

## Riassunto 15

Progetti tristemente falliti

Denver Airport : Sistema smistamento bagagli.

Risultati :

- Inaugurazione dell'aeroporto ritardata 16 mesi (a 1 milione \$ al giorno)
- sovraccarico di 3,2 miliardi di dollari rispetto ai preventivi

Progettazione difettosa :

- jams (mancanza di sincronizzazione)
- No fault tolerance  $\Rightarrow$  Il sistema ripartiva da 0 quando c'erano problemi
- si perdeva traccia di quali carrelli fossero pieni e quali vuoti dopo un riavvio dovuto a un jam

Therac - 25 : 3 persone uccise da sovredosaggi di radiazioni

Problema causato da un difetto latente. L'operatore metteva dati troppo velocemente quindi non riusciva il sistema a controllare i valori mancanti di controlli sui valori immessi

Cause : • errori nel sistema SW, e di interfacciamento SW/HW (erronea integrazione di componenti SW preesistenti nel Therac- 20).

- Poca Robustezza
- Difetto latente

Sistema Autominile Fiatbot : Programmato per funzionare al massimo 14h

intenzionalmente

Cause : • Fu usato per 100h  $\Rightarrow$  Errori nell'orologio interno del sistema accumulati al punto da rendere inservibile il sistema di tracciamento dei mimi da abbattere

- Scarsa robustezza

London Ambulance Service : Sistema per gestire le ambulanze

(ottimizzazione dei percorsi, guida vocale agli autisti)

Risultati :

- 3 versioni, costo totale: 11 000 000 Euro
- L'ultima versione abbandonata dopo soli 3 giorni d'uso

Problemi :

- interfaccia utente inadeguata
  - poco addestramento utenti  $\Rightarrow$  Non avvertivano quando veniva soccorso un paziente, perché l'operatore doveva accendere un pc e comunicarlo nel mezzo di un'emergenza.
  - sovraccarico non considerato
  - nessuna procedura di backup
  - scarsa verifica del sistema
- pì ambulanze nello stesso posto.

Errore di Progettazione (Analisi dei requisiti errata)

Aniane 5 : Il sistema era progettato per l'aniane 4, cioè tentava di convertire la velocità laterale del missile dal formato a 64 bit al formato a 16 bit.

L'aniane 5 eraendo più veloce. Causava un overflow.

L'overflow causava lo spegnimento del sistema di guida e trasferimento del controllo al 2° sistema di guida, ma erendo progettato allo stesso modo, era andato anche lui in tilt

Problema : Test con dati vecchi  $\Rightarrow$  Tanto è mezzo per capire il malfunzionamento

Carattere telex : Disponibili apple crashavano



**Caso toyota:** le macchine acceleravano involontariamente (senza che il guidatore accelerasse)

**problema:** Blocco dell'acceleratore elettronico che controlla la valvola a farfalle.

L'acceleratore si rompe con la valvola aperta e causa l'aumento della potenza richiesta per frenare. Quindi i guidatori non potevano frenare.

- Qualcuno non ha seguito gli standard ingegneristici (Progettuali)

**Caso di successo:**

**linea 14 Metro Parigi:** Prima linea integralmente automatizzata

- 8 km. 7 stazioni. 19 treni. Intervallo tra 2 treni: 85 secondi.
- Siemens Transportation Systems
- B-method di Abrial.
- Abstract machines
- Generazione di codice
  - ADA, C, C++.



**Aspetti Storici.** Negli anni 60 ci fu la curiosità del software cioè che la loro qualità era bassa e quindi nacque l'ingegneria del software

**Ingegneria del software** : Disciplina che ha lo scopo di produrre fault-free software , consegnato nei tempi previsti, che rispetti il budget iniziale , che soddisfi le necessità del committente, facile da modificare.

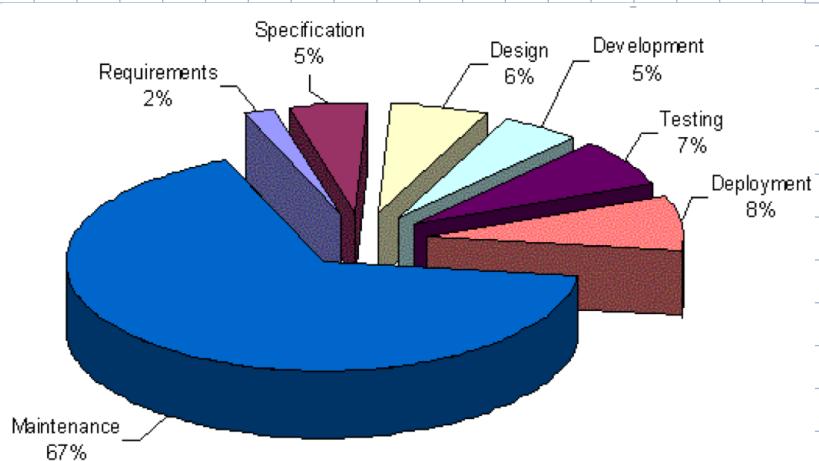
**OSSERVAZIONE:** Tra le cause principali di abbandono di un software l'ignoranza tecnologica è l'orma per importanza.

Tra le prime troviamo: Requisiti incompleti, Scarso coinvolgimento degli utenti, Mancanza di risorse ecc. ..

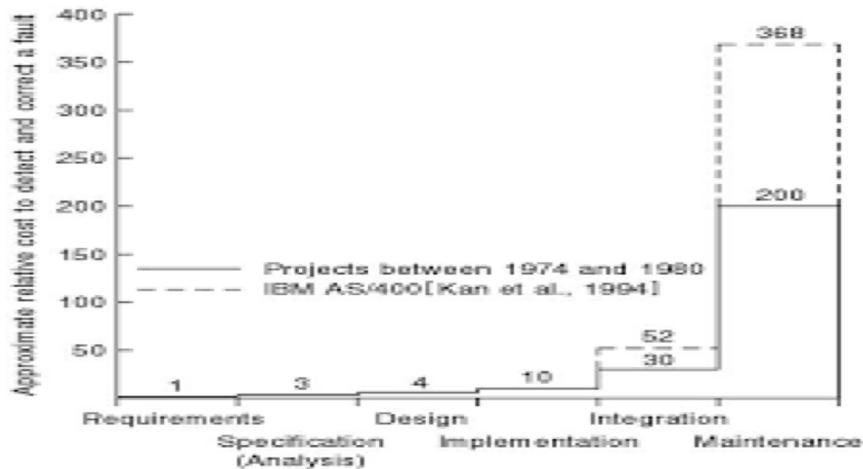
### SPECIFICITÀ DEL SOFTWARE:

- Non è vincolato da materiali , governato da leggi fisiche o da processi manifatturieri
- Si sviluppa , non si fabbrica
- Si deteriora , non si consuma
- Sposto mi anembra , ma larga parte è ancora ad-hoc
- Quando crasha si riavvia e **fault tolerance** ( Si minimizza l'effetto di un fallimento)
- Manutenzione (può essere modificato a seconda delle necessità)

### Costi del SOFTWARE



È importante trovare gli errori durante l'analisi dei requisiti perché il costo per risolverli aumenta per ogni fase del processo software



**Mantenzione** : Incluse tutti i cambiamenti al prodotto software

→ **Mantenzione cognitiva (20%)** : Rimuove gli errori lasciando invariata la specifica

→ **Mantenzione Migliorativa** : Modifica alla specifica e implementazione  
**Perfettiva (60%)** : Modifiche per migliorare la qualità del SW  
introduzione di nuove funzionalità  
miglioramento delle funzionalità esistenti

**Adattiva (20%)** : Modifiche dovute a cambiamenti HW, SO, ambiente legislativo

**Lavoro di team** : lavorare in team è essenziale ma può portare a problemi di coordinazione interna.

**Stakeholders di un prodotto** : 1) **Fornitore** : Chi lo sviluppa

2) **Committente** : chi lo richiede (e paga)

3) **Utente** : chi lo usa

Processo Software: Percorso da seguire per sviluppare un prodotto o sistema software



Inizia con l'esplorazione dell'idea e termina con la disseminazione del software

Fasi : 1) Analisi dei requisiti

2) Specifica

3) Progettazione

4) Codifica (Implementazione)

5) Testing (Integrazione)

6) Manutenzione (Mantenimento)

7) Ritiro

Incluse anche :

1) Strumenti e tecniche per lo sviluppo

2) Professionisti del SW coinvolti

Modellare il Proceso: Il modello decide

- 1) Come strutturo le varie attività (suddiviso in attività il processo)
- 2) Qual' è l'ordine in cui vengono fatte le attività
- 3) Cosa deve produrre ogni attività
- 4) Quando finisce ogni attività

Evoluzione temporale dei modelli: cicli di vita dal più "vecchio" a quelli usati oggi (ognuno di essi ha aggiunto una caratteristica importante)

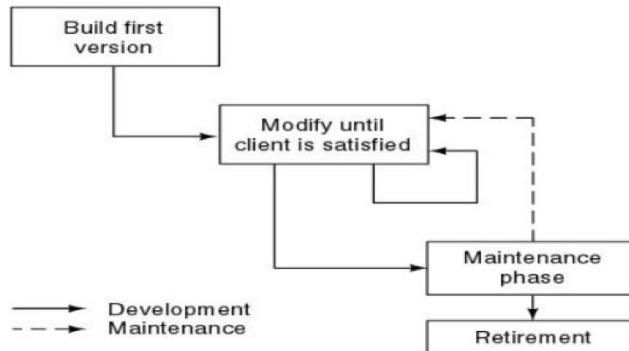
1) Build-and-fix model: Prodotto sviluppato senza specifica e senza progettazione



Non è un modello perché non divide in più fasi la vita del SW

Aspetti: Adatto per piccoli prodotti

e la manutenzione è difficile senza specifica o documentazione

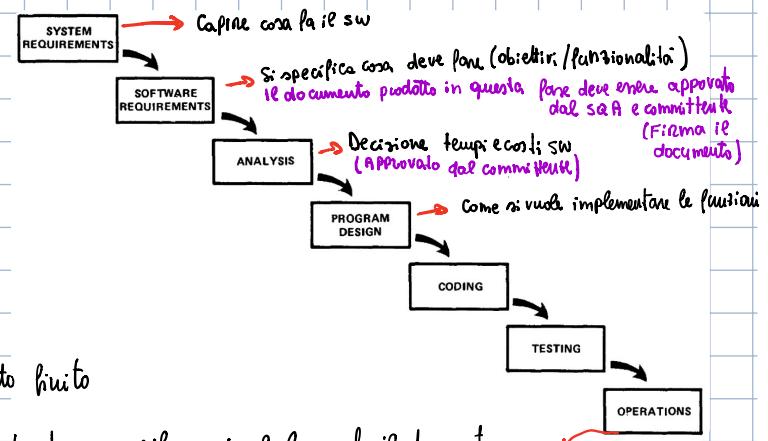


Nel caso ci fosse un errore a livello di progettazione viene a costare molto.

## 2) Modello a cascata → Evidenzia l'importanza delle fasi di analisi e progettazione prima di passare alla codifica

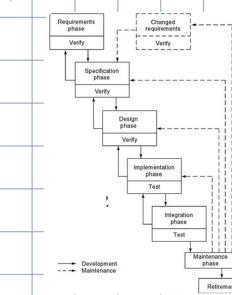


Ogni fase produce un documento che deve essere approvato da un gruppo di valutatori (Software Quality Group) SQA per poter passare alla fase successiva.



### Analisi:

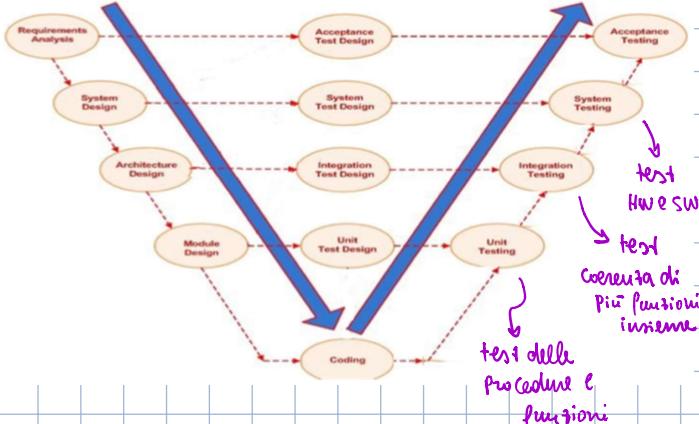
- 1) Eccessiva produzione di documenti (sia positiva che negativa)
- 2) Manca l'interazione con il cliente che vede solo il prodotto finito
- 3) Nella versione con feedback loop, ogni volta che torniamo indietro bisogna rifare sia la fase che il documento
- 4) Definisce e struttura le fasi del SW



Moduli messi insieme e inviati al committente

## 3) Modello a V

→ prodotto soddisfa i requisiti?



Ad ogni fase del processo viene sviluppato contemporaneamente il test in modo da testare il prodotto il prima possibile.

I test vengono poi svolti dopo la fase di coding.

## 4) Rapid prototyping : Costruire un prototipo anche prima di progettazione da dare al committente.

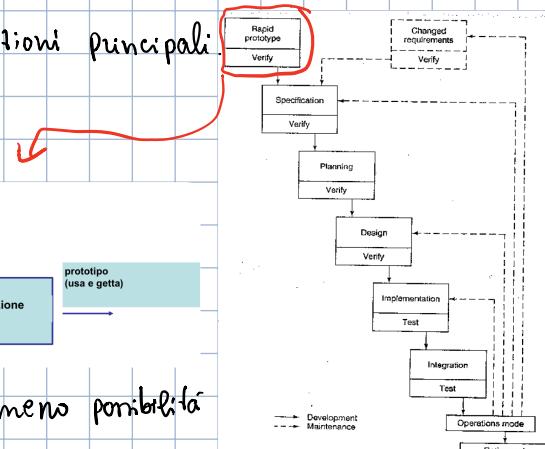
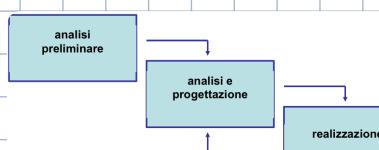


→ con le funzioni principali

- 1) Utile quando non sono chiari i requisiti;

- 2) Aiuta il cliente a descrivere meglio i requisiti;

- 3) Utile in fase di specifica



Usando il prototipo ho meno possibilità di avere feedback loop

Modelli visti fino ad ora sono poco iterativi: Modello a cascata, Modello a V, Rapid Prototyping

↓  
Difficilmente tornano alle fasi precedenti

### 5) Modello incrementale:

I requisiti e il progetto sono definiti inizialmente, poi il sistema è implementato integrato e testato con una serie di passaggi incrementali.

