livello precedente sono ricondotti a oggetti/classi più "piccoli".

Scomposizione in moduli

Esempio: supponiamo di voler realizzare un programma che consente di giocare al gioco **Tic-Tac-Toe** ("Tris")



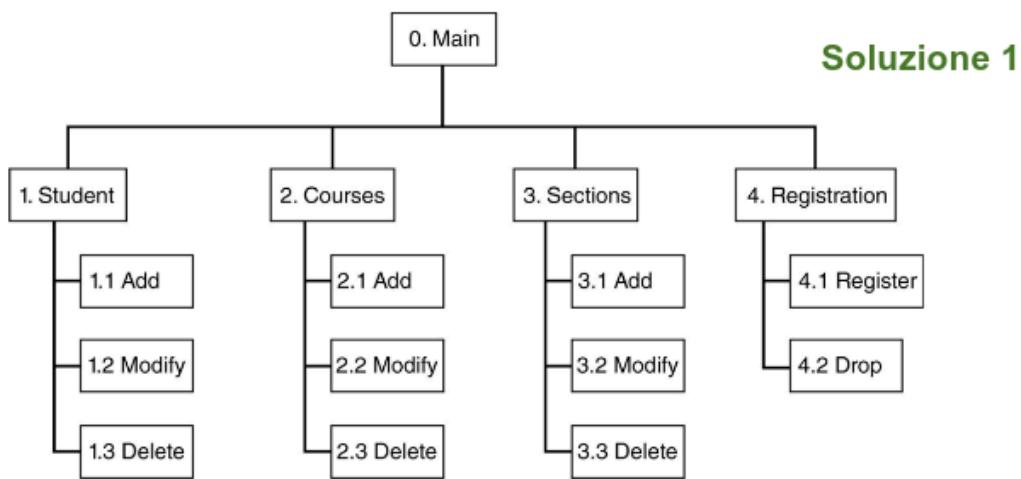
I due approcci possono essere combinati. È essenziale definire correttamente le seguenti informazioni:

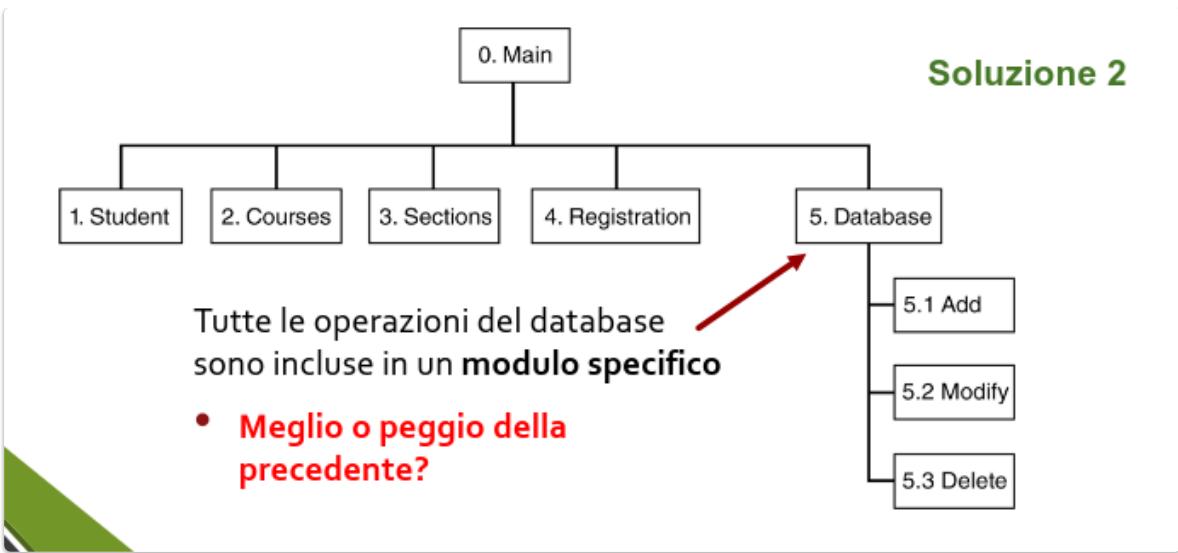
- **Dipendenze** tra moduli;
- **Interfacce** (contratto pubblico di ciascun modulo);
- **Aspetti dinamici**: comportamento ed interazioni tra moduli;

4.2.1.4.1.1 Coesione ed Accoppiamento

Dal momento che esistono svariati modi per scomporre un sistema, è necessario un sistema che consenta di valutarne la **qualità della scomposizione**... si tratta di 2 attributi: la **coesione** e l'**accoppiamento**.

Esempio: per un sistema di registrazione studenti





Un sistema è **scomposto** in maniera **ottimale** se presenta **alta coesione e basso accoppiamento**:

- **Coesione**: misura quanto le parti che sono incluse nello stesso modulo sono **legate** tra di loro (intra-modulo);
- **Accoppiamento**: misura il grado di **interdipendenza** tra moduli diversi (inter-modulo)

Esistono diversi **livelli di coesione**:

Livelli di coesione



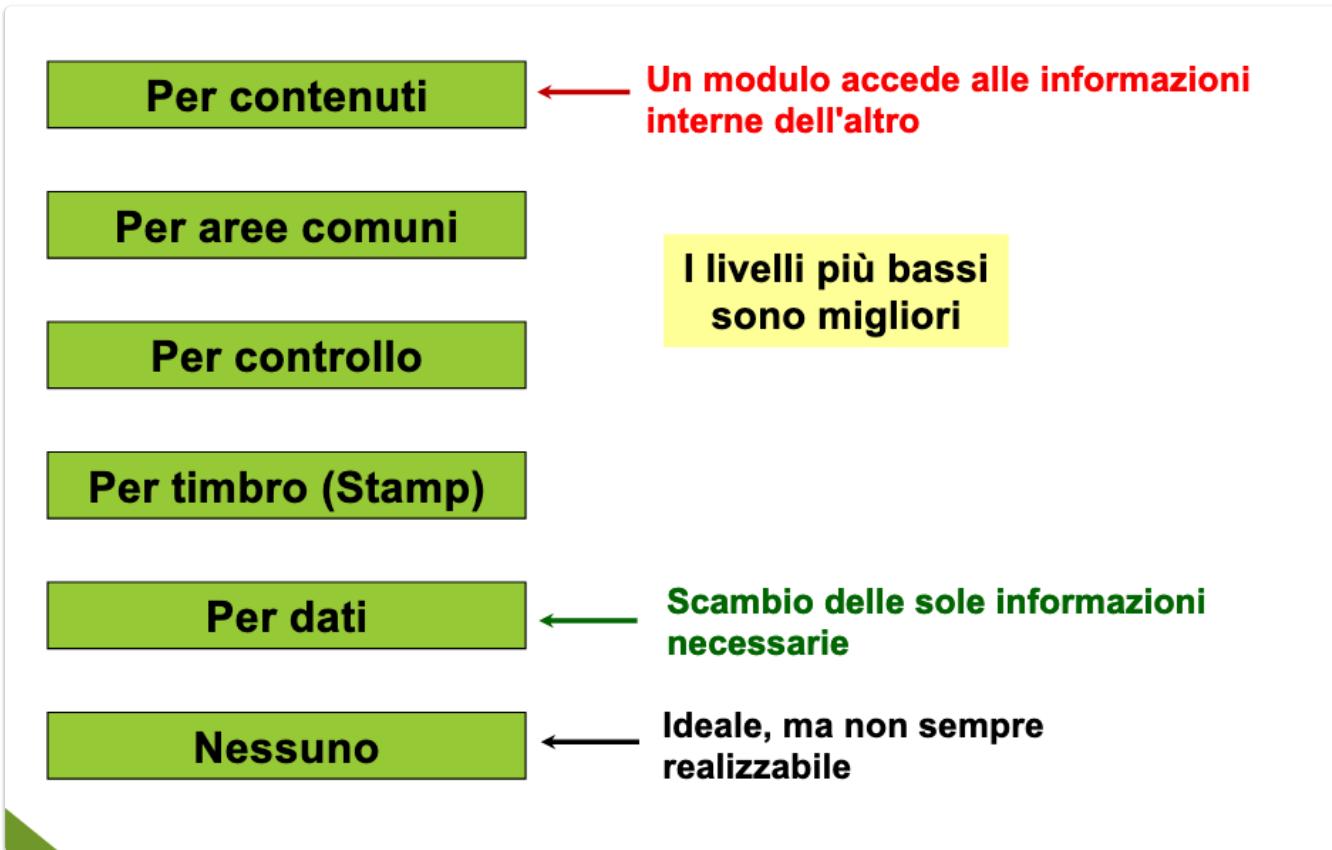
13. **Funzionale**: Il modulo include funzionalità che lavorano insieme per realizzare un singolo compito ben definito;

14. **Sequenziale**: Il modulo include funzionalità che lavorano insieme, e gli output di una sono usati come input da un'altra;

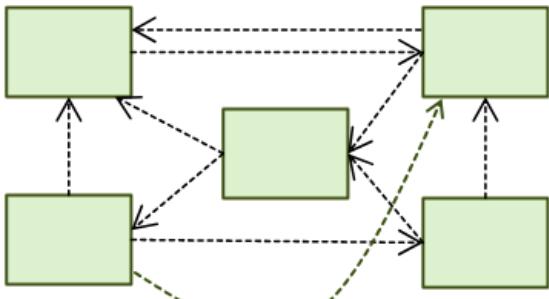
15. **Comunicazionale**: Il modulo include funzionalità che lavorano sugli stessi dati

16. **Procedurale**: Il modulo include funzionalità che vengono spesso usate insieme come parte di una stessa operazione;
17. **Temporale**: Il modulo include funzionalità che vengono usate nella stessa fase di esecuzione del programma;
18. **Logica** : I moduli include funzionalità che svolgono operazioni simili, ma in contesti o su dati non correlati tra loro;
19. **Coincidentale**: il modulo include funzionalità che non hanno nessun collegamento tra loro.