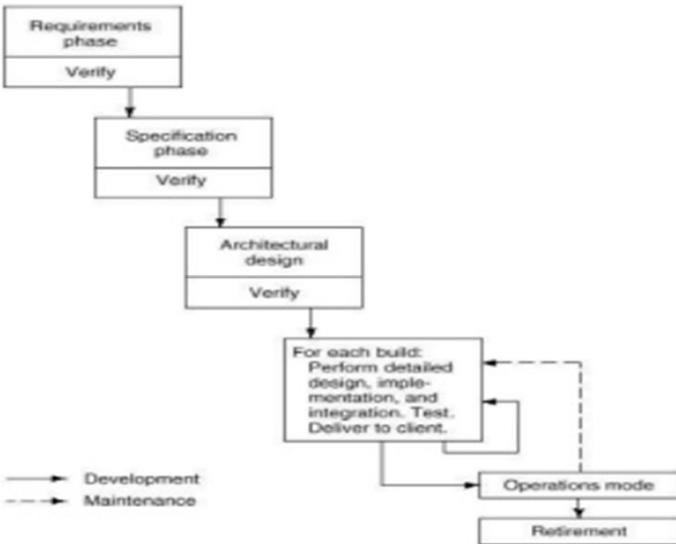
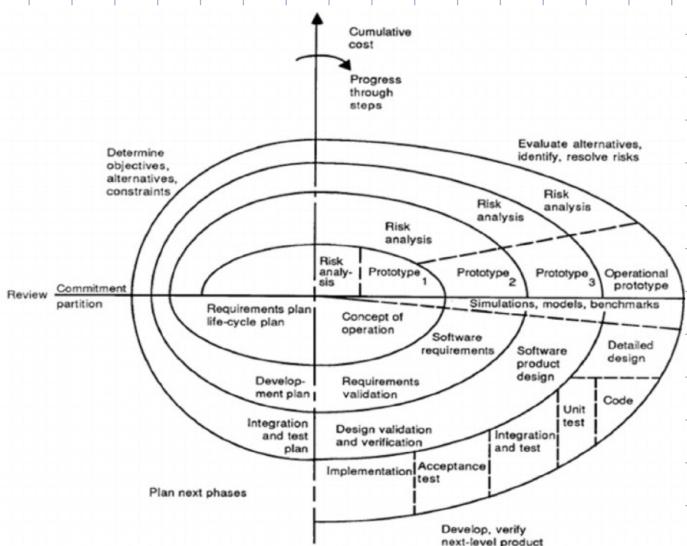
Sono definiti le funzionalità principali e poi si aggiungono altre man mano.

#### Analisi:

- Si applica con requisiti stabili
- Se non progettato bene diventa un build-and-fix → Aggiungere le funzionalità man mano
- Il prodotto non lasciato più essere già utilizzato dal committente → Può essere già pagato
- Fa abitudine più piano all'uso di ero
- Manutenzione = Aggiungere nuove funzionalità e testarle

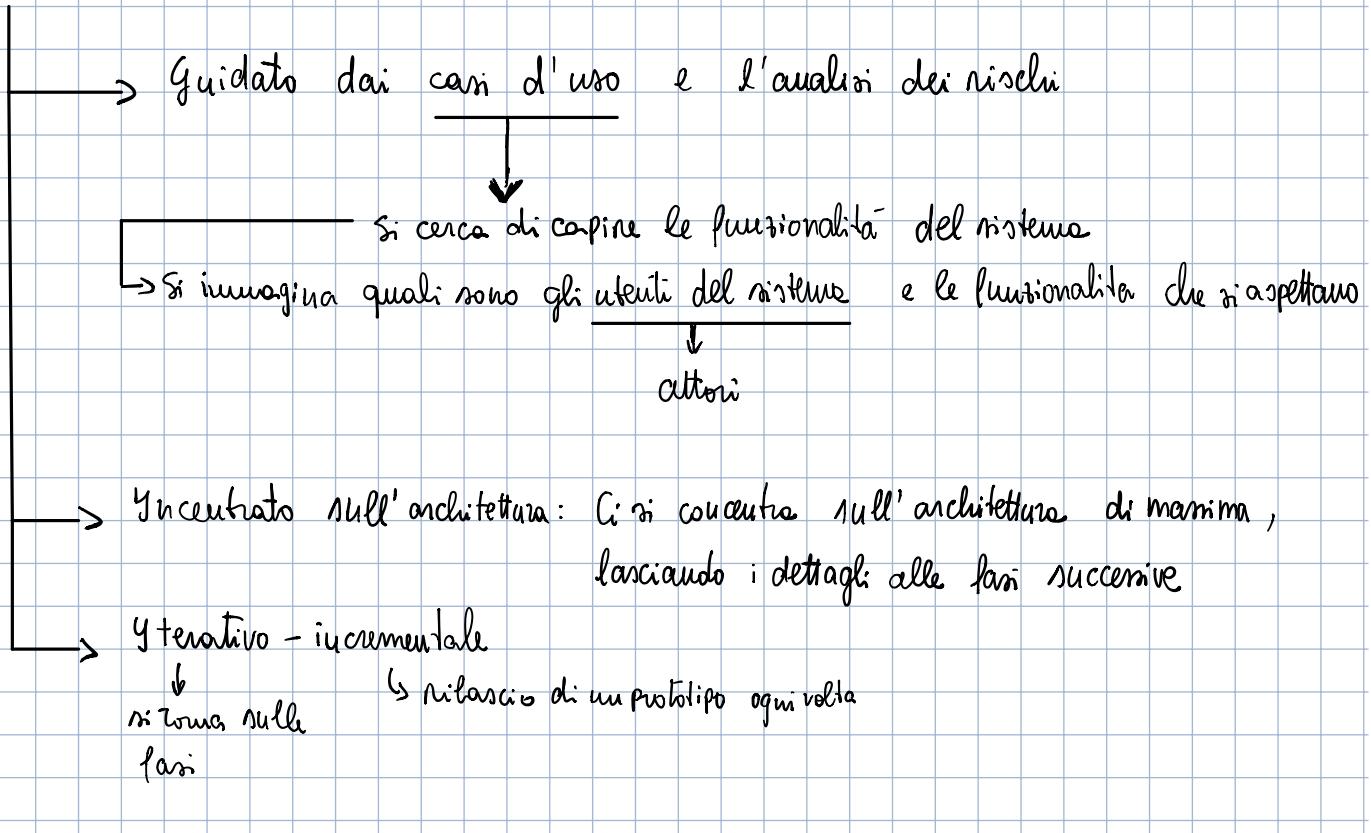
### 6) Modello a spirale: Mette al centro l'analisi dei rischi → Minimizzare i danni dovuti ai rischi

↳ Adatto allo sviluppo di un prodotto



- Ogni fase che andiamo a svolgere inizia con l'analisi dei rischi.
- Iterativo perché partiamo sempre dall'analisi dei rischi.
- Prevede maggior comunicazione e confronto con il committente → A ogni fase c'è il confronto con il committente

→) Undefined Process: Costruiamo dei grafici che siano il più comprensibili (UML)



Colpisce in:

- Serie di cicli in questa forma →



- Ogni ciclo si conclude con il rilascio di un prototipo

- Ogni ciclo ha diverse iterazioni

↓  
Raggruppate in 4 fasi

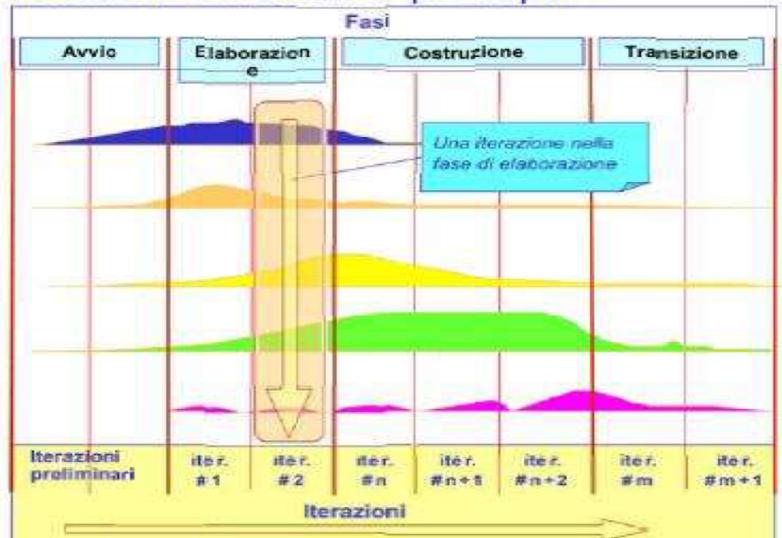
- Ogni fase termina con la decisione del manager di continuare o terminare il progetto



- Ogni fase ha più iterazioni

↓  
mini undefined process

### Fasi, iterazioni e workflow principali



#### Avvio

- Fattibilità; Analisi dei rischi; Requisiti essenziali per definire il contesto del sistema; Eventuale prototipo

#### Elaborazione

- Analisi dei requisiti; Analisi dei rischi; Sviluppo di un'architettura base; Piano per la fase di costruzione

#### Costruzione

- analisi, disegno, implementazione, testing

#### Transizione

- Beta testing, aggiustamento delle prestazioni, creazione di documentazione aggiuntiva, attività di formazione, guide utenti, creazione di un kit per la vendita

## Progetti agili (Particolare undefined Proces) → Metodo per lo sviluppo SW che

- Adatto a progetti con non più di 50 persone

- Si basa sui principi del manifesto agile

↓

### ■ Comunicazione:

- le persone e le interazioni sono più importanti di processi e strumenti
- tutti possono parlare con tutti, e.g. l'ultimo dei programmati con il cliente
- la comunicazione tra gli attori di un progetto sw è la miglior risorsa del progetto
- collaborare con i clienti al di là del contratto
  - la collaborazione diretta offre risultati migliori dei rapporti contrattuali;

### ■ Simplicità: analisti mantengano la descrizione formale il più semplice e chiara possibile

- è più importante avere software funzionante che documentazione
- bisogna mantenere il codice semplice e avanzato tecnicamente, riducendo la documentazione al minimo indispensabile

### ■ Feedback

- rilasciare nuove versioni del software ad intervalli frequenti
- sin dal primo giorno si testa il codice

### ■ Coraggio

- dare in uso il sistema il prima possibile e implementare i cambiamenti richiesti man mano
- rispondere ai cambiamenti più che aderire al progetto

coinvolge quanto più possibile il  
committente



## Extreme Programming (tipo di Proceso agile) → Principi agile portati all'estremo

- Pianificazione flessibile: Basata su scenari proposti dagli utenti e coinvolge i programmati
- Rilasci frequenti: 2-4 settimane e si inizia una nuova pianificazione con l'analisi dei rischi
- Progetti semplici: Comprensibili a tutti
- Testing: Di unità e sistemi (basati sugli scenari)
- Test Driven Development: Casi di test prima del codice
- Clienti sempre a disposizione (cinque ogni settimana)
- Programmazione a coppie: Uno scrive il codice, uno lo correge
- No lavoro sterodinamico
- Collettivizzazione del codice: Tutti possono vedere il codice di tutti
- Code refactoring: Modificare senza cambiare il comportamento e Codice auto-explicativo
- Daily stand up meeting: Meeting tutti i giorni in piedi

**SCRUM (Proceso agile)** : Deriva dal Rugby ed indica un processo in cui un insieme di persone si muove all'unisono per raggiungere un obiettivo predefinito.

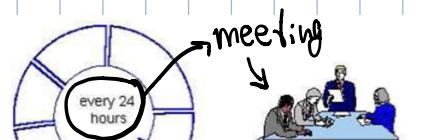
- Adottato per gestire e controllare lo sviluppo di un SW
- Iterativo - incrementale
- Fornisce alla fine di ogni iterazione un prototipo potentialmente rilasciabile

Fasi:

1) Fase pre-game: Definizione del sistema creando una **Product Backlog list** che contiene le funzionalità da implementare viene creato un design di alto livello del sistema in base alla PBL

2) Fase di sviluppo: (Avviene una serie di sprints)

Funzionalità da sviluppare presa dalla lavagna



Scrum: 15 minute daily meeting.  
Teams member respond to basics:  
1) What did you do since last Scrum Meeting?  
2) Do you have any obstacles?  
3) What will you do before next meeting?

New functionality is demonstrated at end of sprint



↑ not done ↑ in progress ↑ pending test ↑ in test ↑ fatto

Quando qualcuno aggiunge un'attività ne viene tolta un'altra

tempo sprint (1 sett - 30 gg)

3) Fase post-game: Chiusura definitiva delle release

Membri:

1) Product owner:



Persona a cui fa riferimento tutti i soggetti

interessati al Progetto compreso il cliente finale

- Accetta / rigetta risultati di un lavoro

- termina uno sprint se necessario

## 2) Membri del team:

- Gruppi di persone (7-9)



- tutti fanno tutto quando è richiesto → Vanno dove c'è il lavoro

- costituiscono il prodotto

- Non hanno un capo

- decidono cosa fare negli sprints

- Ogni persona realizza una cosa alla volta (No multi-tasking)

- Se possibile il gruppo nella stessa sede/ufficio

## 3) Scrum master: Funzione di mettere d'accordo tutti i componenti del team e fornire supporto psicologico

Attention: Non ha autorità sul team

Studio di fattibilità: Valutazione dei costi e dei benefici di una possibile attività di produzione

1) Fattibilità tecnologica: Strumenti per la realizzazione, algoritmi, HW...

2) Aspetti economici e di mercato: Confronto tra mercato attuale e futuro, costo ecc.

Analisi dei requisiti: Studiare e definire il problema da risolvere (cosa deve fare il sistema e non come)

Prodotto: Documento (Descrizione del dominio / dei requisiti)

↳ Può produrre anche: Manuale utente, casi di test  
(anche in parallelo)

scritto in  
linguaggio naturale

Dominio: Campo di applicazione del prodotto  
(Non formale/ambiguo)

→ Gli analisti devono avere conoscenza sul dominio prima di incontrare il committente:  
- Per porre le domande giuste  
- Per distinguere i termini usati dal committente

→ Per definire il dominio: Glossario  
→ Costituito durante lo studio del dominio e ammesso man mano  
mejorarlo prima di incontrare il committente

**Requisiti:** Proprietà che deve essere garantita dal sistema per soddisfare una necessità di un utente (Funzionalità/qualità per l'utente)

1) Requisito funzionale: Descrivono le funzionalità che il sistema deve realizzare

↑  
É bene tenerli separati  
↓

- Azioni che il sistema deve compiere
- Come il sistema SW reagisce a specifici input
- Come si comporta in situazioni particolari

2) Requisiti non funzionali: Descrivono le proprietà del sistema SW in (di qualità) relazione a determinati servizi o funzioni e possono anche essere relativi al processo;

- Caratteristiche di qualità: efficienza, affidabilità ...
- Caratteristiche del processo: standard di processo, di sviluppo linguaggi di prog. ecc..
- Caratteristiche esterne: vicoli, leggi, ecc..
- HW

### Approchi diversi all'analisi dei requisiti

- Basato su Documenti in linguaggio naturale  $\Rightarrow$  - glorioso

- Specifica dei requisiti

Funzionali  
Non Funzionali

- Basato su linguaggi formali  $\longrightarrow$  - UML

Modello  
del dominio

caso d'uso (req. funzionali)

1) Documento in linguaggio naturale → output: Documento dei requisiti  
(comprende il glossario)

Documento dei requisiti: - Precede la stipula del contratto tra sviluppatore e committente e ne è parte integrante



- Specifica cosa deve fare e i vincoli del prodotto
- Specifica deadline per le consegne di un prodotto

## 1) Acquisizione:

- Interviste: team di analisti incontrano i membri dell'organizz. del committente

→ Strutturate: Domande con risposta limitata

→ Non strutturate: Domande con risposta aperta

- Questionari a risposte multiple

- Prototipi

- Observazione dipendenti

- Studio di documenti

- Casi d'uso (per req. funzionali)

## 2) Elaborazione: Requisiti espauri e raffinati → Def. del Documento dei requisiti



### Struttura:

- 1) Introduzione: Descrizione del sistema
  - 2) Glossario: termini utilizzati
  - 3) Definizione requisiti funzionali
  - 4) Definizione requisiti non funzionali
  - 5) Architettura
  - 6) Specifica di system and sw requirements → Specifica Dettagliata req. funz.
  - 7) Modelli astratti del Sistema → Modelli formali/semiformali che descrivono ciascuno un punto di vista: controllo dati, funz.
  - 8) Evoluzione del Sistema → Previsione succ. cambiamenti
  - 9) Appendici → Descrizione HW, requisiti di DB, Manuale utente, Caso di test
  - 10) Indici → Lemmanio: lista di termini, con puntatori ai req. che li usano
- Definizione dei termini chiave del dominio
- Validazione dei requisiti

3) Convalida dei requisiti: Validatione di un documento già strutturato in linguaggio naturale

- Difetti da evitare:
- 1) Omissioni → Mancanza presenza di un requisito (incompletezza)
  - 2) Inconsistente → Controddizioni fra requisiti
  - 3) Ambiguità → Requisiti con significati multipli
  - 4) Sinonimi e Omofoni → termini diversi con medesi mo significato o termini uguali con significati diversi
  - 5) Presenza di dettagli tecnici
  - 6) Ridondanza: può essere ridondanza, ma tra sezioni diverse

Come si cercano:

- Lettura sequenziale: leggere il documento dall'inizio alla fine
- lettura strutturata:
  - Partendo dal lessicario vado a leggere tutti i requisiti che parlano di quel termine
    - Ricerca di Generalizzazione, cancellazione e Diferenziazione (Chomsky)

tecniche del lessicario: Utilizzando i puntatori ai req. che nominano un certo termine posso facilitare la ricerca di:  
Inconsistente, Sinonimi, Omofoni, Ridondante

Tecniche di Chomsky

Generalizzazione: Processo attraverso cui le persone partendo da un'esperienza specifica, le decontextualizzano traeendo un significato universale

Quantificatori universali: Sempre, tutto, mai, nessuno, chiunque...

Operatori modali: Devo, posso, voglio.

Bisogna cercare questi termini per essere sicuri che la cosa descritta sia sempre vera

**Cancellazioni:** Processo di selezione dell'esperienza → Bisogna inserire i soggetti



chi? Quando?

**Deformazione:** Percezione distorta della realtà



davvero?

**a) Negoziazione:** Modifica alla lista dei requisiti

→ Divisione in clami:

- 1) Must have → Imminutibili per il cliente
- 2) Should have → Non necessari, ma utili
- 3) Could have → Relativamente utili, da realizzare se c'è tempo
- 4) Want to Have → Contattabili per successive versioni

**5) Gestione dei requisiti:**

1) Ogni Requisito ha un identificatore unico

2) Attributi dei requisiti : - Stato: Proposto, approvato, rifiutato, incorporato

3) Tracciabilità

- Vantaggi: Importanza

- Spese in giorni/uomo

- Rischio: Valutazione delle fattibilità tecnica

- Stabilità

- Versione: Per lo sviluppo incrementale

**Modello:** Astrazione del sistema usata per specificare struttura e comportamento

↳ La rappresentazione del sistema è ridotta al minimo per minimizzare la complessità di gestione

**Proprietà:**

- È composto da un insieme di viste (Punti di vista diversi) che insieme danno una visione completa del modello.
- Contiene la conoscenza sul "cosa" e sul "come" di un sistema
- Viene utilizzato come strumento di comunicazione, discussione e per la documentazione

**Uso di un modello:**

- Abbazzone (sketch) : Non completo, rende visibili soluzioni alternative
- Progetto dettagliato (blueprint) : Permette di creare un sistema senza scelte di progettazione
- Eseguibile (UML come linguaggio di prog.) : Molto dettagliato

**Descrizione di un modello:**

- Linguaggio comune, modello semi formale
- Descrive : Struttura, Bolla di progetto e dominio

**UML (Unified modeling language)** : Linguaggio di modellazione unificato

- Caratteristiche:**
- Unificazione a livello di linguaggio
  - Modellazione di ogni fase di sviluppo
  - Indipendente dal linguaggio di sviluppo e di ciclo di vita

**Obiettivi:**

- 1) Visualizzazione
- 2) Specifica e Documentazione
- 3) Realizzazione

**UML analizza due aspetti fondamentali:**

- Struttura statica: Oggetti necessari e relazioni tra questi
- Struttura dinamica: Come gli oggetti collaborano per raggiungere uno scopo

**Modelli in UML:** Un modello in UML è descritto da un insieme di diagrammi (Viste)

**Diagramma:** È composto da entità e relazioni fra esse

↓  
Prono essere  
a livello di classificati:  
o istante a seconda  
del tipo di entità

classificatori  
( classi, attori,  
Casi d'uso,  
ecc... )

**Diagrammi dei casi d'uso:** Viene descritto il sistema

↓  
Modello sia statico che dinamico:  
statico = Diagramma dei casi d'uso  
dinamico = Narrazioni associate ai casi d'uso

**Casi d'uso:** Funzionalità offerta dal sistema a uno o più **attori** (è un insieme di scenari)

**Attori:** Entità esterna al sistema che interagisce con esso in un determinato ruolo

**Scenario:** Sequenza di interazioni tra sistema e attori (usato nel diagramma dinamico)

**Relazioni:** Relazioni significative tra gli attori e casi d'uso

## Sintassi casi d'uso

- Attore: omino con nome (Maiuscolo, volendo UpperCamelCase) (è una classe)
- Associazione: senza nome
- Caso d'uso: ovale con nome (Maiuscolo, volendo UpperCamelCase) (verb)
- Sistema: rettangolo con ~~Nome~~ nome, contiene i casi d'uso
- L'associazione attori—caso d'uso è molti a molti.
- Casi d'uso senza attori solo se inclusi (included)

## Semantica casi d'uso

- Un attore è un utente o un altro sistema, in un particolare ruolo
- Un caso d'uso è un compito (task) che gli attori eseguono con l'ausilio del sistema
- L'associazione attori—caso rappresenta un'interazione (sequenza di messaggi).
- Un caso d'uso è iniziato SOLO da un attore (principale: la narrativa definisce che è l'attore principale)
- Eventualmente Tempo

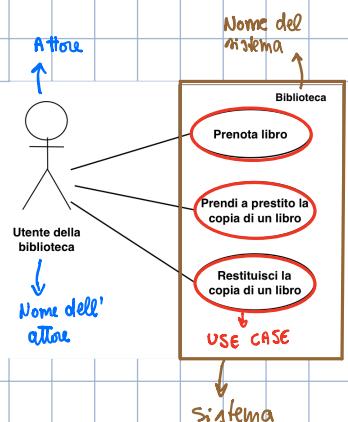
## Processo di modellazione dei casi d'uso

- 1) Individuare confine del sistema
- 2) Individuare gli attori
- 3) Individuare i casi d'uso
- 4) Individuare le relazioni attore - caso d'uso
- 5) Descrizione narrativa per ogni caso d'uso (descrivere il modello dinamico)  
(Documento)

**Nome:**  
**Breve descrizione:**  
**Attori primari:**  
**Attori secondari:**  
**Precondizioni:**  
**Sequenza degli eventi principale:**  
**Postcondizioni:**  
**Sequenze alternative degli eventi:**

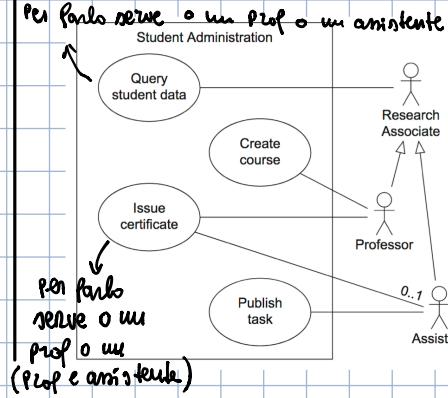
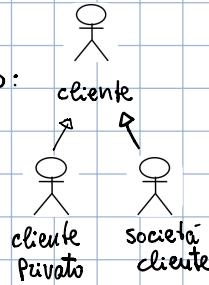
→ Nome caso d'uso  
→ Descrizione  
→ Attore che svolge l'azione  
→ Attore che svolge un'azione sul sistema dovuta al caso d'uso  
→ Condizioni che devono essere rispettate  
→ Sequenza delle azioni che svolge → Posso avere:  
→ Condizioni dopo aver svolto lo use-case  
→ Sequenza di azioni alternative che possono succedere

- (es: Firmare)
1. Fare a
  2. Se (espressione booleana)
    1. Fare b
    3. Altrimenti
      1. Fare c [OPZIONALE]
    4. Per (espressione di iterazione)
      - 4.1 Fare d
    5. While (espressione booleana)
      - 5.1 Fare e
      - 5.2 Fare f



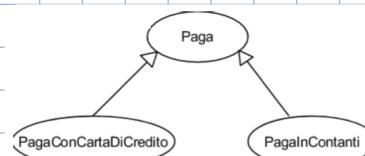
## Generalizzazione:

**Entità:** Generalizza le entità specifiche  $\Rightarrow$  Esempio:

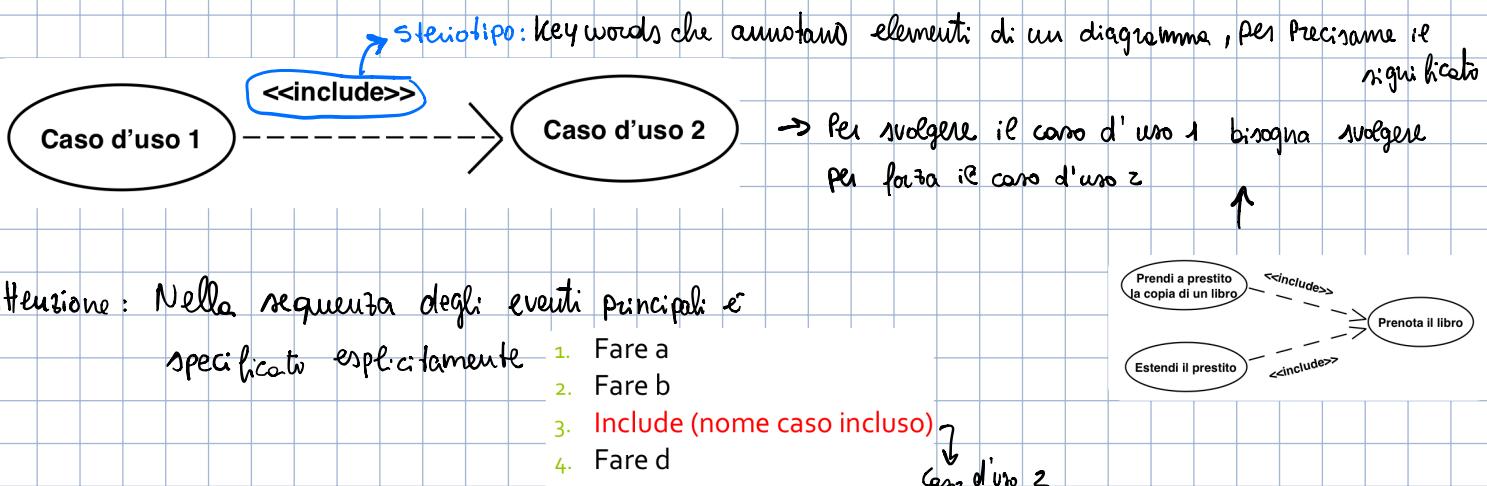


**Caso d'uso:** Generalizza i casi d'uso specifici  $\Rightarrow$  Esempio:

**Attenzione:** Nella Generalizzazione deve essere rispettato il principio di **sostituzione** di Liskov



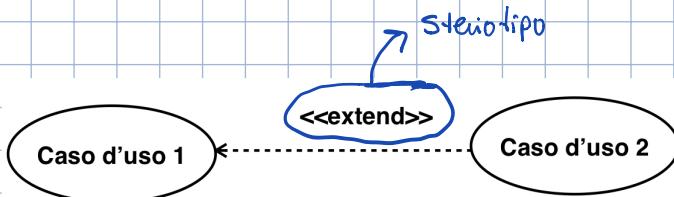
**Inclusione:** Indica che un caso d'uso include l'interazione di un altro caso d'uso



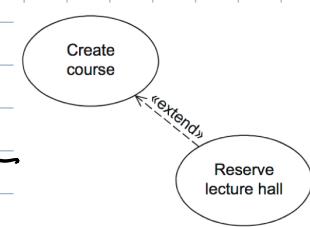
Gli casi d'uso includono puo' essere:

- **Instantiabile:** Avviato da un attore (Avere un attore)
- Non instantiabile: Eseguito solo quando incluso (Non avere un attore)

**Ereditazione:** Quando un caso d'uso ne include un altro ma in modo non obbligatorio



Esempio:



**Attenzione:** Il verso della freccia e' opposto all' include

# Classi e oggetti

**Classe:** Descrive un insieme di oggetti con caratteristiche simili (stesso tipo)

↳ classificatore

Non sono intese come insieme di oggetti ma come la classificazione di un insieme di oggetti. Es: classe Supermercato → Descrive le caratteristiche dei supermercati

**Oggetto:** Entità caratterizzata da: 1) Una identità

↳ istanza

- 2) Uno stato
- 3) Un comportamento

## CLASSE

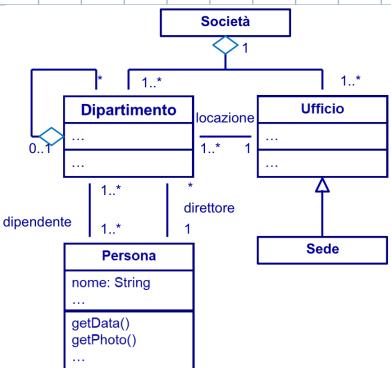
Course
name: String
semester: SemesterType
hours: float

## OGGETTO

helenLewis
firstName = "Helen"
lastName = "Lewis"
dob = 04-02-1980
matNo = "9824321"

**Diagramma delle classi** → Descrivono: 1) Il tipo degli oggetti che fanno parte del SW o del suo dominio

Esempio:



2) Relazioni statiche tra le classi

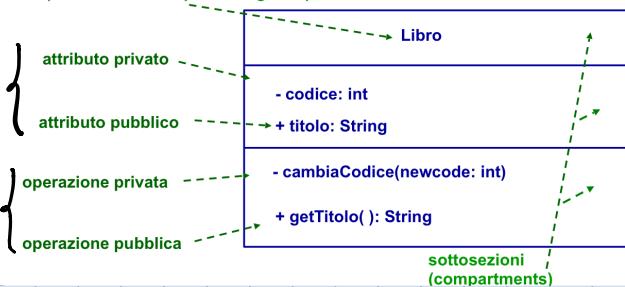
Attenzione: Gli elementi e le relazioni non cambiano nel tempo

## Struttura delle classi:

Nome (maiuscolo e sempre al singolare)

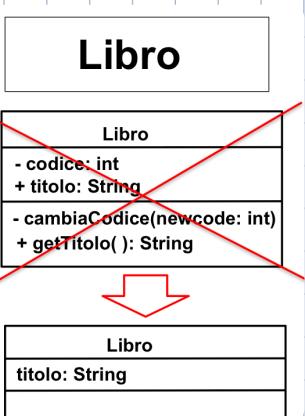
Definiscono lo stato dell' oggetto

Definiscono il comportamento dell' oggetto



Potrei omettere gli attributi e operazioni

ad esempio quando uso il diag. delle classi per descrivere il dominio



## Sintassi attributi:

array di valori

↑

vincoli sui valori che l'attributo può assumere

visibilità **nome**: tipo [molteplicità] = valoreIniziale {proprietà}

**obbligatorio solo il nome**

chiamato di default nel libro, ma si intende iniziale

Esempi :

n: char carattere, tipo predefinito  
 n: String stringa, tipo predefinito  
 g: Gra con tipo Gra definito nel modello  
 n: Integer =1 {>= 0} numero intero non negativo, inizialmente = 1  
 p : Integer [2] {>0} punto del quadrante positivo  
 s: Integer[10] {>3, <33, unique} insieme (unique → senza ripetizioni) di 10 numeri compresi tra 3 e 33  
 col: Integer [3] {>=0, <=255, ordered} lista (ordered → la posizione è significativa) di 3 numeri, compresi tra 0 e 255  
 nome: String [1..2] {ordered, unique} si deve avere un nome, optionalmente si può avere un secondo nome, ma diverso dal primo