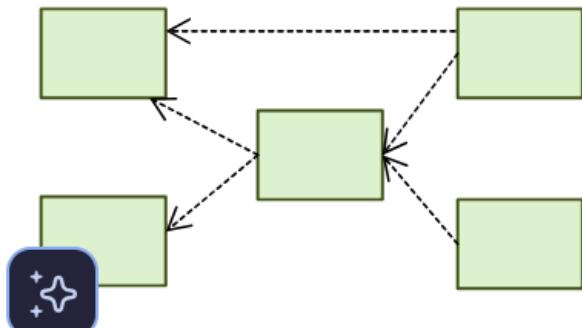
Esistono diversi **livelli di accoppiamento**:



20. Per **contenuti**: un modulo accede agli aspetti implementativi dell'altro (eg. un modulo accede direttamente alla struttura dati di un altro modulo);
21. Per **aree comuni**: due moduli si scambiano informazioni attraverso un'area dati comune (eg. variabili globali);
22. Per **controllo**: Un modulo passa informazioni di "controllo" a un altro, in base alle quali viene cambiata la logica di funzionamento di quest'ultimo (eg. il flusso di esecuzione di un metodo dipende da un flag passato da un'altra classe);
23. Per **timbro (Stamp)**: Un modulo passa una struttura dati a un altro; la struttura passata contiene anche informazioni che non sono necessarie all'altro modulo;
24. Per **dati** : Un modulo passa all'altro solo le informazioni strettamente necessarie per il suo funzionamento;



**Soluzione
peggiore**



**Soluzione
migliore**

right

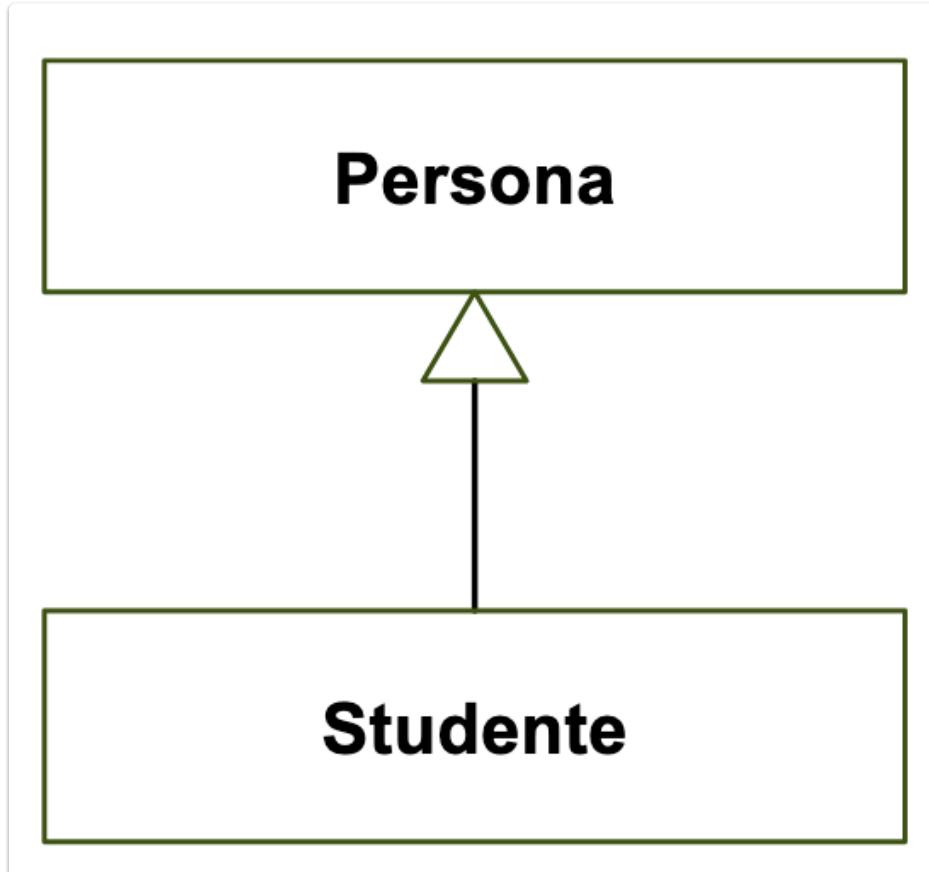
4.3 Progettazione OO

25. Individuare **classi principali**;
26. Si **descrive** il sistema in linguaggio naturale
27. Descrizione dei **requisiti**;
28. Descrizione **comportamento** del sistema;
29. Definire le **relazioni** tra classi;
30. Specificare le **interfacce** .

4.3.1 Ereditarietà e contratti

4.3.1.1 Principio di sostituzione di Liskov

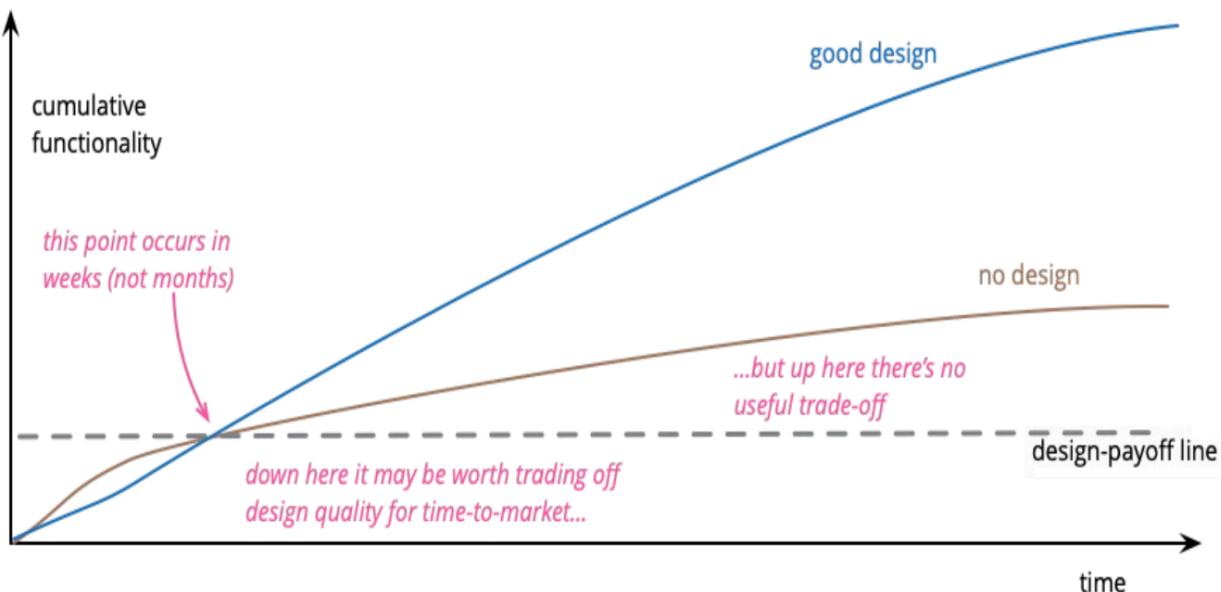
Che relazione c'è tra il contratto di una classe base e quello di una classe derivata?



Nell'analogia con i contratti del mondo reale, questa situazione corrisponde al caso in cui l'azienda che fornisce un servizio subappalta l'erogazione effettiva del servizio a un'altra azienda.

5 Principi di buona progettazione#

Effetto della qualità della progettazione



Fonte: martinfowler.com

Una buona progettazione software rallenta le prime fasi dello sviluppo ma rende più produttive le fasi successive.

Con **debito tecnico** si indica la **cattiva progettazione** (o la mancata progettazione) di un qualche aspetto software. Avere una parte di software progettata male consente di rilasciare il software più velocemente ma l'effort speso alla fine sarà maggiore di quello necessario per una buona progettazione in quanto bisognerà correggere, aggiornare ed estendere le funzionalità.

	Sconsiderato	Prudente
Deliberato	"Non abbiamo tempo per fare la progettazione!"	"Dobbiamo fare il rilascio ora, e poi affronteremo le conseguenze di questa scelta."
Accidentale	"Che cos'è la divisione del software in Layer?"	"Adesso abbiamo capito come avremmo dovuto farlo."

- **Attitudine:** sconsiderata o prudente;
- **Consapevolezza:** deliberato o accidentale;

L'attività destinata alla riduzione dei debiti tecnici prende il nome di **refactoring**.

5.1 KISS - Keep It Simple, Stupid!

[Evitare complessità](#) non necessaria, preferendo:

- classi semplici;
- metodi brevi;
- struttura chiara

[SINE \(Simple Is Not Easy\)](#): la soluzione semplice non è sempre la più semplice da realizzare.

5.2 DRY - Do Not Repeat Yourself

Ogni idea deve essere implementata in un [solo punto](#). Se per esempio operazioni simili sono svolte da parti di codice distinte, è generalmente vantaggioso combinarle in una sola. [Evitare il Copy & Paste](#).

5.3 YAGNI - You Are not Going to Need It

Non introdurre funzionalità non necessarie. L'implementazione di funzionalità extra porta al [code bloat](#): il software diventa inutilmente più esteso e complesso del necessario.

5.4 Separazione delle preoccupazioni

Aspetti diversi del sistema devono essere gestiti da [moduli distinti](#) e non sovrapposti (o minimamente sovrapposti). Quando i problemi sono ben separati, le singole parti possono essere riutilizzate, oltre che sviluppate e aggiornate in modo indipendente.

5.5 Ortogonalità

Le cose che non sono collegate concettualmente non dovrebbero essere "legate" in maniera fissa nel sistema, ma dovrebbero poter essere cambiate indipendentemente l'una dall'altra.