Visibilità degli attributi :

+ public: accessibile ad ogni elemento che può vedere e usare la classe

# protected: accessibile ad ogni elemento discendente

- private: solo le operazioni della classe possono vedere e usare l'elemento in questione

~ package: accessibile solo agli elementi dichiarati nello stesso package

## Sintassi delle operazioni:

visibilità **nome (listaParametri)** : tipoRitorno

**obbligatorio solo nome()**

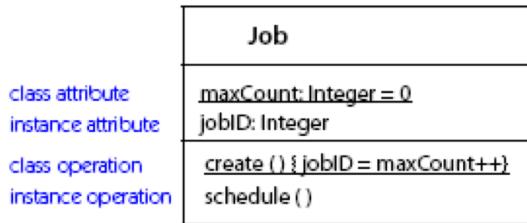
Esempi :

+ sum (a: Integer; b: Integer) : Integer  
metodo pubblico che, dati due interi restituisce un intero

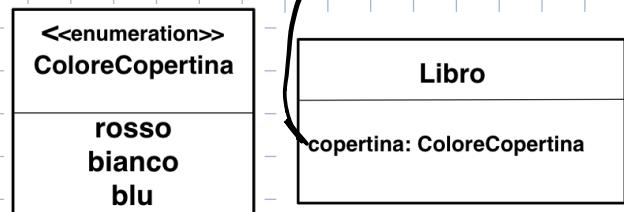
+ sum (a: Integer; b: Integer = 10) : Integer  
come sopra, con 10 valore di default del secondo parametro

- gra () : Gra  
metodo privato che restituisce un oggetto di tipo Gra

Attenzione : Attributi e operazioni statiche sono sottolineati ( Ambito di classe )



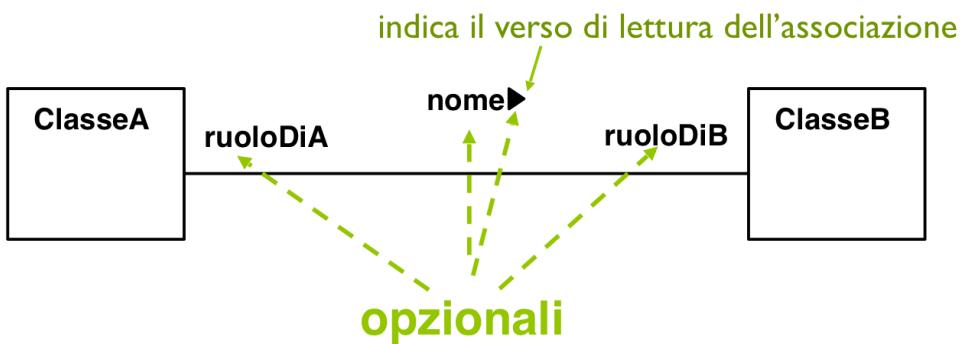
Enumerazioni: Sono utilizzate per specificare un insieme di valori prefissati



più ammene solo i valori dell' enum.

**Relazioni:** Rappresenta un legame tra due o più oggetti, normalmente di classi diverse

## Associazione tra classi



Nome / ruoli: minuscoli

Ejemplo:

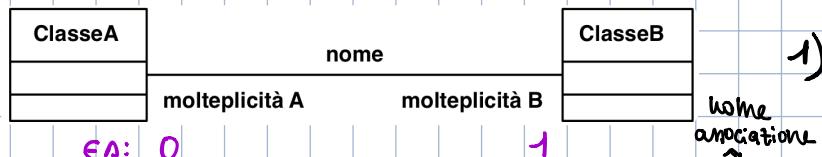
Nome associazione : Un verbo

Muerte: Un sustitutivo

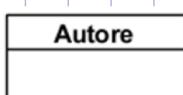
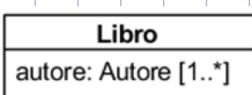
Nelle associazioni abbiamo vincoli di moltiplicità

Numero di oggetti coinvolti nell' associazione

in un dato istante:



Una oggetto della classe B puo' essere in relazione con  $\emptyset$  oggetti della classe A, ma un oggetto delle classe A e' sicuramente in relazione con uno della classe B.



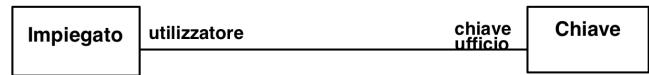
## Associazioni riflessive:

In questo caso è fondamentale indicare il ruolo

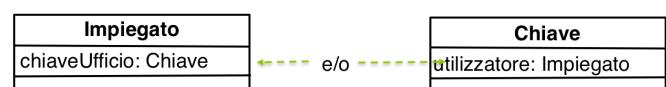
Si mette almeno uno  
tra nome e ruoli.  
Non entrambi.



- Se i è un Impiegato, e c una Chiave, c è stata consegnata ad i

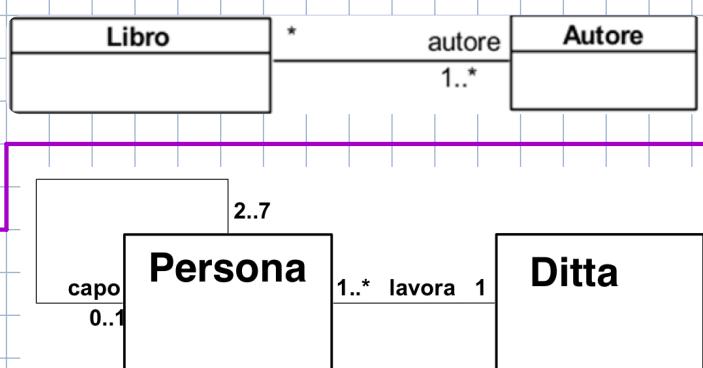


- Si esplicitano i ruoli degli oggetti nella relazione
    - c è la chiave dell'ufficio di i
    - i ne è l'utilizzatore
  - Quando si trasforma il modello in codice:



Tipi di molte plicata: Es: Intervallo 2...4

- 1)  $n \dots n$  dove  $n$  può essere un qualiasi  
uno intero o  $\emptyset \Rightarrow$  Uguale a scrivere " $n$ "
  - 2)  $0 \dots *$   $\Rightarrow$  da 0 a molti  $\Rightarrow$  Uguale a scrivere " $*$ "
  - 3)  $-1 \dots *$   $\circ$   $0 \dots 1$   $\circ$   $1 \dots n$



**Aggregazione e composizione:** Sono tipi particolari di associazioni

↳ Specificano che un oggetto di una classe è una parte di un oggetto di un'altra classe. Cioè un oggetto della classe A è composto da oggetti della classe B

**Aggregazione:** Relazione tra oggetti poco forte es: es. calcolatore con le periferiche

(può essere composta dall'oggetto della classe B)

B



**Sintassi dell' aggregazione:**

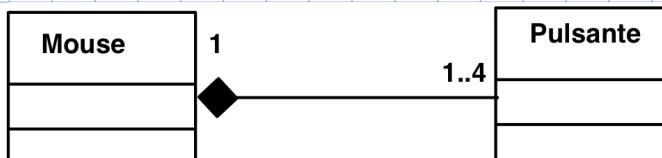
L'aggregazione è una relazione del tipo tutto-parte

- La stampante nel tempo può essere collegata a calcolatori diversi
- La stampante esiste anche senza calcolatore

**Composizione:** Relazione tra oggetti forte

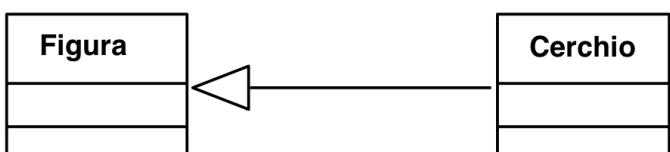
(deve essere composta dall'oggetto della classe B)

es: es. albero e le sue foglie

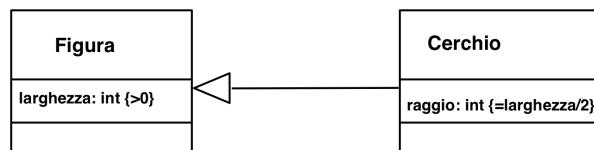


**Sintassi della composizione:**

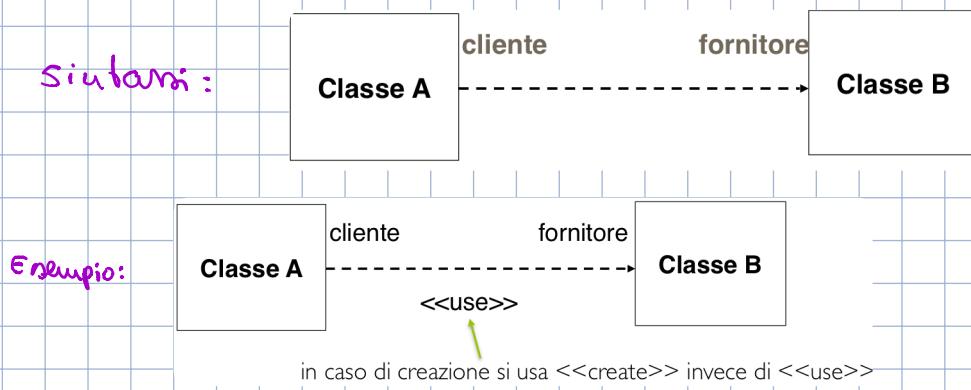
**Generalizzazione:**



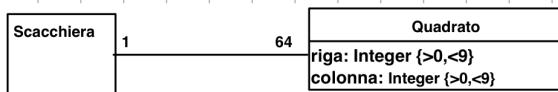
- Relazione tra un elemento generico e uno più specializzato
- L'elemento più specializzato è completamente consistente con quello più generico ma contiene più informazione
- Vale il principio di sostituzione della Liskov: l'elemento specializzato può essere usato al posto dell'elemento generico
- "è un tipo di"
- Le sottoclassi ereditano tutte le caratteristiche della superasse:
  - attributi, operazioni, relazioni e vincoli
- Le sottoclassi possono aggiungere caratteristiche e ridefinire le operazioni



**Dipendenza**: Una modifica alle classe A modifica anche la classe B

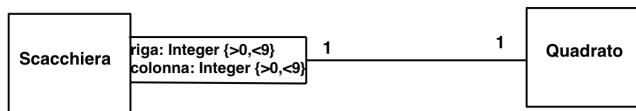


## Associazioni qualificate

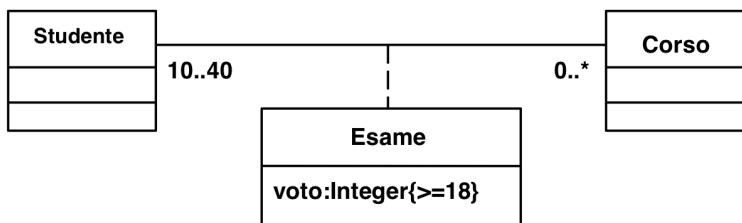


Le associazioni qualificate riducono un'associazione \* a molti ad un'associazione \* a 1 specificando un unico oggetto scelto nell'insieme destinazione (specificandone una chiave).

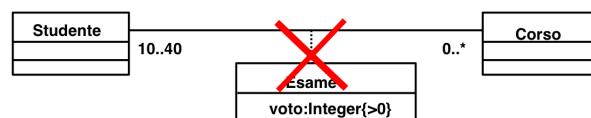
La combinazione riga,colonna specifica un unico destinatario



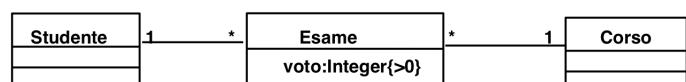
## classi associazione



Per ogni coppia di oggetti collegati tra loro può esistere un unico oggetto della classe associazione



Se vogliamo tenere traccia dei voti negativi non si possono usare le classi associazione



## Corso sono le classi

- Corrispondono a concetti concreti del dominio : - Es: Concetti descritti nel glossario
- Abstrazione di uno specifico elemento del dominio - Ciascuna classe di analisi sarà raffinata in una o più classi di progettazione
- Numero ridotto di responsabilità (functions)
- Operazioni e attributi solo quando veramente utili

## Identificazioni delle classi

- Approccio data driven : Si identificano tutti i dati del sistema e si dividono in classi
- Approccio responsibility driven: Si identificano le responsabilità e si dividono in classi

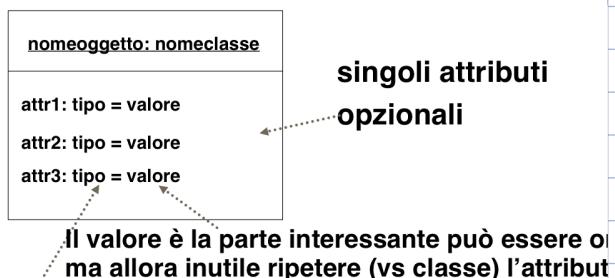
### Analisi nome - verbo :

- Sostantivi → classi o attributi
- Verbi → responsabilità o operazioni
- Passi:
  - Individuazione delle classi
  - Assegnazione di attributi e responsabilità alle classi
  - Individuazione di relazioni tra le classi

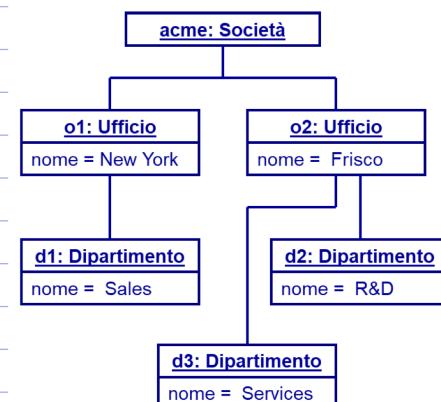
### Problemi ricorrenti :

- Tagliare le classi inutili
- Trattare i casi di sinonimia
- Individuare le classi nascoste cioè le classi implicite del dominio del problema che possono anche non essere mai menzionate esplicitamente
- In un sistema di prenotazione di una compagnia di viaggi si potrebbe parlare di prenotazione, richiesta, ma tralasciare il termine ordine

## Diagramma degli oggetti



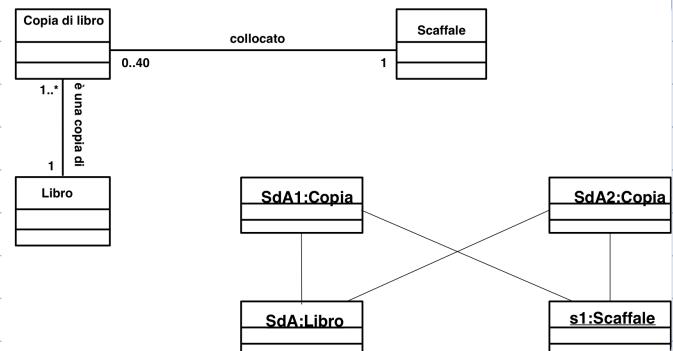
### Esempio:



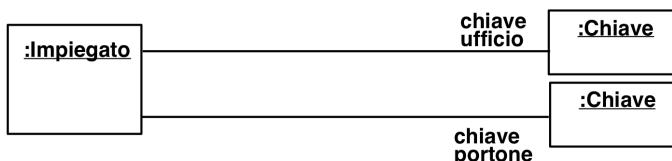
## Collegamenti

- Un collegamento è una istanza di una associazione
- Collega due (o più) oggetti
- Non ha un nome
- Se utile si possono indicare i ruoli
- Non ha molteplicità, è sempre 1 a 1
  - la molteplicità di una associazione dice quanti collegamenti ci saranno a livello di istanza