Ortogonalità

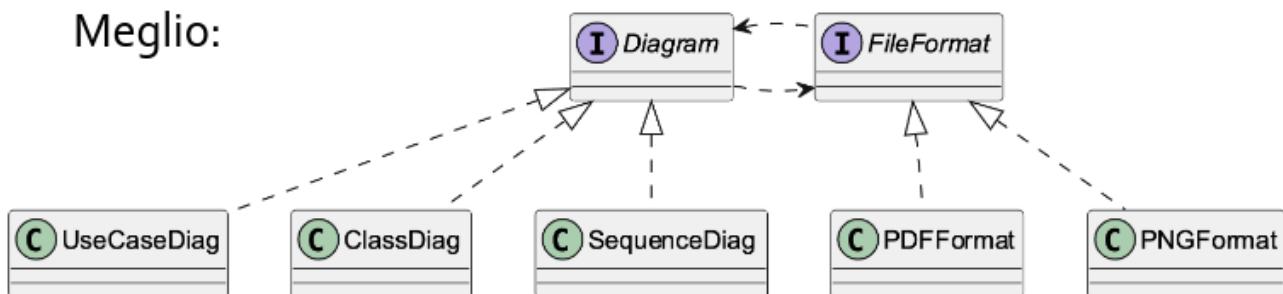
Esempio: dobbiamo realizzare un programma che produce automaticamente 3 tipi di diagrammi UML (use case, class e sequence). Il programma deve produrre i file generati sia come immagini PNG che come documenti in formato PDF.

- Il tipo di diagramma non è collegato concettualmente al tipo di file da generare
- Quindi è sbagliato avere una classe `UseCasePDFGenerator` che lega insieme un tipo di diagramma e un formato di file

Ortogonalità

Esempio: dobbiamo realizzare un programma che produce automaticamente 3 tipi di diagrammi UML (use case, class e sequence). Il programma deve produrre i file generati sia come immagini PNG che come documenti in formato PDF.

Meglio:



In questo modo posso combinare ciascun tipo di diagramma con ciascun formato di file

5.6 Principio della minima sorpresa

Il codice deve essere strutturato in modo da non sorprendere o confondere chi dovrà leggerlo o manutenerlo.

5.7 Evitare l'ottimizzazione precoce

Pensare all'ottimizzazione del codice solo quando il codice funziona correttamente ma è più lento di quanto vorremmo.

"La strategia è sicuramente: prima farlo funzionare, poi farlo funzionare bene e, infine, renderlo veloce"
(Stephen Johnson e Brian Kernighan).

"L'ottimizzazione prematura è la radice di tutti i mali"
(Donald Knuth)

5.8 Regola del boy-scout

"Lasciate il campo più pulito di come lo avete trovato"

- Quando si apportano modifiche a un codice esistente, la qualità tende a diminuire.
- In questi casi occorre prestare molta attenzione a non compromettere la qualità
- Ogni volta che ci si imbatte in codice non sufficientemente chiaro, si dovrebbe cogliere l'opportunità di correggerlo subito.

5.9 Principi orientati agli oggetti: SOLID

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

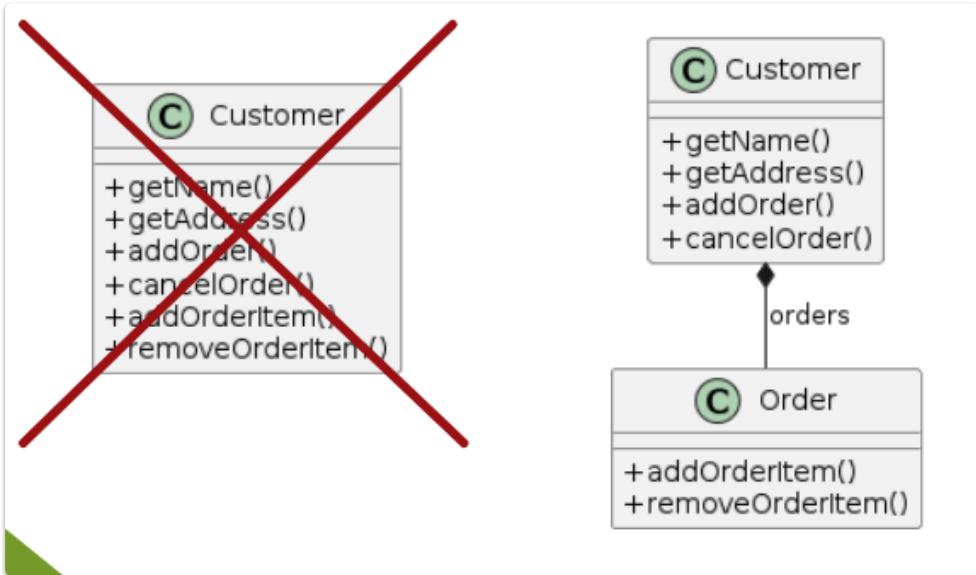
Interface Segregation Principle

Development Dependency Inversion Principle

Si tratta di 5 principi fondamentali che favoriscono l'**alta coesione e basso accoppiamento**.

5.9.1 Singola responsabilità

Un componente del codice deve svolgere un [singolo compito](#) ben definito.

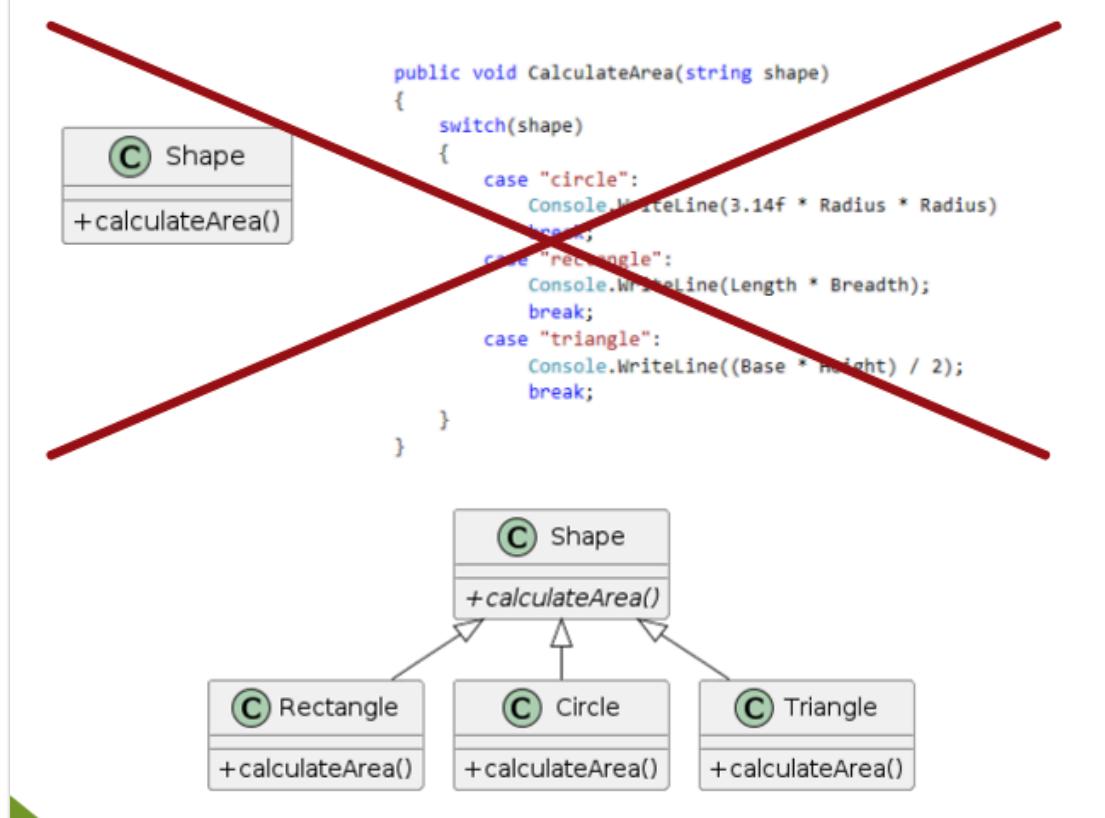


5.9.2 Aperto/Chiuso

Le unità software (classi, moduli, funzioni, ecc.) dovrebbero essere [aperte all'estensione](#), ma [chiuse](#) alla modifica:

- Chiusura rispetto alla modifica: evitare modifiche che possano avere un impatto sui client dell'unità;
 - Apertura all'estensione: consentire modifiche che consentano di adattare l'unità a nuovi scenari, diversi da quelli considerati inizialmente
- I meccanismi dell'ereditarietà e il polimorfismo aiutano a raggiungere questo obiettivo.

Principio aperto/chiuso



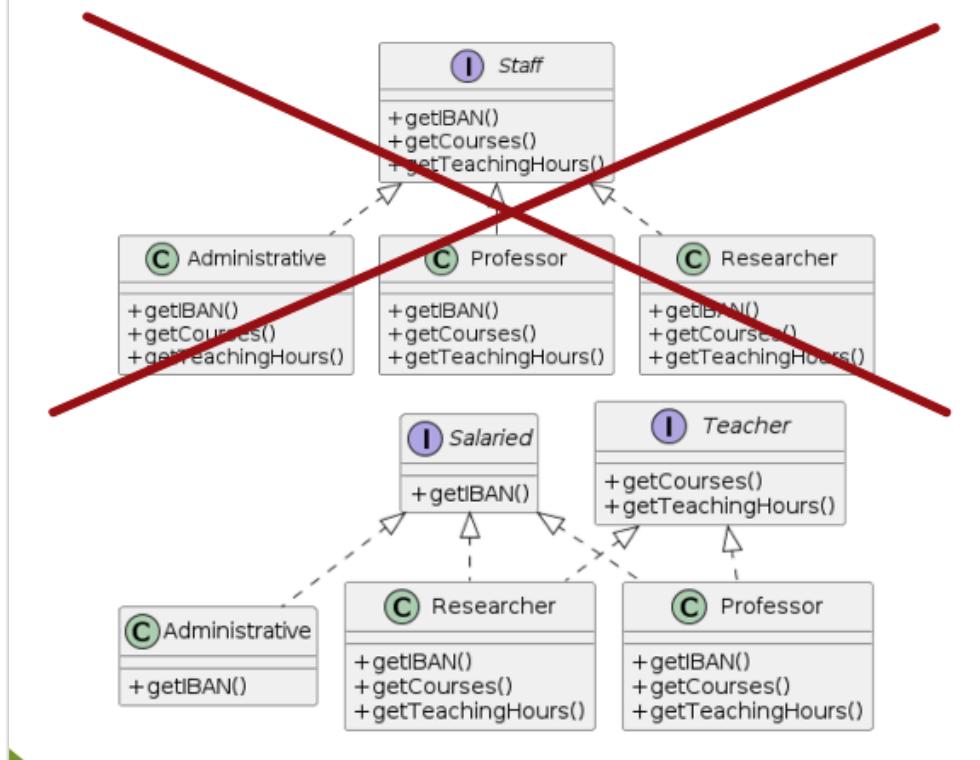
5.9.3 Principio di sostituzione di Liskov

Gli oggetti devono poter essere sostituiti con istanze dei loro sottotipi senza alterare la correttezza del programma.