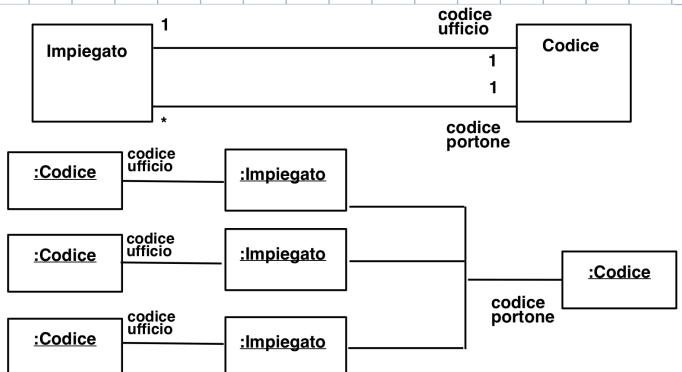
## Classi e oggetti



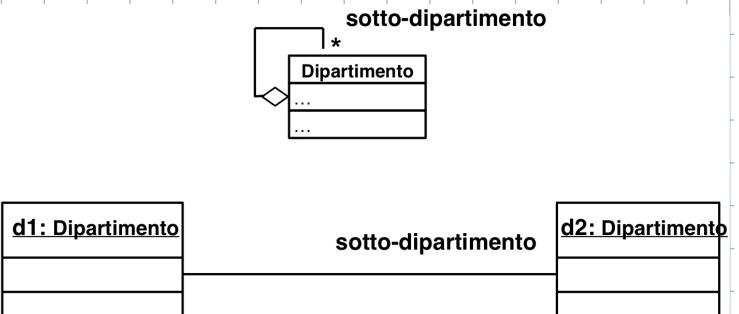
### Esempio: Ruoli:



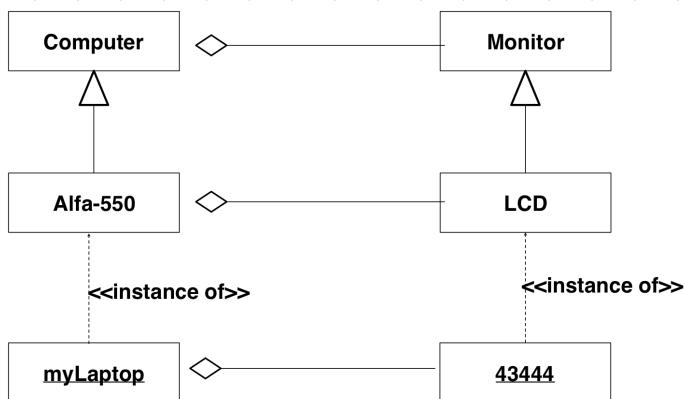
Moltiplicità esempio:



Ruoli e ogn. esempio:



Esempio Generalizzazione, istanza e aggregazione



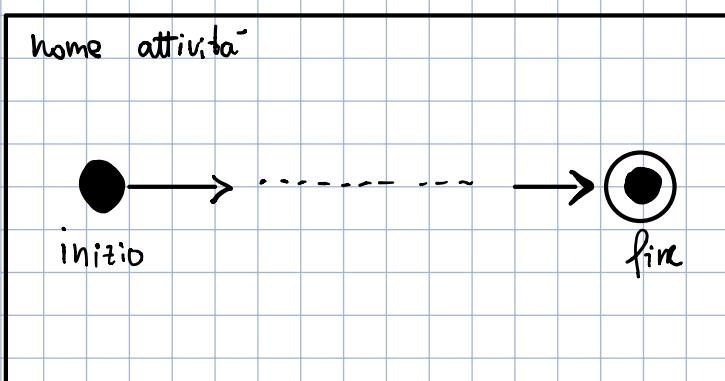
Diagrammi di attività → Modellano il flusso di lavoro:

- Di un compito o algoritmo
- Di un processo / attività

Un'attività descrive la coordinazione di un insieme di azioni.



Grafo diretto i cui nodi rappresentano le componenti dell'attività come le attori e gli archi rappresentano il flusso di controllo (possibili path eseguibili per l'attività)



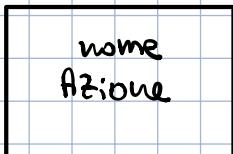
nodo di fine flusso



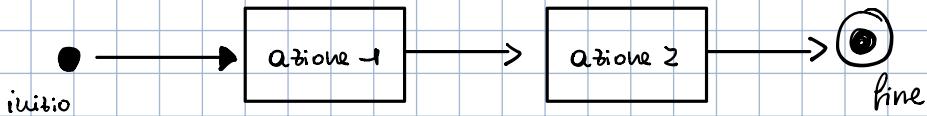
nodo di fine attività



## Azioni

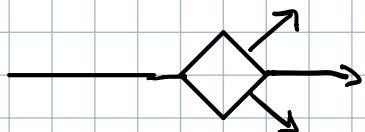


Indicano un'azione che deve essere svolta nel svolgere quella attività.



Quando un'azione termina il proprio lavoro scatta una transizione automatica in uscita dall'azione che porta all'azione successiva (la remanica è descritta dal token game: l'azione può essere eseguita quando riceve il token)

## Elementi principali



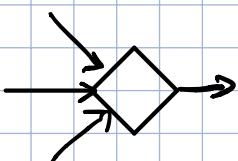
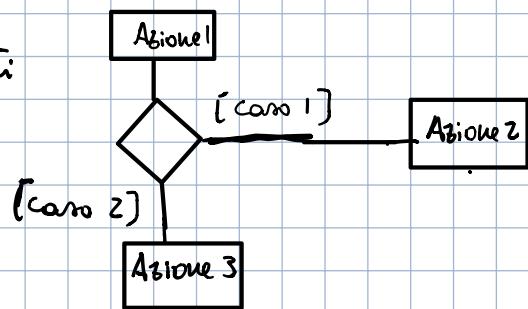
### Nodo di decisione

nelle guardie  
posso usare le  
parole altrimenti:

Il token segue uno e  
uno solo dei percorsi multipli:

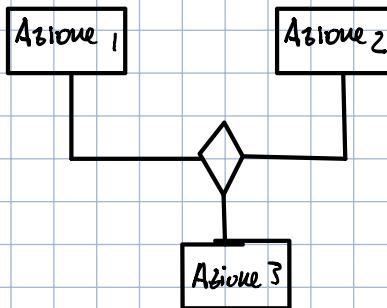
(le guardie devono coprire tutti i casi  
possibili! Giacché il token deve prendere  
per forza un percorso)

⇒ Permette di decidere tra 2 casi  
o più, quale dei due flussi voglie

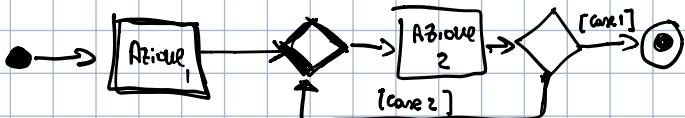


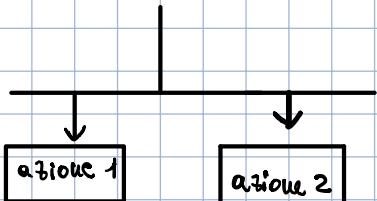
### nodo di giunzione (fusione)

⇒ Permette di riunire flussi diversi

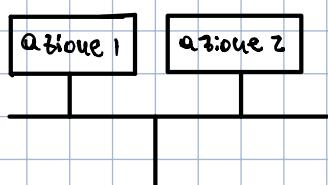


## loops





Fork (biforcazione) → Permette di dividere un flusso in più flussi paralleli. Quindi da un token, ne ottengo tanti quanti sono le frecce uscenti.



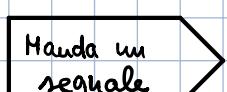
Join → Da più flussi, li riportano in uno. Quindi da più token ne ottengo uno solo.

Attenzione: Se metto un nodo di giunzione (flusso) per unire più flussi diversi, l'azione dopo la fusione viene eseguita più volte, tante quante sono i token.

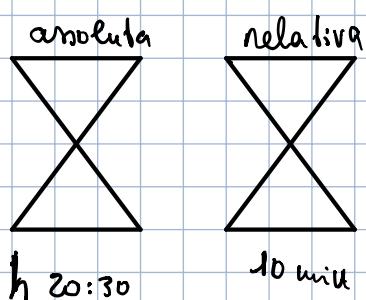
## Segnali ed eventi



Accettazione di un evento esterno



Invio di un segnale → è attivante e non blocca l'attività



Accettazione di un evento temporale

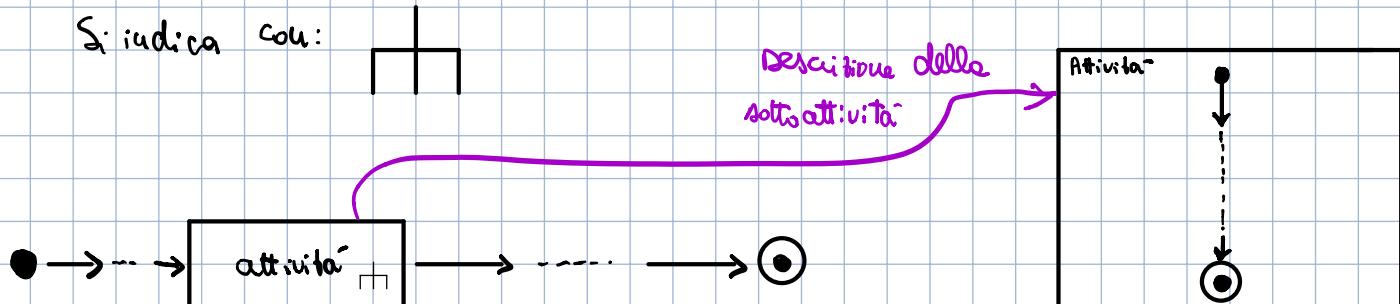
Attenzione: Si usa un'azione quando è effettuata dal classificatore / viene da classificatori di cui si sta studiando il comportamento.

Segnali ed eventi quando si comunica con un entità esterna

## Sottoattività

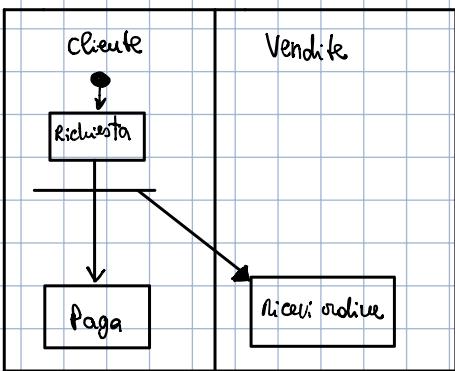
Un'azione può chiamare un'attività secondaria che deve essere anch'essa descritta.

Si indica con:



**Partitività:** Posso dividere le azioni in gruppi, in modo da assegnare le responsabilità delle azioni.

Ese:



**Macchine a stati:** Descrivono il comportamento dinamico delle istanze di un classificatore.



Viene rappresentata con un grafo di stati e transizioni, associata a un classificatore.

**Stato:** Un insieme di valori di un oggetto

- Ha un nome unico
- Può essere composto

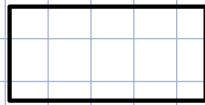
Sintassi:



Stato iniziale

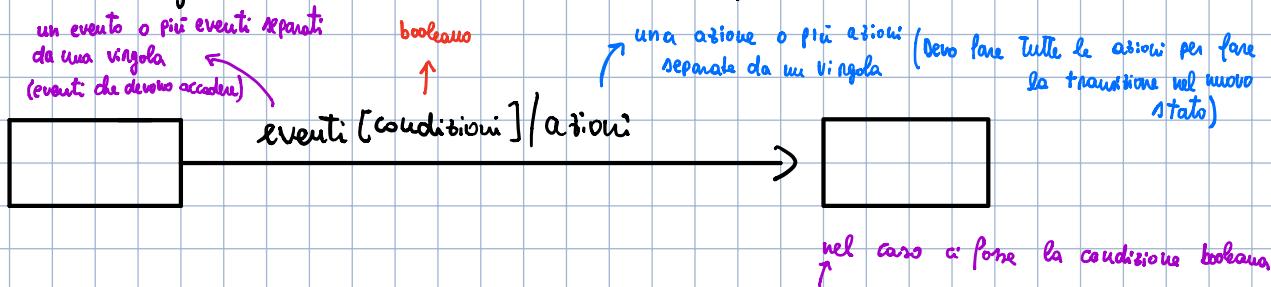


Stato finale



Stato

**Transizione**: Collega tra di loro 2 stati con una Preziosa



La transizione viene presa **soltamente** se la condizione divenuta vera

Esempio:



nel diag. delle classi!



OSSERVAZIONE: Gli eventi (anche se non tutti) corrispondono alle operazioni della classe.

**Evento**: Occorrenza di un fenomeno collocato nel tempo e nello spazio.

- Occorre istantaneamente
- Viene modellato come evento qualcosa che ha delle conseguenze
- Gli eventi che arrivano in uno stato per cui non c'è prevista alcuna transizione vengono ignorati
- Un evento può fare da trigger a più transizioni, le quali escono dallo stesso stato, ma viene scelta una non-deterministicamente.

**Tipi di evento**:

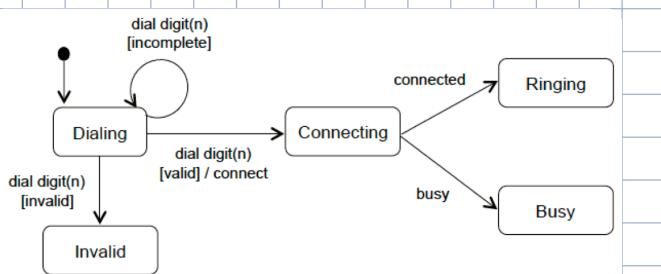
- 1) Operazione / segnale  $op(a:T)$  → La transizione è abilitata quando l'oggetto riceve una chiamata di metodo o un segnale con parametri ( $a$ ) e tipo ( $T$ )

può prendere dei parametri, non obblig.

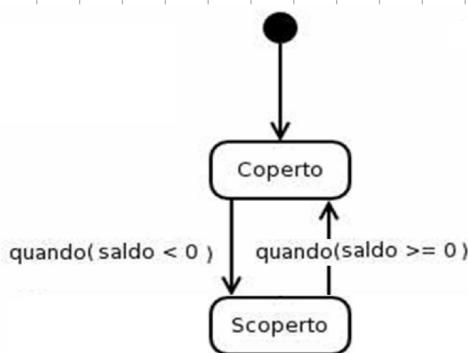
- 2) Evento di variazione:  $quando(exp)$  → La transizione è abilitata appena l'espressione diventa vera.

3) Evento temporale:  $dopo(time)$  → La transizione è abilitata dopo che l'oggetto è stato ferito "time" in quello stato.

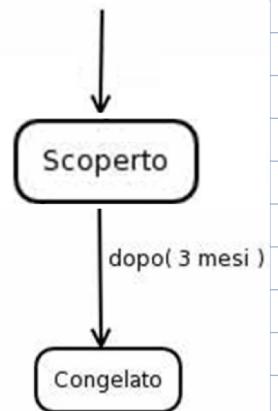
## Evento Operazione / Segnale



## Evento di variazione



## Evento temporale

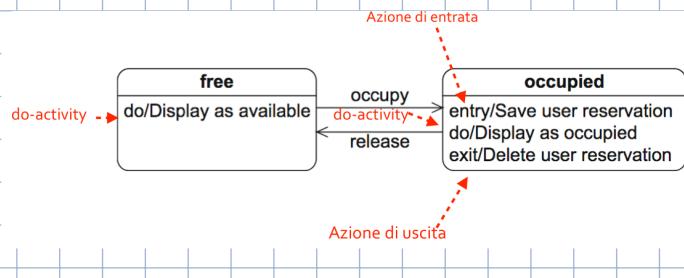


**Transizione interna:** Risposta a un evento che causa l'esecuzione di azioni

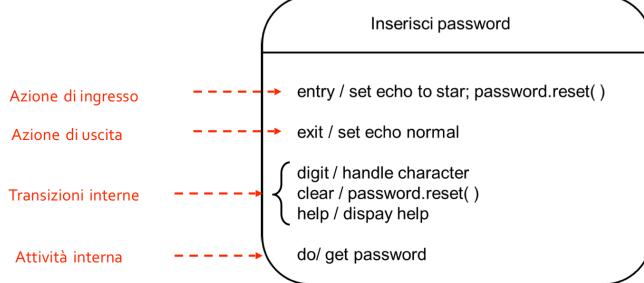
- ↳ 1) Azione di entrata: Eseguita all'ingresso in uno stato
- 2) Azione di uscita: Eseguita all'uscita di uno stato
- 3) Transizione interna: Risposta a un evento

**Attività interna ( Do-activity):** Eseguita in modo continuo mentre l'oggetto si trova in quello stato. Al contrario di tutte le altre azioni che sono atomiche, queste consumano tempo e possono essere interrotte (uscita dallo stato).