5.9.4 Principio di segregazione delle interfacce

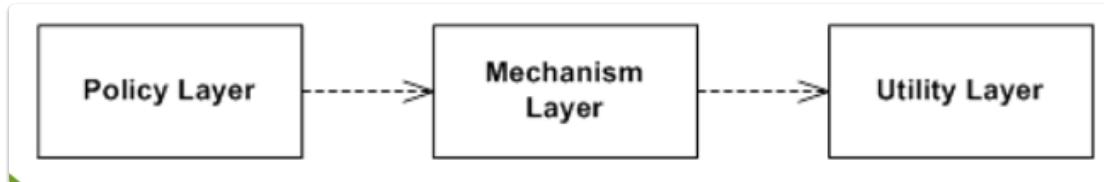
Un client non dovrebbe dipendere da metodi che non utilizza. Per cui: le interfacce devono essere **molte, specifiche e piccole** (pochi metodi).

Principio di segregazione dell'interfaccia

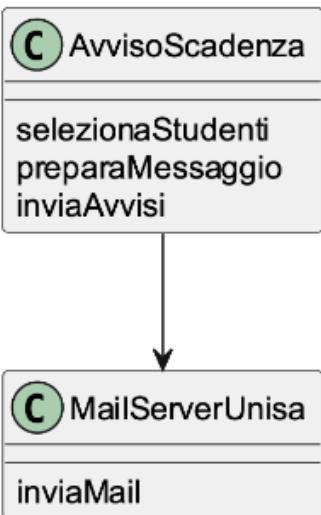
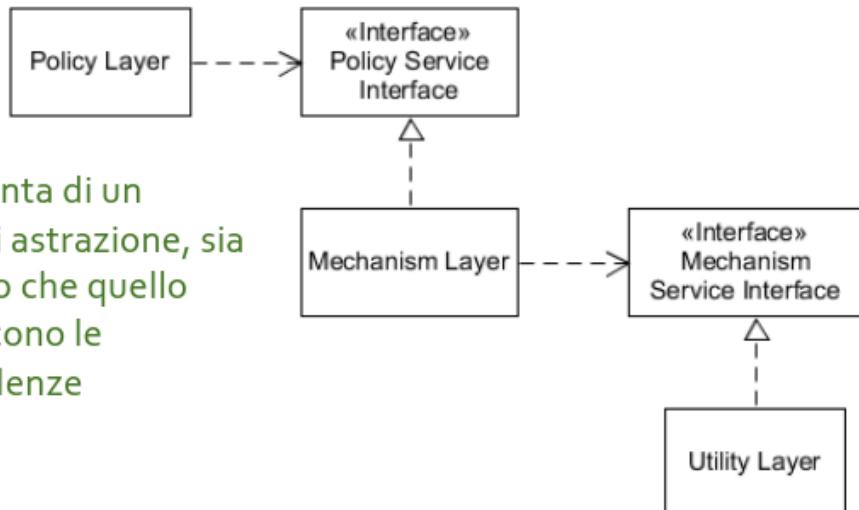


5.9.5 Principio di inversione della dipendenza

Nelle architetture convenzionali, i componenti di **livello inferiore** sono progettati per essere **usati** da componenti di **livello superiore**, consentendo di costruire sistemi via via più complessi. I componenti di livello superiore sono pertanto strettamente dipendenti da quelli sottostanti e ciò ne limita le opportunità di riuso.



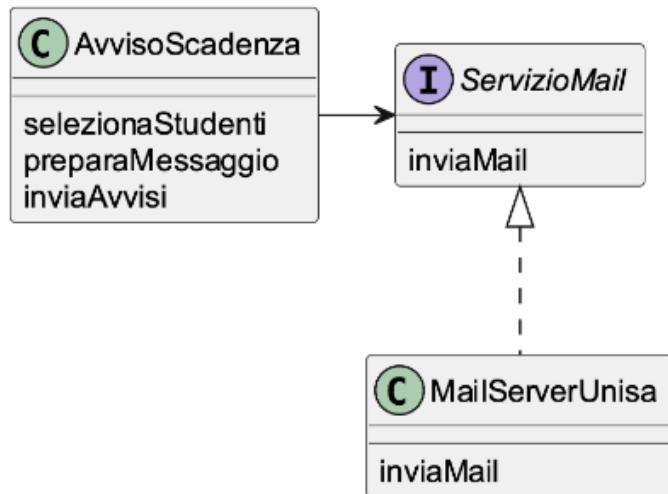
- Con l'aggiunta di un livello di astrazione, sia il livello alto che quello basso riducono le loro dipendenze



Componente di alto livello che stabilisce le policy (decisioni su chi deve ricevere l'avviso, il contenuto del messaggio, quando fare l'invio ecc.)



Componente di basso livello che implementa un meccanismo (come connettersi al server mail di UNISA, come fare l'autenticazione ecc.)



5.9.6 Privilegiare l'associazione rispetto all'ereditarietà

L'ereditarietà è una forma di **accoppiamento** più forte della associazione e composizione. Utilizzando l'ereditarietà, le sottoclassi possono facilmente fare assunzioni sbagliate e infrangere il principio di sostituzione di **Liskov**. L'associazione comporta un minore accoppiamento tra le classi.

5.10 Principio di Robustezza

Il contratto di un componente software prevede che se il client soddisfa le precondizioni, il provider soddisferà le post-condizioni. Questo è sufficiente per garantire la **correttezza**. Tuttavia, nel caso in cui le precondizioni non sono rispettate, è preferibile progettare il componente in modo da "limitare i danni" (**robustezza**).

6 Testing

6.1 Introduzione al testing

Insieme di procedure e tecniche volte a misurare la robustezza e la bontà del software sviluppato.

- **Assicurazione delle qualità (QA)**: attività che consentono di **migliorare le qualità** di un prodotto;
- **Controllo qualità (QC)**: attività volte a **convalidare e verificare la qualità** del prodotto attraverso l'individuazione di difetti. Risponde alle domande:
 - Il software è **conforme ai requisiti** specificati? Stiamo costruendo correttamente il sistema?
 - Il software **soddisfa le esigenze** degli utenti, è idoneo all'uso? Stiamo costruendo il sistema nel modo corretto?

Distinguiamo diverse tipologie di guasti/errore:

- **Fallimento** (guasto): deviazione del comportamento osservato del sistema rispetto al comportamento specificato;
- **Difetto** (bug): condizione che in alcune circostanze può causare fallimento. È il risultato di un errore commesso dallo sviluppatore.

Il software viene esaminato da un team adatto in un processo detto "**software review**":

- **walk-through**: lo sviluppatore presenta l'API e la relativa documentazione al team di revisione che formula commenti sulla copertura dei requisiti e la coerenza con l'architettura;
- **ispezione**: simile al walk-through ma lo sviluppatore non è autorizzato a presentare gli artefatti ossia il materiale da analizzare per la verifica della qualità (eg. codice sorgente, documentazione, test cases...)

6.2 Definizioni di Test del software

- **Definizione debole**: processo di **verifica sperimentale** (comporta l'esecuzione del programma a differenza della software review) volto a dimostrare l'**assenza di difetti** in un sistema software;
 - I test riescono a mostrare solamente la presenza di bug non la loro assenza!
- **Definizione forte**: il test è il **tentativo** sistematico di **trovare** (sperimentalmente) in modo pianificato i **difetti** nel **software** implementato;
 - Implica che gli sviluppatori siano disposti a smantellare le cose al fine di correggere gli errori ed aumentare l'affidabilità del sistema.

6.2.1 Concetti di test

31. **Componente testato**: parte del sistema che può essere isolata per il test (eg. un metodo, un gruppo di oggetti ...)

32. **Caso di test**: insieme di input e risultati attesi che esercita un componente atteso. Lo scopo è quello di **provocare guasti** per rilevare difetti. Può essere descritto mediante una serie di attributi.

Attributes	Description
name	Name of test case
location	Full path name of executable
input	Input data or commands
oracle	Expected test results against which the output of the test is compared
log	Output produced by the test

33. **Oracolo di test**: Un'entità **al di fuori del componente testato** in grado di fornire i risultati attesi per un caso di test (oracolo **esplicito**), o almeno di verificare se i risultati ottenuti sono accettabili (oracolo **implicito**). Sostanzialmente è l'entità (anche un uomo volendo), che verifica se i risultati ottenuti sono sufficienti o meno.

1. Le post-condizioni stabilite nella fase di Design possono essere usate per definire un oracolo;

34. **Debugging**: Se il test rileva la presenza di un difetto, è necessaria un'ulteriore attività (debugging) per individuare l'errore

35. **Correzione** (fix): modifica al componente finalizzato a riparare un difetto

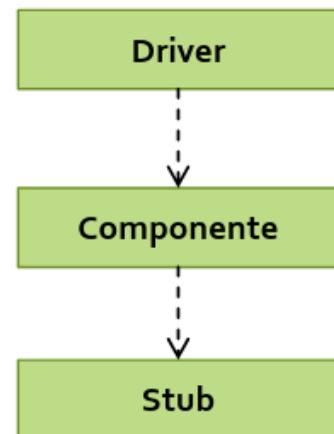
36. **Test stub**: implementazione parziale dei componenti da cui dipende il componente testato;

37. **Test driver**: implementazione parziale di un componente che dipende dal componente testato

Gli **stub** e i **driver** di test consentono di isolare i componenti dal resto del sistema per i test.

Test stub

- Un'implementazione *parziale* dei componenti **da cui dipende il componente testato**

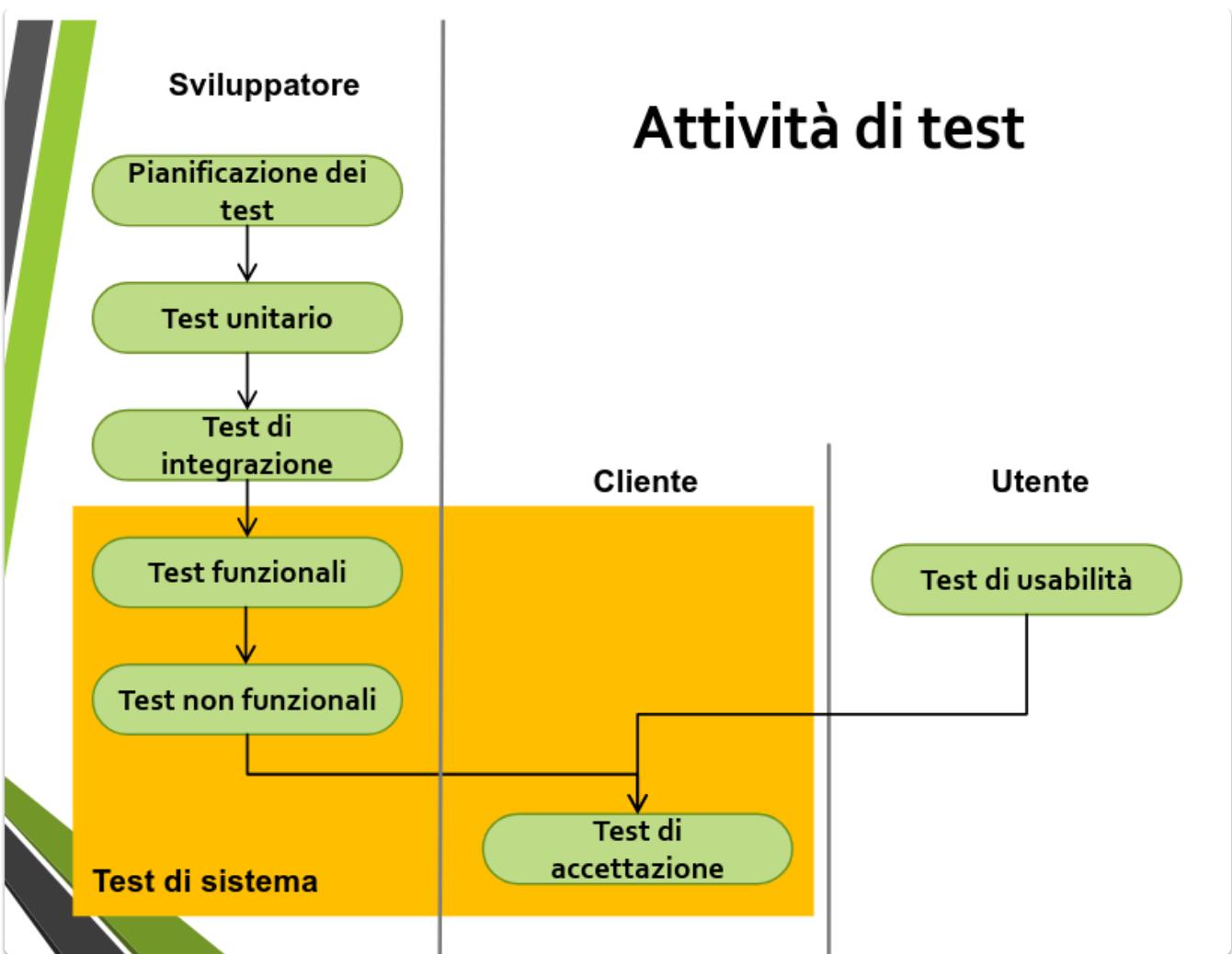


Test driver

- Un'implementazione parziale di un componente **che dipende dal componente testato** (es. una funzione che richiama la funzione testata passando gli input del caso di test)

- **Test di regressione**: ogni qualvolta si modifica un componente difettoso è necessario effettuare nuovamente i test. Con test di regressione si intende la ri-esecuzione di tutti i test precedenti dopo una modifica.

6.3 Attività di test



Il testing del progetto software si articola in svariate fasi:

38. **Pianificazione** dei test: si allocano le risorse e si programmano i test;
39. Test di **usabilità**: test volti alla ricerca di problemi nelle interfacce grafiche
40. Test **unitari**: test effettuati sui singoli componenti software, svolti dagli stessi sviluppatori;
41. Test di **integrazione**: cercano di rilevare difetti testando componenti in combinazione con altri componenti;
42. Test di **sistema**: si testa il sistema nella sua interezza. Includono:
 1. Test **funzionali**: verificano i requisiti funzionali;
 2. Test **non funzionali**: verificano i requisiti non funzionali;
 3. Test di **accettazione**: validano il sistema rispetto all'accordo stipulato con il cliente

6.3.1 Test unitari

Si tratta di test effettuati sui **singoli componenti** software: oggetti e sottoinsiemi. I candidati sono scelti dal diagramma delle classi. I sottoinsiemi si testano dopo aver testato singolarmente ogni componente che lo compone.

6.4 Tecniche per l'individuazione dei casi di test

6.4.1 Black-box testing

Black-Box testing (approccio funzionale o basato sulla specifica): i testCase sono definiti a partire dalla **specific**a del componente testato. Come può essere effettuato:

- **Partizionamento in classi di equivalenza e test di equivalenza**: gli input possibili vengono suddivisi in **classi di equivalenza** e per i test si usano solamente alcuni membri di ogni classe. L'ipotesi è che i sistemi **si comportino in modo simile** per tutti i membri di una classe.

Test di equivalenza

Esempio: testare un metodo che restituisca il numero di giorni in un mese, dati il mese e l'anno

```
class MyGregorianCalendar {  
    ...  
    public static int getNumDaysInMonth(int month, int year) {...}  
    ...  
}
```

Classi di equivalenza per il parametro **mese**:

- Mesi con **31 giorni** (es. **1, 3, 5, 7, 8, 10, 12**)
- Mesi con **30 giorni** (es. **4, 6, 9, 11**)
- **Febbraio**, che può avere **28 o 29 giorni**

Valori non validi: numeri interi non positivi e superiori a **12**

Test di equivalenza

I casi di test si ottengono selezionando **un valore valido per ogni classe di equivalenza** e combinando gli input

Equivalence class	Value for month input	Value for year input
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Month with 30 days, leap year	6 (June)	1904
Month with 28 or 29 days, non-leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

- Analisi dei valori limite (**boundary testing**): speciale test di equivalenza.

Boundary Testing

Il **test dei casi limite** è un caso speciale di **test di equivalenza**

- Gli elementi sono selezionati dai confini tra le classi di equivalenza
- Gli sviluppatori spesso trascurano i **casi speciali**

Esempio:

- Gli anni che sono **multipli di 4** sono anni bisestili
- Gli anni che sono **multipli di 100**, tuttavia, non sono bisestili, a meno che non siano anche **multipli di 400**.
- Il **2000** era un anno bisestile, mentre il **1900** non lo era.
- 2000 e 1900** sono buoni casi limite per l'**anno**
- 0 e 13** si trovano ai confini di classi non valide per il **mese**

Boundary Testing

Casi limite aggiuntivi per la verifica del metodo
getNumDaysInMonth()

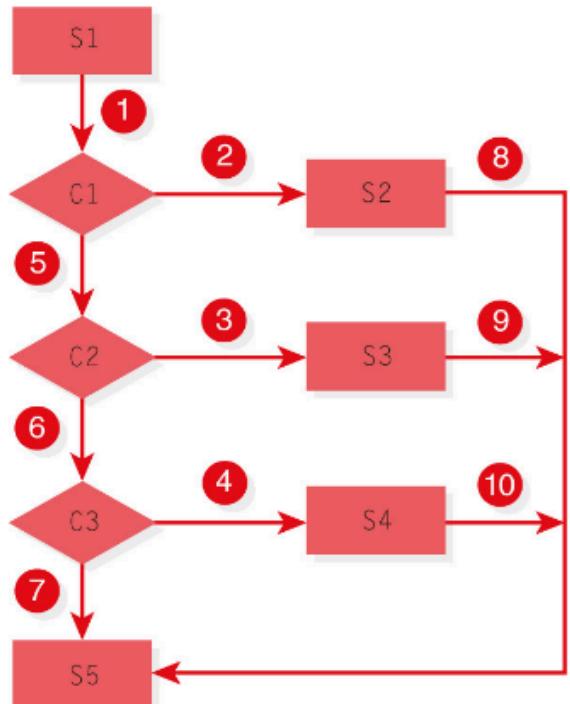
Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

6.4.2 White-box testing

I casi di test sono definiti usando la **specifica** e la **conoscenza del codice** del componente testato. Ha l'obiettivo di garantire che tutte le parti del codice siano adeguatamente sollecitate dai casi di test (copertura).

Per le tecniche di test white-box si parte del [diagramma di flusso](#) (control flow diagram).

Esempio di diagramma di flusso

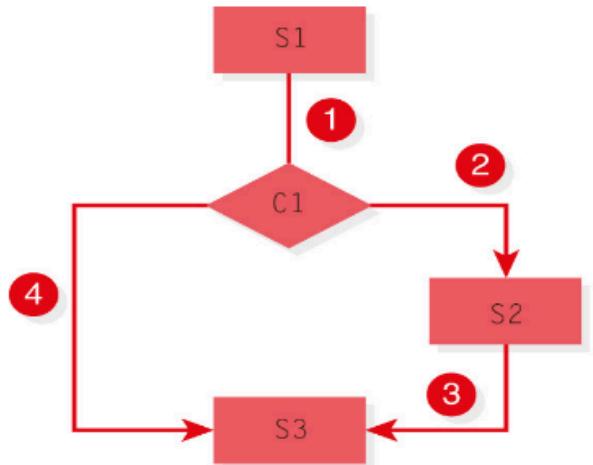


```
int function(char c) {  
    int res=0; // S1  
    if (c=='y'). // C1  
        res=1; // S2  
    else if (c=='n') // C2  
        res=2; // S3  
    else if (c=='?') // C3  
        res=3; // S4  
    return res; // S5  
}
```

6.4.2.1 Copertura delle istruzioni (statement coverage)

Si scelgono i test in modo da garantire che tutti i [nodi del diagramma](#) di flusso siano [attraversati](#) in almeno un caso di test.