### Esempio:

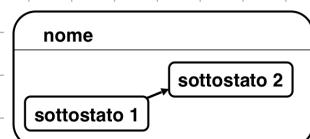


### Siutarmi

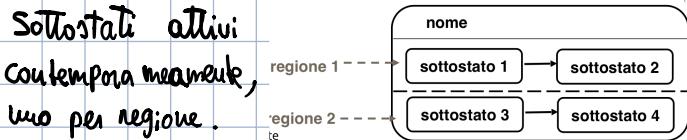


### Stati Composti:

- 1) Composito sequenziale: Uno stato attivo in ogni istante.

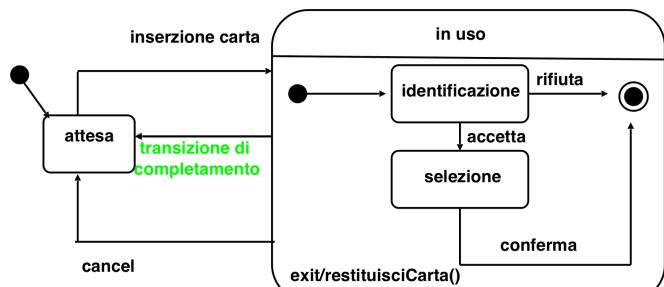


- 2) Composito parallelo: Sottostati attivi contemporaneamente, uno per regione.



Esempio:

### Composite seq. esempio.

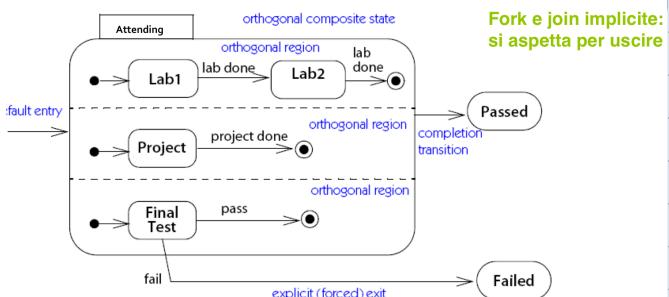


- Ogni transizione che arriva sul bordo prosegue nello stato iniziale
- Dallo stato finale (dopo le exit) si prosegue nella transizione di completamento
- Ogni transizione (non di completamento) che parte dal bordo si intende possibile da un bisogno di un evento per essere attivata!

transizione che non ha bisogno di un evento per essere attivata!

↳ cioè sta da ident. che selezione !!

### Composite parallelo esempio

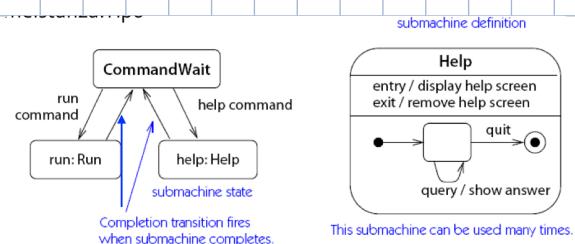


Fork e join implicite: si aspetta per uscire

- Ogni transizione che arriva sul bordo prosegue in tutti gli stati iniziali
- Una volta raggiunti tutti gli stati finali si prosegue nella transizione di completamento
- Possono esserci transizioni che bucano il bordo e si intendono possibili dal solo stato interno a cui sono collegate

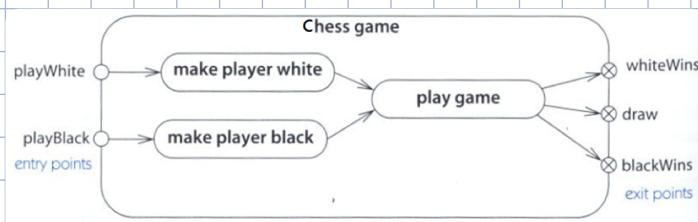
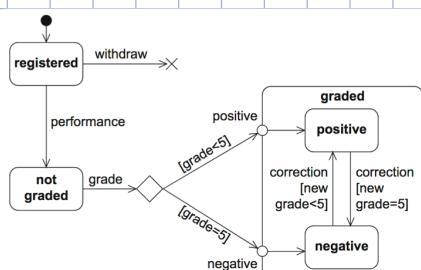
**Sotto macchine:** Si usa quando si vuole descrivere uno stato composito in un diagramma a parte, per leggibilità o per definirlo una volta per tutte e riutilizzarlo in più contesti.

Esempio:



Una sotto macchina può definire entry e exit points che servono per collegare le transizioni della macchina principale

Esempio:



Riepilogo sulle transizioni di completamento:

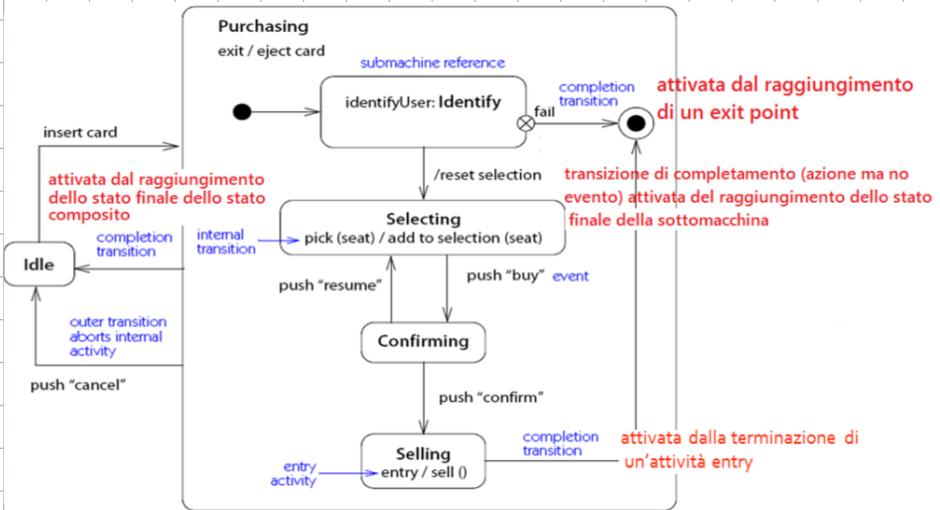
+ transizioni senza evento

■ Senza evento, scattano al raggiungimento

- Della terminazione di un'attività composita, i.e. al raggiungimento
  - Dello stato finale in un stato composito non-ortogonale
  - Degli stati finali di tutte le regioni ortogonali di un stato composito
  - Di un exit point
- Alla terminazione di entry e/o di do activity (la exit activity viene eseguita quando scatta la transizione di completamento)
- Di uno pseudo-stato giunzione (lo vedremo in un attimo)

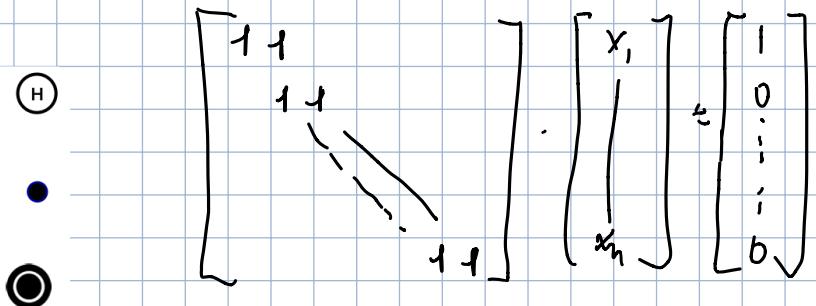
■ Hanno priorità sugli eventi normali

## Esempio transizioni di completamento:



## Altri tipi di stato

Giunzione	●	Storia	(H)
Decisione	◇	Iniziale	●
Fork, join		Finale	○



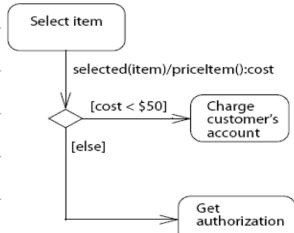
$$\begin{aligned}x_1 + x_2 &= 1 & 1 & 0 \\x_2 + x_3 &= 0 & 0 & 1 \\&\vdots && \\x_{n-2} + x_{n-1} &= 0 && \end{aligned}$$

$$x_{n-1} + x_n = 0$$

## 1) Decisione:

↓  
esco se è vero  
l'evento

e poi valuto la decisione



- Condizioni valutate dinamicamente
- Come per la choice dei diagrammi di attività:
- La disgiunzione delle guardie deve valere "true"
- È ammesso il non-determinismo

## 2) Giunzione: - Statica

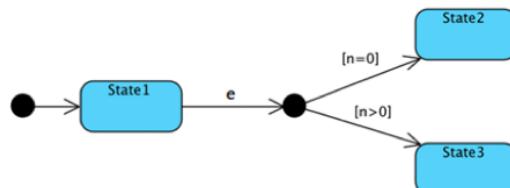
↓  
esco solo se è vero

l'evento e anche  
una delle scelte risulta vera  
(valuto la dec. prima di uscire)

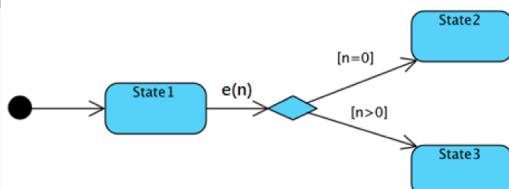
↓  
vengono valutate dopo

l'evento per sapere

se uscire o no

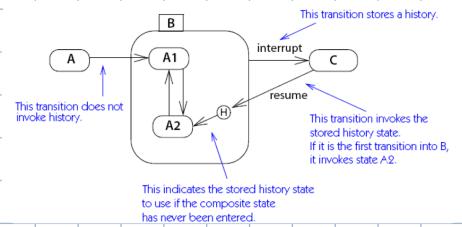


Le guardie sono valutate prima di uscire da State1. Se  $n < 0$ , l'evento è ignorato e nessuna transizione viene presa.



Le guardie sono valutate dopo  $e(n)$ . In questo esempio occorre avere garanzia che  $n$  sia maggiore o uguale a zero.

3) History State: servono in presenza di una transizione esterna da uno stato composto, per ricordare l'ultimo stato attivo in modo da poter riutnare esattamente in quello stato



## Nomi degli stati e delle azioni

### Nomi degli stati:

- Aggettivi: Attivo
- Partici passati: accesa, spenta, finita
- Gerundi: Dialing, connecting
- Altri: InAttesa

### Nomi delle azioni:

- Verbi all'indicativo, imperativo, infinito: crea, inviare
- Sostantivi che indicano un'azione: funzione DB

Scelgere il diagramma più appropriato per descrivere il modello dinamico:

- Se il focus è mettere in ordine un insieme di azioni da fare → Attività
- Se il focus è mostrare l'evoluzione di un oggetto in risposta a eventi → Stati

## Riassunto macchine a stati

Table 7-1: Kinds of Events

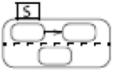
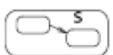
Event Type	Description	Syntax
call event	Receipt of an explicit synchronous call request by an object	op (a:T)
change event	A change in value of a Boolean expression	when (exp)
signal event	Receipt of an explicit, named, asynchronous communication among objects	sname (a:T)
time event	The arrival of an absolute time or the passage of a relative amount of time	after (time)

State Kind	Description	Notation
submachine state	A state that references a state machine definition, which conceptually replaces the submachine state	
entry point	A externally visible pseudostate within a state machine that identifies an internal state as a target	
exit point	A externally visible pseudostate within a state machine that identifies an internal state as a source	

Table 7-2: Kinds of Transitions and Implicit Effects

Transition Kind	Description	Syntax
entry transition	The specification of an <b>entry activity</b> that is executed when a state is entered	entry/ <b>activity</b>
exit transition	The specification of an <b>exit activity</b> that is executed when a state is exited	exit/ <b>activity</b>
external transition	A response to an <b>event</b> that causes a change of <b>state</b> or a self-transition, together with a specified <b>effect</b> . It may also cause the execution of exit and/or entry activities for states that are exited or entered.	e(a:T)[guard]/ <b>activity</b>
internal transition	A response to an event that causes the execution of an effect but does not cause a change of state or execution of exit or entry activities	e(a:T)[guard]/ <b>activity</b>

**Table 7-3: Kinds of States**

<i>State Kind</i>	<i>Description</i>	<i>Notation</i>
simple state	A <b>state</b> with no substructure	
orthogonal state	A state that is divided into two or more regions. One <b>direct substate</b> from each <b>region</b> is concurrently active when the composite state is active.	
nonorthogonal state	A composite state that contains one or more direct substates, exactly one of which is active at one time when the composite state is active	
initial state	A <b>pseudostate</b> that indicates the starting state when the enclosing state is invoked	
final state	A special state whose activation indicates the enclosing state has completed activity	
terminate	A special state whose activation terminates execution of the object owning the state machine	
junction	A pseudostate that chains <b>transition segments</b> into a single <b>run-to-completion transition</b>	
choice	A pseudostate that performs a dynamic branch within a single run-to-completion transition	
history state	A pseudostate whose activation restores the previously active state within a composite state	

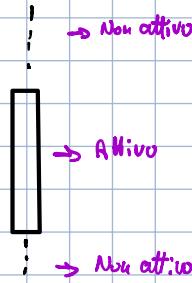
Diagrammi di sequenza → Usati per descrivere lo scambio di messaggi e dati tra oggetti

## Sintaxis :

- 4) Ogni oggetto è rappresentato da un rettangolo

: Uteuke

- 2) la linea di vita dell' oggetto è una linea verticale : - è tralungata quando l' oggetto non è

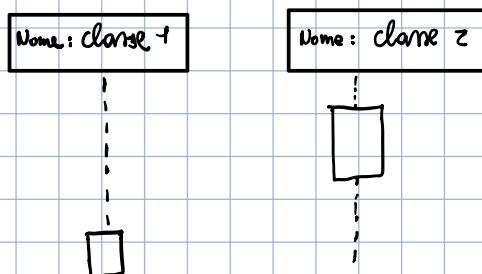


- Continua e doppia quando è attivo

3) le linee che collegano le linee di vita degli oggetti: sono detti "monaggi".