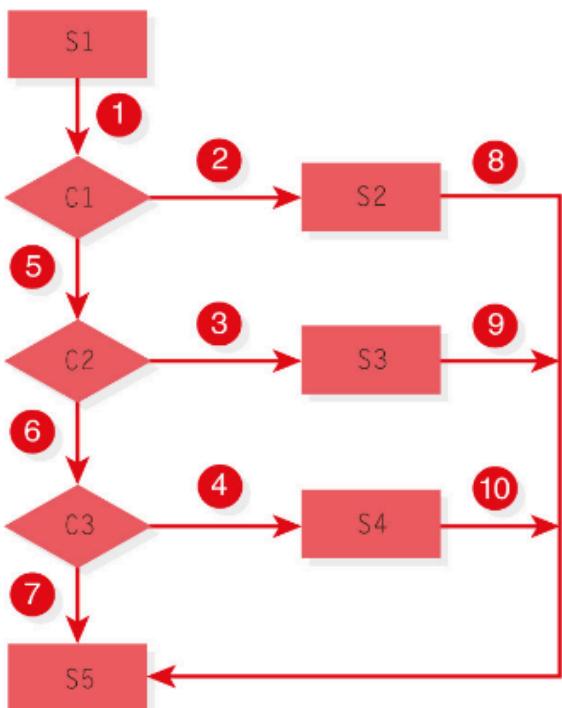
Esempio



```
int function(int a) {  
    int res=a/2; // S1  
    if (a%2 == 1) // C1  
        res++; // S2  
    return res; // S3  
}
```

Scegliendo il caso di test: $a = 7$
copriamo tutte le istruzioni di
questa funzione

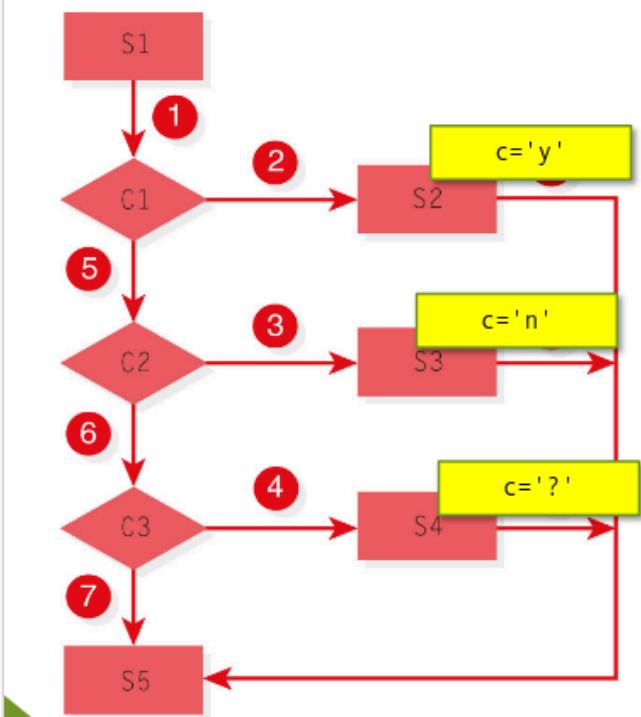
Esempio



```
int function(char c) {  
    int res=0; // S1  
    if (c=='y'). // C1  
        res=1; // S2  
    else if (c=='n') // C2  
        res=2; // S3  
    else if (c=='?') // C3  
        res=3; // S4  
    return res; // S5  
}
```

In questo caso non ci basta un singolo caso di test...

Esempio

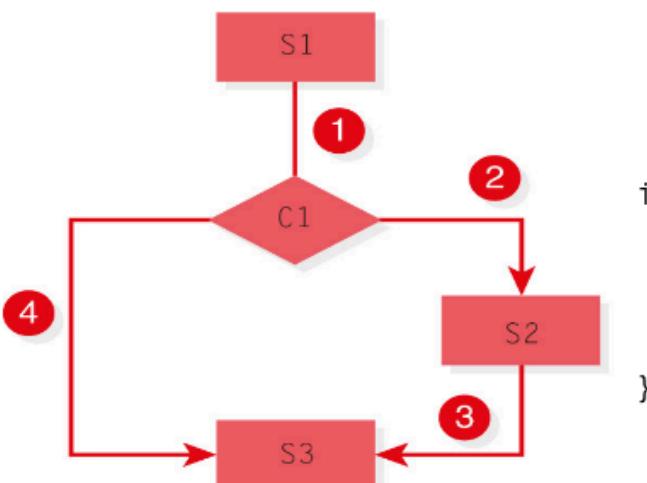


```
int function(char c) {  
    int res=0; // S1  
    if (c=='y'). // C1  
        res=1; // S2  
    else if (c=='n') // C2  
        res=2; // S3  
    else if (c=='?') // C3  
        res=3; // S4  
    return res; // S5
```

Con tre casi di test possiamo coprire tutte le istruzioni

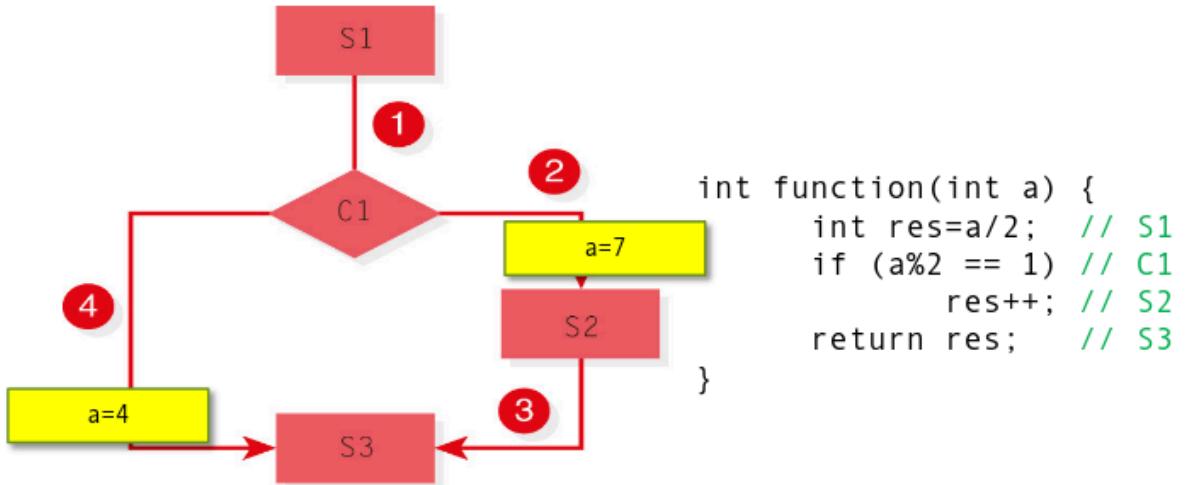
6.4.2.2 Copertura delle diramazioni (branch coverage)

Si scelgono i casi di test in modo da garantire che tutti gli **archi** del diagramma di flusso siano **attraversati** in almeno un caso di **test**. Equivale al fatto che, per ogni condizione nel diagramma di flusso, ci sia almeno un caso in cui la **condizione è vera** e uno in cui è **falsa**.

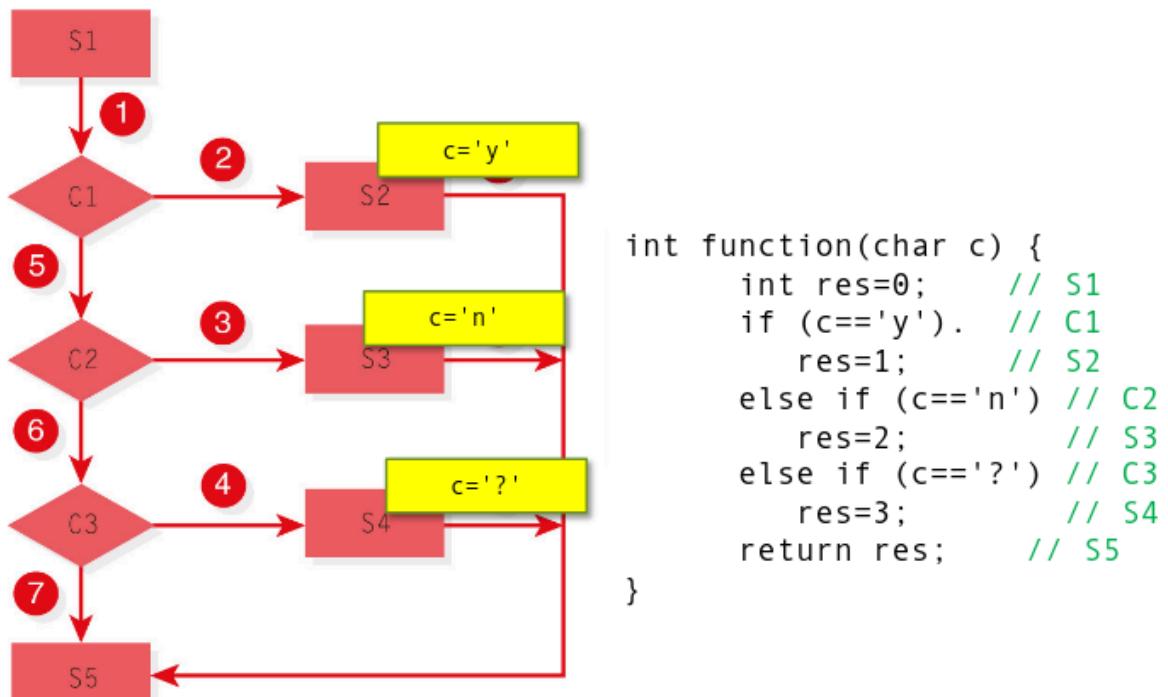


```
int function(int a) {  
    int res=a/2; // S1  
    if (a%2 == 1) // C1  
        res++; // S2  
    return res; // S3
```

Con il caso di test: a=7
NON copriamo tutte le diramazioni di questa funzione (non viene attraversato l'arco 4)



Con due casi di test riusciamo a ottenere la copertura delle diramazioni.