- Sincroni
- - Li sîntoare ambele sincroni
- Asincroni
- Asincroni sau atomici

Essi rappresentano una invocazione di operazione o segnali

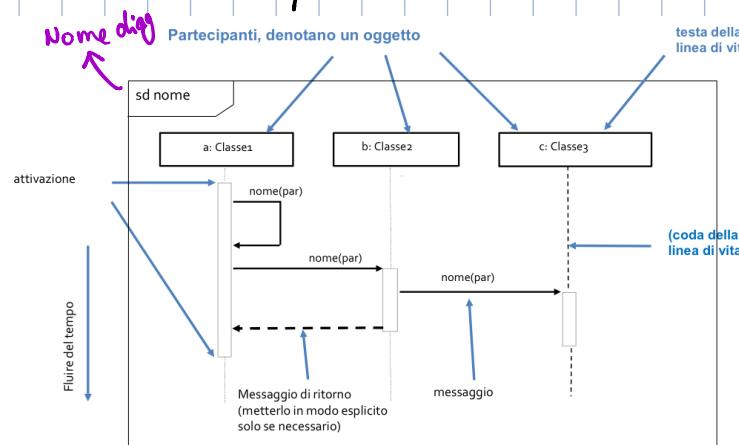


Sintomi che fanno adottare:

attributo = Nome Mex (arg1, ... ) : Valore di ritorno  
  ↓  
  opzionale  
                ↓  
                opzionale  
                ↓  
                opzionale

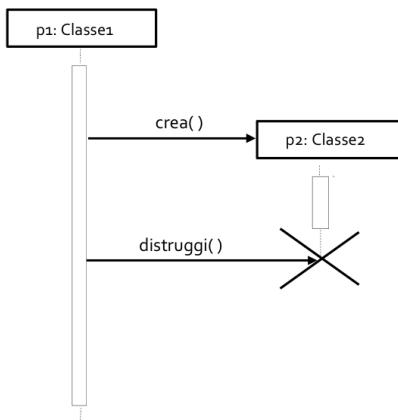
Esempio:

T Pomo avere anche 2 oggetti della stessa classe che comunicano tra loro

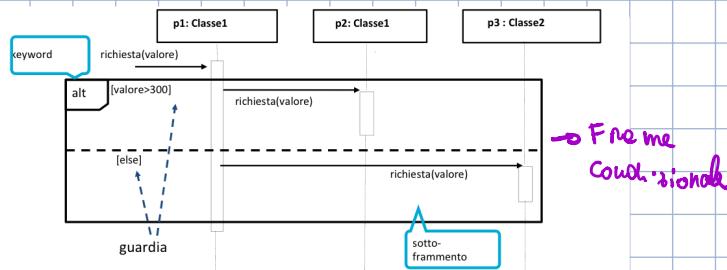


Messaggi scambiati, l'ordine cronologico è dall'alto in basso

Create e distruggere partecipanti → Possiamo creare ogj. dinamicamente / Cancellati



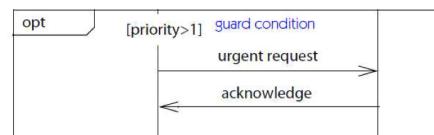
### Frame Condizionale



- senza guardia = [true]
- più guardie vere: scelta non-deterministica
- tutte le guardie false: il frame viene saltato

### Frame opzionale

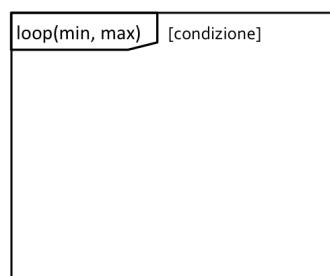
- if then (senza else)
- le interazioni contenute nel frame vengono eseguite solo se la guardia è vera, altrimenti si salta il frame



### Frame Iterativo

Si itera:

- almeno min e non più di max volte
  - (indipendentemente dal valore della condizione)
- tra min e max si valuta la condizione e si esegue il frame solo se questa è vera, altrimenti si esce



- Il frame deve essere eseguito almeno una volta
- Alla seconda (e se non si è già usciti alla terza) iterazione si controlla la guardia
- Dopo 3 iterazioni si esce comunque

## Modellare while, do while, for:

- loop(0,\*) [guardia] (oppure loop [guardia])
  - modella: while(guardia) { ... }

- loop(1,\*) [guardia]
  - modella: do { ... } while(guardia)

- loop(n, n) (oppure loop(n)) (senza guardia)
  - modella: for(i=0; i<n; i++)
  - attenzione, non loop(0,n)!!!

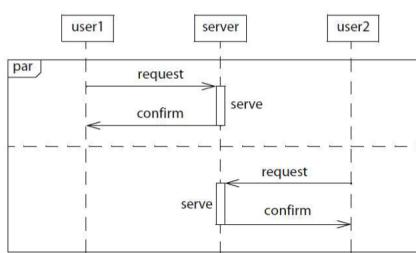
## Frame parallelo

- Le interazioni contenute nei due sotto-frammenti sono eseguite in parallelo

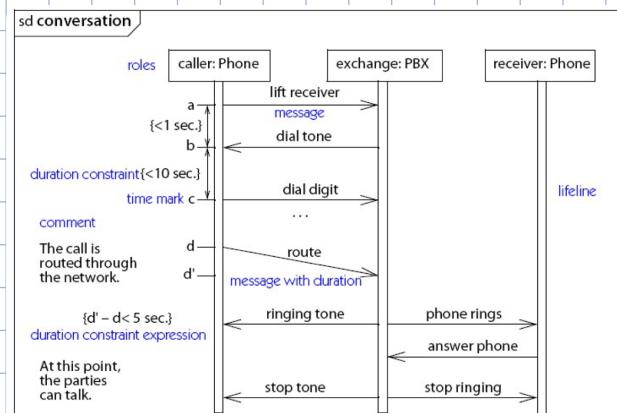
- Semantica a interleaving

### Nell'esempio:

- Le richieste dei due clienti possono arrivare in un ordine qualsiasi



## Vuoli di durata

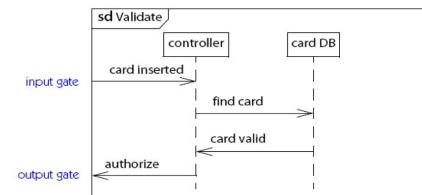


## Gates (cancelli)

- un gate (cancello) è un punto sul bordo del diagramma a cui è collegato un messaggio, in ingresso o in uscita

- il nome del gate è quello del messaggio

- utili quando si riferiscono (ref) altri diagrammi



Includere un diagramma di sequenza esterno in un diagramma di sequenza

### ref

- Include il diagramma di sequenza indicato

## Seconda Parte del Programma

Progettazione: Fase tra specifica e codifica → Si decide come devono essere fatte le cose

↳ Produce l'architettura software

- Definisce la struttura del SW

- Specifica la comunicazione tra componenti

- Considera aspetti: non funzionali

- E' un'astrazione

- E' un artefatto complesso

E' la struttura del sistema costituita dalle parti del sistema, dalle relazioni tra le parti e dalle loro proprietà visibili.

**Vista:** c'è un'astrazione della architettura SW secondo un aspetto del sistema

- 1) **Vista comportamentale:** la struttura come insieme di unità con comportamenti e interazioni, a tempo d'esecuzione
- 2) **Vista strutturale:** la struttura come insieme di unità di realizzazione (codice)
- 3) **Vista logistica:** le relazioni con strutture diverse dal SW nel contesto del sistema

## Vista comportamentale →

### Componenti :



classificatore

- unità concettuali di decomposizione di un sistema a tempo d'esecuzione,
  - Per esempio: processi, oggetti, serventi, depositi di dati, ...
- incapsula un insieme di funzionalità e/o di dati di un sistema
- restringe l'accesso a quell'insieme di funzionalità e/o dati tramite delle interfacce definite
- ha un proprio contesto di esecuzione
- può essere distribuito e installato in modo (possibilmente) indipendente da altri componenti

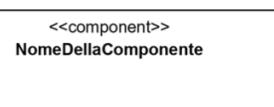
La vista C&C descrive un sistema software come composizione di componenti software

- specifica i componenti con le loro interfacce
- descrive le caratteristiche dei connettori
- descrivere la struttura del sistema in esecuzione
  - flusso dei dati, dinamica, parallelismo, replicazioni, ...

Utile per:

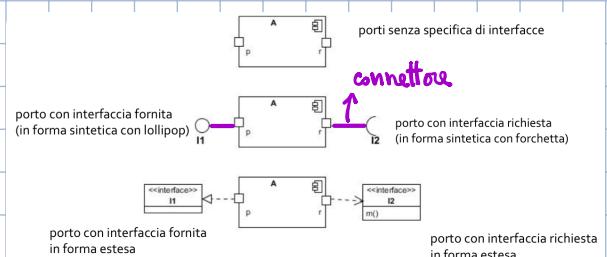
- analisi delle caratteristiche di qualità a tempo d'esecuzione
  - prestazioni, affidabilità, disponibilità, sicurezza, ...

## Rappresentazione



**Porti:** Identificano i punti di interazione di un componente

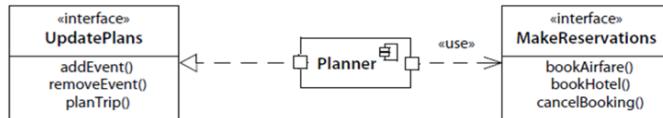
- 1) Un componente può avere più porti, uno per ogni tipo di connessione con altri componenti
- 2) Un porto fornisce o richiede una o più interfacce (omogenee)
- 3) Un porto può avere associata una molteplicità



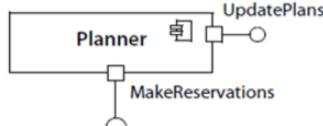
**Sistema Software:** Composizione di componenti software realizzata sulla base delle interfacce e collegati mediante i connettori

Y interface : Possono essere in descrizione estesa / sintetica

Estesa:



Sintetica:

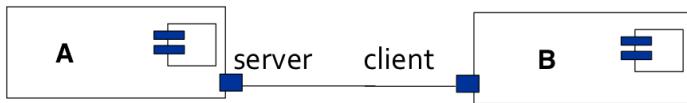


Connettori: linee che collegano i componenti. Sui connettori si può specificare:

1) Protocollo di interazione:

- <<clientServer>>; <<dataAccess>> (specializzazione di clientServer); <<pipe>>; <<peer2peer>> (<<p2p>>); <<publish-subscribe>>

2) Ruoli componenti:



Stili comportamentali: Sono schemi standard caratterizzati da:

- Caratteristiche generali delle componenti in gioco
- Particolari interazioni tra le componenti

1) Pipe & filters:

#### ■ Componenti

- sono di tipo filtro: trasformano uno o più flussi di dati dai porti d'ingresso in uno o più flussi sui porti d'uscita

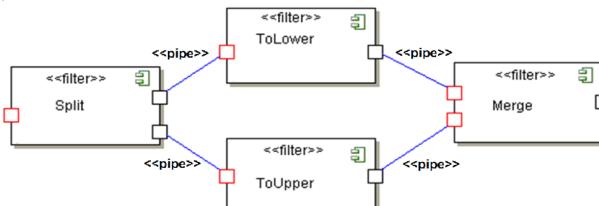
#### ■ Connettori

- sono di tipo condotta (pipe): canale di comunicazione unidirezionale bufferizzato che preserva l'ordine dei dati dal ruolo d'ingresso a quello d'uscita

#### ■ Usi

- pre-elaborazione in sistemi di elaborazione di segnali
- analisi dei flussi dei dati, e.g. dimensioni dei buffer

Esempio:

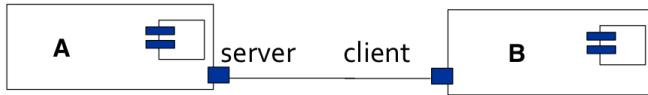


Per esempio «ciao» o «CIAO» o «Ciao» viene trasformato in «claO»

2) Client - Server : Componenti : client e server

↓  
Invia le richieste  
ed attende una risposta

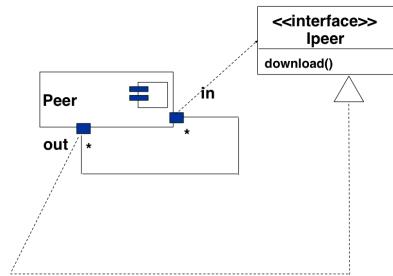
Realizza un servizio



(server)

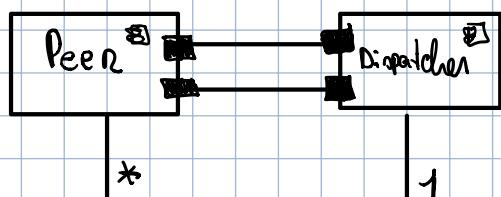
3) Master - Slave : Uguale al client - Server solo che uno slave serve un solo master (client)

4) P2P : Uguale al client - Server solo che tutti i componenti agiscono sia da client che da server.



5) Publish - Subscribe :

- In questo stile, mittenti e destinatari di messaggi dialogano attraverso un tramite, che può essere detto dispatcher o broker. Il mittente di un messaggio (detto publisher) non deve essere consapevole dell'identità dei destinatari (detti subscriber); esso si limita a "pubblicare" (in inglese to publish) il proprio messaggio al dispatcher. I destinatari si rivolgono a loro volta al dispatcher "abbonandosi" (in inglese to subscribe) alla ricezione di messaggi. Il dispatcher quindi inoltra ogni messaggio inviato da un publisher a tutti i subscriber interessati a quel messaggio.
- In genere, il meccanismo di sottoscrizione consente ai subscriber di precisare nel modo più specifico possibile a quali messaggi sono interessati. Per esempio, un subscriber potrebbe "abbonarsi" solo alla ricezione di messaggi da determinati publisher, oppure avendo certe caratteristiche.
- Questo schema implica che ai publisher non sia noto quanti e quali sono i subscriber e viceversa. Questo può contribuire alla scalabilità del sistema.



## 6) Model - View - Controller (MVC)



- L'utente interagisce con la vista
- Il controller riceve le azioni dell'utente e le interpreta
  - Se si fa clic su un pulsante, è compito del controllore capire che cosa significa e come il modello dovrebbe essere manipolato in base a tale azione.
- Il controller chiede al modello di cambiare il suo stato
- Il modello notifica la vista quando il suo stato è cambiato
  - Quando qualcosa cambia nel modello, in risposta a qualche azione (es. fare clic su un pulsante) o per altri motivi (es. è iniziato il brano successivo nella playlist), il modello notifica alla vista che il suo stato è cambiato.
- La vista chiede lo stato al modello
  - Per esempio, se il modello notifica la vista che è iniziata un nuovo brano, la vista richiede il nome del brano al modello e lo mostra.

■ Stile in cui si isola la logica di business dal controllo sull'input e dalla presentazione (vista sui dati), consentendo sviluppo indipendente, test e manutenzione di ciascuno.

### ■ Modello

- È la rappresentazione del dominio dei dati su cui opera l'applicazione. Quando un modello cambia il suo stato, notifica le sue viste associate in modo che si possano aggiornare.

### ■ Vista

- Rende il modello in una forma adatta all'interazione, in genere un elemento dell'interfaccia utente. Ci possono essere più viste per un singolo modello, per scopi diversi.

### ■ Controllore

- Riceve l'input e effettua chiamate agli oggetti del modello.

## 7) Coordinatore di processi :

- Il Coordinatore di processi conosce la sequenza di passi necessari per realizzare un processo.
- Riceve la richiesta, chiama i servers secondo l'ordine prefissato, fornisce una risposta
- Normalmente usato per realizzare processi complessi
- Disaccoppiamento: i server non conoscono il loro ruolo nel processo complessivo né l'ordine dei passi del processo. Ogni server semplicemente definisce un servizio
- Comunicazione flessibile: sincrona o asincrona.