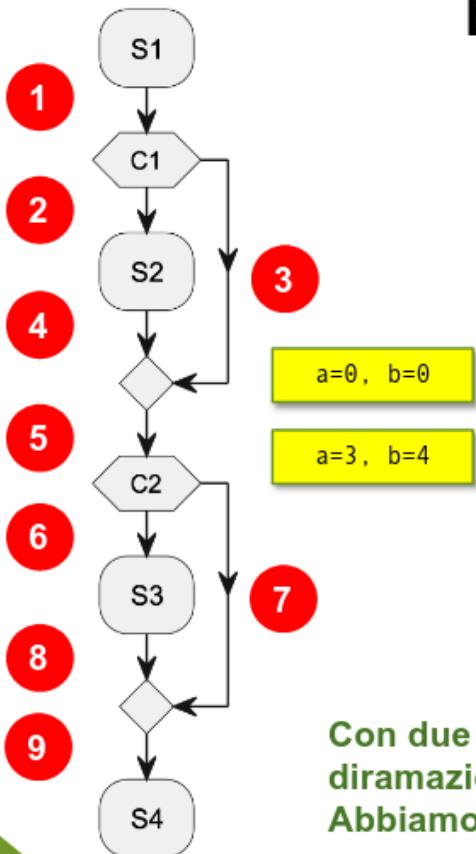
Con i tre casi di test indicati NON copriamo tutte le diramazioni (manca l'arco 7)

6.4.2.3 Copertura dei percorsi (path coverage)

Un **percorso** (path) è la **sequenza di nodi** (o equivalentemente, di archi) attraversata durante un'esecuzione del diagramma di flusso, partendo dal nodo iniziale e terminando nel nodo finale.

Si scelgono i casi di test in modo da garantire che tutti i percorsi possibili del diagramma di flusso siano attraversati in almeno un caso di test.

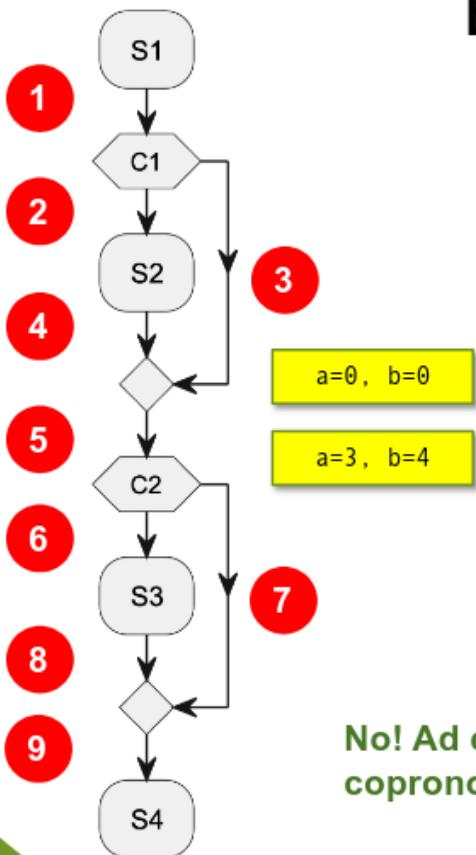
Esempio



```
int f(int a, int b) {  
    int r=0; // S1  
    if (a>0) // C1  
        r=a; // S2  
    if (b>0) // C2  
        r=b; // S3  
    return r; // S4  
}
```

Con due casi di test copriamo tutte le diramazioni di questa funzione.
Abbiamo coperto tutti i percorsi?

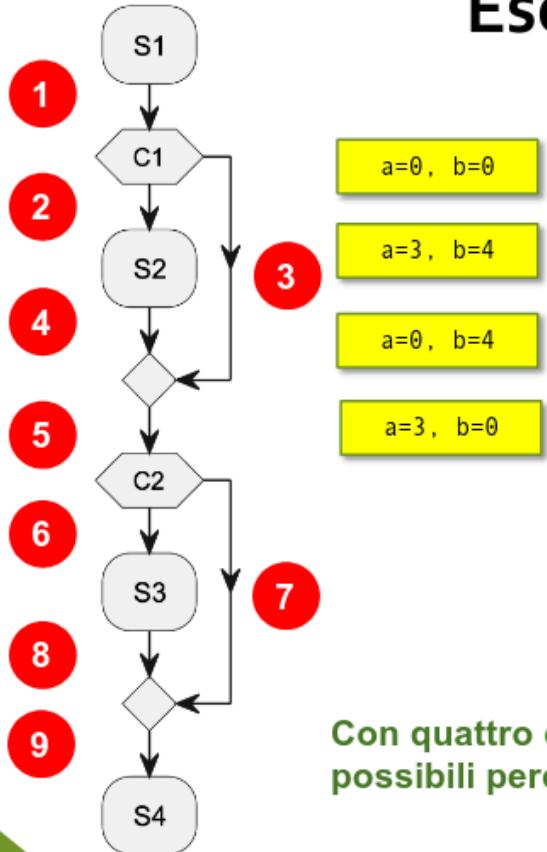
Esempio



```
int f(int a, int b) {  
    int r=0; // S1  
    if (a>0) // C1  
        r=a; // S2  
    if (b>0) // C2  
        r=b; // S3  
    return r; // S4  
}
```

No! Ad esempio, i casi di test non coprono il percorso S1-C1-C2-S3-S4

Esempio



```
int f(int a, int b) {  
    int r=0; // S1  
    if (a>0) // C1  
        r=a; // S2  
    if (b>0) // C2  
        r=b; // S3  
    return r; // S4  
}
```

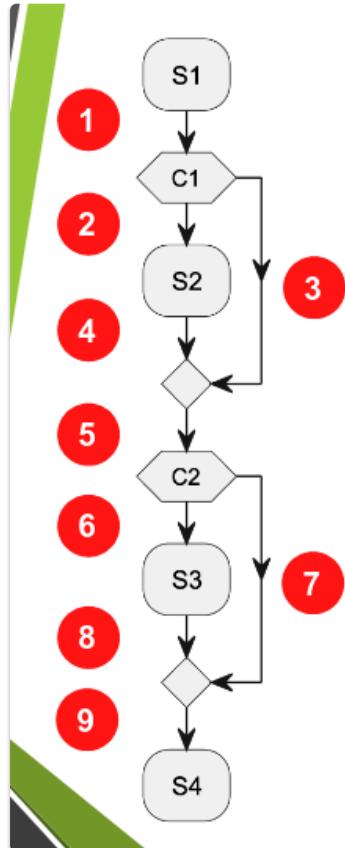
Con quattro casi di test copriamo tutti i possibili percorsi.

Per evitare la necessità di considerare infiniti percorsi nella scelta dei casi di test, il criterio di copertura dei percorsi viene semplificato richiedendo di coprire tutti i **percorsi linearmente indipendenti**.

Per definire i percorsi linearmente indipendenti, ad ogni percorso si associa un **vettore** che ha tante **componenti** quanti sono gli **archi** del diagramma di flusso e, i cui **elementi** indicano **quante volte** il percorso attraversa l'arco corrispondente. Un insieme di percorsi è linearmente indipendente se:

- I vettori associati ai percorsi sono linearmente indipendenti, ovvero nessuno dei vettori è ottenibile come combinazione lineare degli altri;

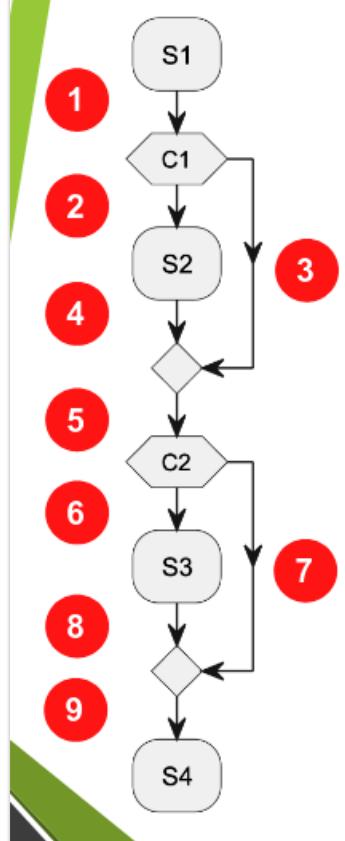
Esempio



	1	2	3	4	5	6	7	8	9
P1	1	1	0	1	1	1	0	1	1
P2	1	0	1	0	1	0	1	0	1
P3	1	0	1	0	1	1	0	1	1
P4	1	1	0	1	1	0	1	0	1

Consideriamo i vettori corrispondenti
(poiché in questo diagramma ogni arco viene attraversato al più una volta, gli elementi dei vettori saranno 0 o 1)

Esempio



	1	2	3	4	5	6	7	8	9
P1	1	1	0	1	1	1	0	1	1
P2	1	0	1	0	1	0	1	0	1
P3	1	0	1	0	1	1	0	1	1
P4	1	1	0	1	1	0	1	0	1

Possiamo ottenere il vettore di P4 come:
$$P4 = P1 + P2 - P3$$

Quindi il percorso P4 non è linearmente indipendente da P1, P2, P3, e possiamo evitare di testarlo.

6.4.2.3.1 Complessità Ciclomatica

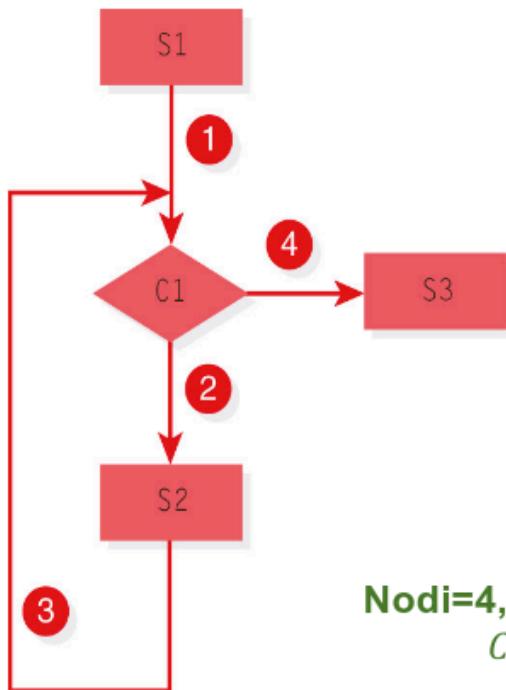
La **complessità ciclomatica** rappresenta il massimo numero di percorsi indipendenti attraverso il diagramma di flusso:

$$CC = E - N + 1$$

- E : numero di archi;

- N : numero di nodi

Esempio



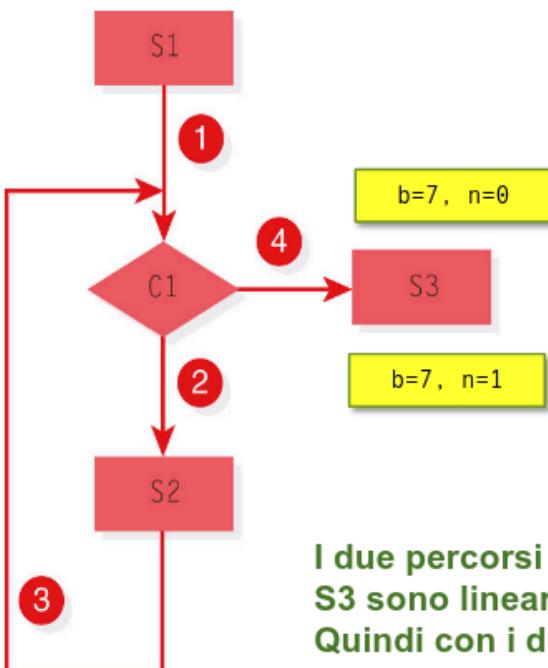
```

int pot(int b, int n) {
    int p=1;           // S1
    while (n>0) {    // C1
        p*=b;          // S2
        n--;
    }
    return p;          // S3
}
  
```

Nodi=4, Archi=4

$$CC = 4 - 4 + 2 = 2$$

Esempio



	1	2	3	4
P1	1	0	0	1
P2	1	1	1	1

I due percorsi $P1=S1-C1-S3$ e $P2=S1-C1-S2-C1-S3$ sono linearmente indipendenti.
Quindi con i due casi di test indicati abbiamo la copertura dei percorsi.

6.5 Sviluppo guidato dai test (Test Driven Development)

Metodologia di sviluppo software in cui i **test unitari** vengono **sviluppati prima del codice**. Si basa su tre punti:
43. Test Early

44. Test Often

45. Test Automatically

Vantaggi del TDD

Risultati di un sondaggio tra sviluppatori

95.8% | Reduced Debugging Efforts

92% | TDD yielded high-quality code

87.5% | Better requirements understanding

79% | Promoted simpler design

78% | Improved overall productivity

50% | Decreased overall development time

6.5.1 Test Early

46. Quando si sviluppa un nuovo metodo, la sua interfaccia si scrive con un'implementazione vuota;

47. Si scrive il codice per testare la funzione (unit test);

48. Si implementa la funzione e si fa in modo che superi i relativi test;

49. Si controlla che tutte le altre funzioni già esistenti superino i rispettivi test;

50. Solo a questo punto si può aggiungere una nuova funzione

Test Early

Vantaggi

- Gli sviluppatori verificano la loro comprensione dei requisiti *prima* di implementare la funzione (black-box testing)
- Gli sviluppatori verificano se l'interfaccia progettata per la funzione è realmente usabile *prima* di implementarla (il codice di test *usa* la funzione da testare)
- Quando un test fallisce, è più facile individuare la causa (è stato modificato poco codice, ed è "fresco" nella mente degli sviluppatori)
- La scrittura dei test non è rimandata alla fine, quando la pressione della scadenza potrebbe spingere gli sviluppatori a cercare "scorciatoie"

6.5.2 Test Often

Ogni volta che il sistema viene modificato, gli unit test vengono eseguiti nuovamente. Il nuovo codice viene integrato nel sistema solamente se tutti gli unit test sono superati.

Test Often

Vantaggi

- Si scopre subito se la modifica effettuata ha compromesso il funzionamento di un'altra parte del codice
- Gli sviluppatori possono essere più tranquilli quando modificano il codice (ad esempio, per un refactoring), sapendo che i test rileveranno eventuali errori introdotti
- Si evita il "ritorno" di bug già risolti in passato (la cosiddetta "regressione" del codice)
 - Quindi gli unit test servono anche come test di regressione

6.5.3 Test Automatically

Perché tutto questo sia possibile, i test devono poter essere eseguiti automaticamente (senza intervento dello sviluppatore).

Test Automatically

Vantaggi

- Se i test sono automatici, gli sviluppatori sono incoraggiati a eseguirli spesso
- I test possono anche essere eseguiti periodicamente (es. ogni notte) da tool di automazione del processo di build
- L'esigenza di rendere automatici i test spinge gli sviluppatori a separare la logica dell'applicazione dall'interfaccia utente, e più in generale a privilegiare in fase di progettazione soluzioni più modulari

6.5.4 TDD e bug

I test non consentono di rilevare ogni problema presente nel software. Quando viene scoperto un bug non rilevato durante la fase di testing:

51. Si scrive un nuovo test che riproduce il bug trovato;
52. La causa del bug viene rilevata e corretta;
53. Si lanciano nuovamente tutti i test

6.6 Test Automatizzati

I casi di test sono specificati in termini di:

- Sequenza di dati in input
- Sequenza di output prevista

L'infrastruttura di testing esegue i test e confronta l'output ottenuto con quello atteso. In java si può usare Junit5.

7 Debugging

- **Logging**: si usano istruzioni di stampa per tracciare il flusso di esecuzione e monitorare il valore delle variabili;
- **Debugger** (simbolici): permette di esaminare l'esecuzione del programma. Lavora con simboli di alto livello del codice sorgente piuttosto che indirizzi di memoria o istruzioni binarie.

7.1 Processo di debugging

- Si impostano i punti di interruzione (**breakpoint**) in corrispondenza di linee specifiche del codice sorgente
- Una volta arrivato al breakpoint il sistema **sospende** l'esecuzione
- È possibile ispezionare o modificare il valore delle variabili
- È possibile riprendere l'esecuzione un passo alla volta

8 Build Automation

Build: attività necessarie a produrre tutti i file necessari all'esecuzione di un programma in un determinato ambiente di esecuzione. Esistono degli strumenti come make e maven che consentono di automatizzare la fase di building.

8.1 Build in C: MAKE

Le operazioni di building sono specificate in un file detto "Makefile", composto da una serie di regole. Ogni regola specifica:

- Target: nome del file che deve essere costruito durante la fase di building;
- Prerequisiti: nomi dei file da utilizzare per la generazione del target;
- Comandi: Sequenza di comandi che ha il compito di generare il target a partire dai requisiti

Formato del Makefile

- Sintassi:

```
target: prerequisito ...
        comando
        comando
        comando
        ...
        
```

- Esempio:

```
scacchiera.o: scacchiera.c scacchiera.h
        cc -c scacchiera.c
        
```

Comando: `make [target]`. Se il target non è specificato costruisce il primo target del Makefile.

Makefile:

```
tictactoe: tictactoe.o scacchiera.o strategia.o interfaccia_utente.o  
    cc -o tictactoe tictactoe.o scacchiera.o strategia.o \\\n        interfaccia_utente.o  
  
tictactoe.o: tictactoe.c scacchiera.h strategia.h interfaccia_utente.h  
    cc -c tictactoe.c  
  
scacchiera.o: scacchiera.c scacchiera.h  
    cc -c scacchiera.c  
  
strategia.o: strategia.c strategia.h scacchiera.h  
    cc -c strategia.c  
  
interfaccia_utente.o: interfaccia_utente.c interfaccia_utente.h \\  
    scacchiera.h  
    cc -c interfaccia_utente.c
```

Se l'esecuzione di un comando di una regola restituisce un exit status diverso da zero, make termina l'esecuzione della regola (non esegue i comandi successivi di quella regola, e non esegue le eventuali regole che avevano richiamato la regola corrente).

8.1.1 Variabili

All'interno del Makefile si possono definire delle variabili.

- Definizione di una variabile: `nome_variabile = valore`
- Uso di una variabile: `$(variabile)`

54. `$@` : sostituito dal target;

55. `$^` : sostituito dall'elenco dei prerequisiti;

56. `$<` : sostituito dal primo dei requisiti.

Makefile:

```
# Comando usato per compilare un file .c  
COMP=cc -c  
# Elenco dei file oggetto  
OBJS=tictactoe.o scacchiera.o strategia.o interfaccia_utente.o  
  
tictactoe: $(OBJS)  
    cc -o $@ $^  
  
tictactoe.o: tictactoe.c scacchiera.h strategia.h interfaccia_utente.h  
    $(COMP) $<  
  
scacchiera.o: scacchiera.c scacchiera.h  
    $(COMP) $<  
  
# ... etc etc etc
```

8.2 Build in Java : Maven

Scarica ed installa automaticamente le librerie necessarie al building. A differenza di Make, la configurazione di Maven non contiene comandi per eseguire il building bensì informazioni sulle dipendenze e sui plug-in da

usare. Tali informazioni sono salvate nel file *pom.xml*.

Il file pom.xml

- Esempio:

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>it.unisa.diem.softeng</groupId>
    <artifactId>tictactoe</artifactId>
    <version>1.0</version>
</project>
```

8.2.1 Comandi

- Creazione progetto vuoto: `mvn archetype:generate`
 - `groupId` : identificativo azienda/organizzazione. Formato package Java.
 - `artifactId` : identificativo del programma da creare;
 - `version` : versione del programma da creare;
- `mvn compile`
- `mvn test`
- `mvn package`
- `vn clean`