Viste strutturali :



Modulo : Unità software che realizza un insieme coerente di responsabilità (classi, collezione di classi)

A cosa servono

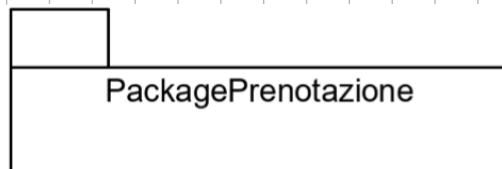
- costruzione
  - la vista può fornire lo schema del codice e directory e file sorgente hanno una struttura corrispondente
- analisi
  - tracciabilità dei requisiti
  - analisi d'impatto per valutare eventuali modifiche
- comunicazione
  - se la vista è gerarchica, offre una presentazione top-down della suddivisione delle responsabilità nel sistema ai novizi

Relazioni tra : Parte di, eredita da, dipende da, può usare elementi

Classi :

Prenotazione
tipoAuto : TipoAuto
puntoDiRitrovo
+cancella()
+conferma()
+Prenotazione (ta : TipoAuto, pdr : Posizione)

Packages :



-) Decomposizione: Relazione "parte di" => Una classe fa parte di un package, un package di uno più grande



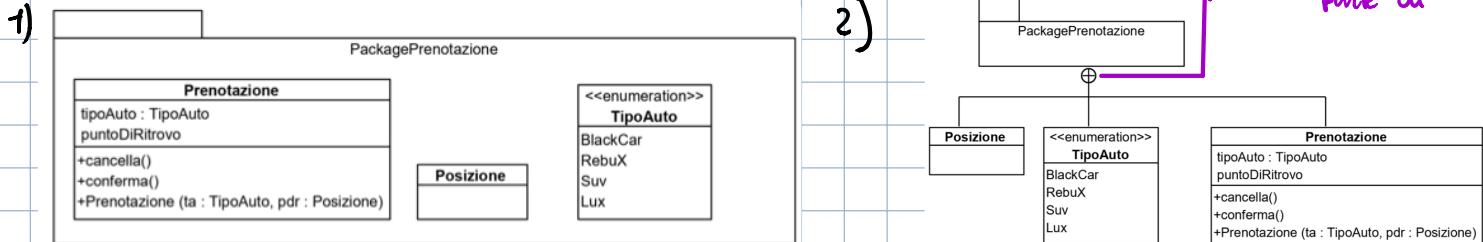
Criteri per raggruppare

- encapsulamento per modificabilità
- supporto alle scelte costruisco/compra
- moduli comuni in linee di prodotto

A cosa serve questa vista

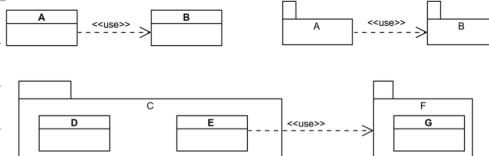
- apprendimento del sistema
- punto di partenza per l'allocazione del lavoro

## Notazione:



2) D'uso : Relazione "USA" =>

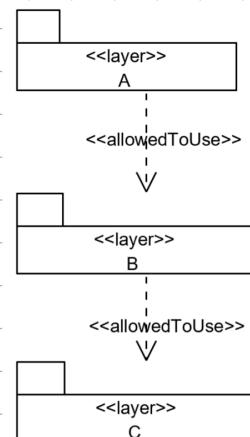
Notazione:



3) A strati:

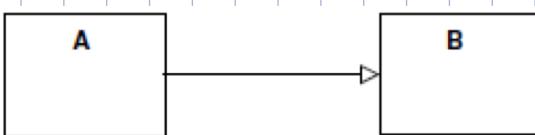
- Elementi: strati
- uno strato è un insieme coeso di moduli
  - a volte raggruppati in segmenti
- offre un'interfaccia pubblica per i suoi servizi (macchina virtuale)
- Relazione: può usare
  - Caso particolare di relazione d'uso
    - antisimmetrica (a meno di rare eccezioni)
    - non implicitamente transitiva
- A cosa serve
  - modificabilità e portabilità
  - controllo della complessità

Notazione:



4) Generalizzazione

Notazione UML generalizzazione:



## Vista logica di dislocazione (Deployment)

### ■ Elementi

- software: **artefatti**  $\Rightarrow$  informazione frutta prodotta da un processo di sviluppo SW o funz. di sistema:
- dell'ambiente: hardware, ambiente di esecuzione

### ■ Relazione

- elemento software allocato a elemento dell'ambiente

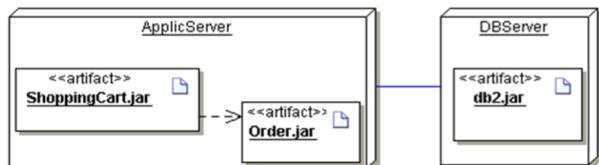
### ■ A cosa serve questa vista

- analisi delle prestazioni
- Guida per l'installazione

1) Codice Sorgente, Script ...

2) tabella DB

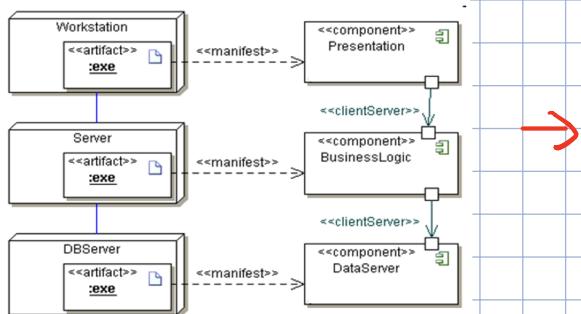
3) Doc. WORD, Mex Posta ...



- Esempio a livello di istanza, un diagramma di dislocazione può essere anche a livello di classificatori
- **Nodo**: rappresenta un nodo hardware o, in generale, un ambiente di esecuzione.
- Le relazioni tra nodi sono connessioni fisiche o protocolli di comunicazione.
- **Artefatto**: informazione prodotta dallo sviluppo o esecuzione di sw

## Viste sblide:

Esempio di architettura 3-tier



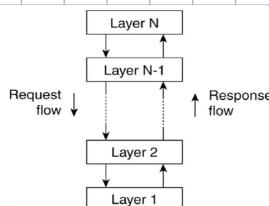
■ Un artefatto «manifesta» un componente

■ Si parla di deployment di componenti, mentre in realtà si disloca un artefatto

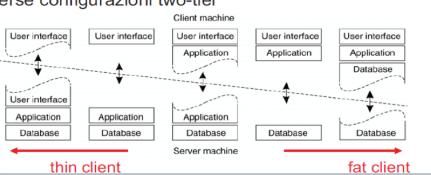
- L'artefatto è una copia di un'implementazione di componente che è stata installata/rilasciata (deployed) su un particolare computer/ambiente di esecuzione
- l'installazione avviene su un ambiente di esecuzione (nodo)
- l'installazione comprende la configurazione e la registrazione del componente in tale ambiente
- **Un artefatto «manifesta» un componente**

## Architettura a livelli:

- Componenti organizzati in livelli (**layer**)
- Un componente a livello *i* può invocare un componente del livello sottostante *i-1*
- Le richieste scendono lungo la gerarchia, mentre le risposte risalgono
- Concetto ambiguo
  - Vista C&C: catene di client-server
  - Vista Strutturale: Layers



- Mapping tra livelli logici (**layer**) e livelli fisici (**tier**)
- Architettura ad un livello (single-tier): configurazione monolitica mainframe e terminale "stupido" (non è C/S!)
- Architettura a due livelli (two-tier): due livelli fisici (macchina client/singolo server)
- Architettura a tre livelli (three-tier): ciascun livello su una macchina separata
- Diverse configurazioni two-tier



## Architettura Multi-level

- Da un livello ad N livelli
- Con l'introduzione di ciascun livello
  - L'architettura guadagna in flessibilità, funzionalità e possibilità di distribuzione
- Ma l'architettura multilivello potrebbe introdurre un problema di prestazioni
  - Aumenta il costo della comunicazione
  - Viene introdotto più complessità, in termini di gestione ed ottimizzazione

Principi di progettazione  $\Rightarrow$  1) Pianificazione del lavoro

2) Manutenzione

3) Riuso



Principi: 4) Information - Hiding

Unità:

1) Componente = elementi a run-time

2) Astrazione: Dati e controllo

2) Modulo = elementi a design time

3) Cohesione

4) Disaccoppiamento

1) Information - Hiding: Variabili private a cui si accede solo con setters and getters

- set()
- get()

OSSERVAZIONI: 1) Il get() non deve modificare lo stato dell'oggetto

2) Con il get si restituisce un dato e non il riferimento in modo che non venga modificato nell'oggetto in caso di cambiamento

3) Il set() permette una modifica controllata allo stato dell'oggetto

2) Astrazione:

Astrazione sul controllo: È un modulo identificato con il concetto di subroutine.

o procedura.

Le procedure sono utilizzate come parti di alcune classi di moduli, che prendono il nome di librerie

**Astrazione sui dati:** Modo di incapsulare un dato in una rappresentazione tale da regolamentare l'accesso e la modifica  
l'interfaccia rimane stabile anche in presenza di modifiche alle strutture dati

3) **Cohesione:** Proprietà che descrive:

- 1) Un'unita' realizza "uno e un solo concetto"
- 2) Funzionalità vicine devono stare nella stessa unità

**obiettivo:** Creare sistemi coesi, in cui tutti gli elementi di ogni unità di programmazione hanno strettamente collegati tra loro.

**ci interessa questa** es: Metodi di una classe strettamente collegati tra loro.

**IDEALE**  
**Cohesione funzionale:** Ragggruppa parti che collaborano per realizzare una funzionalità

**NON BUONE**  
**Cohesione Comunicativa:** Cohesione fra elementi che operano sugli stessi dati

di input o contribuiscono agli stessi dati di output

**Cohesione procedurale:** fra elementi che realizzano i passi di una procedura

**Cohesione temporale:** fra azioni che devono essere fatte in uno stesso arco di tempo

**Cohesione logica:** fra elementi che sono logicamente correlati e non funzionalmente

**Cohesione accidentale:** Tra elementi non correlati ma praticati insieme  
**Peggior**

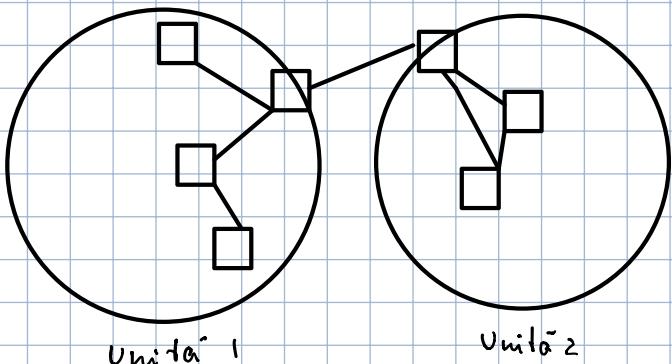
4) **Disaccoppiamento**: Creare sistemi disaccoppiati, in cui le unità di progettazione non sono strettamente legate tra loro.

↓

Si valuta tra unità diverse, non all'interno delle unità.

## Coesione vs disaccoppiamento

- Si hanno vantaggi con sistemi che esibiscono
  - un alto grado di coesione
  - un basso accoppiamento
- Maggiore riuso e migliore mantenibilità
- Ridotta interazione fra (sotto)sistemi
- Miglior comprensione del sistema
- Garantire un alto grado di coesione normalmente riduce il grado di accoppiamento.



**SOLID** : 5 principi di base di progettazione e programmazione Object-Oriented

- Single Responsibility Principle
  - Una classe (o metodo) dovrebbe avere solo un motivo per cambiare.
- Open Closed Principle
  - Estendere una classe non dovrebbe comportare modifiche alla stessa.
- Liskov Substitution Principle
  - Istanze di classi derivate possono essere usate al posto di istanze della classe base.
- Interface Segregation Principle
  - Fate interfacce a grana fine e specifiche per ogni cliente.
- Dependency Inversion Principle
  - Programma guardando le interfacce non l'implementazione.

1) **Single Responsibility Principle**: Se abbiamo 2 motivi per cambiare una classe, dobbiamo dividerla in 2

↓

classe funzionalmente coerente

Se una classe disegna e descrive una ferma geometrica, si viola se in una classe che disegna e una che descrive

2) Open Closed principle : le entità software devono essere aperte per estensione, ma chiuse per modifiche.

↓  
Usare astrazioni per aprire alle estensioni

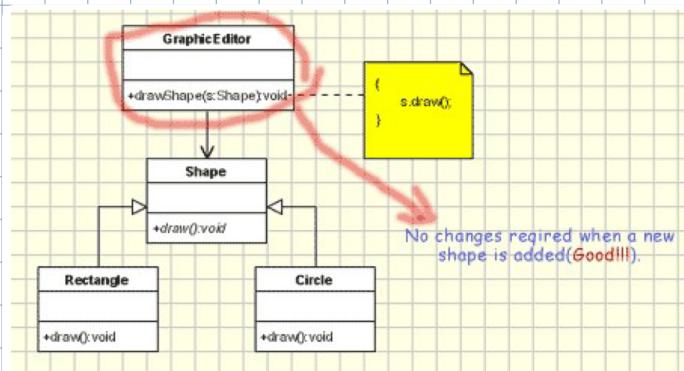
1) Ynface

2) Classi astratte

Usare delega + astrazioni

per chiusura al cambiamento

Disegnare moduli che non cambiano ma estendono il comportamento del modulo aggiungendo codice a quello vecchio



3) Princípio di sostituzione di liskov:

le classi derivate devono poter sostituire alle classi base

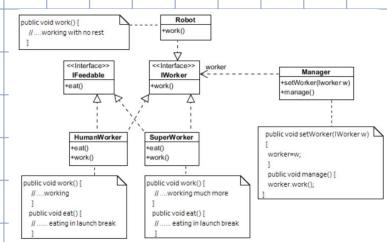
Se S è sottotipo di T, allora per ogni oggetto o1 di tipo S esiste un oggetto o2 di tipo T, tale che, dato un qualsiasi programma P definito in termini di T, il comportamento di P è immutato quando o1 è usato al posto di o2

4) Ynface Segregation principle:

Attention al modo in cui si scrivono

le interfacce:

- 1) Mettere solo metodi necessari
- 2) Ogni metodo nell'interfaccia deve essere implementato



5) Dependency Inversion Principle: Programmare per l'interfaccia, non per l'implementazione

- 1) I moduli di alto livello non devono dipendere dai moduli a basso livello. Entrambi devono dipendere da astrazioni
- 2) Le astrazioni non devono dipendere dai dettagli, ma i<sup>e</sup> viceversa

## **GRASP: progettazione guidata dalla realizzazione dei casi d'uso**

Con realizzazione del caso d'uso si intende:

- Descrivere come un particolare caso d'uso è realizzato nel progetto, in termini di oggetti collaborativi.
- Il lavoro di realizzazione del caso d'uso è un'attività di progettazione, il progetto cresce con ogni nuova realizzazione del caso d'uso.
- Per realizzare i casi d'uso si usano diagrammi di interazione e pattern
- Per realizzare i casi d'uso si **assegnano responsabilità alle classi**

**Assegnare responsabilità**



**Comportamenti di un oggetto**

Esistono due tipi principali di responsabilità:

- Fare:
  - Fare qualcosa, come creare un oggetto o fare un calcolo
  - Iniziare l'azione di altri oggetti
  - Controllare e coordinare le attività di altri oggetti.
- Conoscere:
  - Conoscere i dati privati
  - Conoscere gli oggetti correlati.
  - Conoscere dati che possono derivare o calcolare.
  - Questo tipo di responsabilità si può normalmente dedurre dal modello di dominio, dove sono illustrati gli attributi e le associazioni.

**BSS:** Una responsabilità non è un metodo, ma un metodo è implementato per soddisfare le responsabilità

**GRASP** si basa sull'assegnazione delle responsabilità:

- Si definiscono gli oggetti e i loro metodi
- Guidati da pattern di assegnazione delle responsabilità



**I 9 patterns di GRASP**

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

**Stili dell'architettura e qualità del software**

Valutiamo le caratteristiche di alcuni stili architettonici in base alle seguenti caratteristiche di qualità:

- Disponibilità
  - Capacità di offrire un servizio in modo il più possibile costante
- Fault tolerance
- Modificabilità
- Performance (efficienza)
- Scalabilità

**Scalabilità:** Capacità di aumentare il throughput pur in proporzione all'aumento dell'hardware utilizzato per ospitare l'applicazione

→ **verticale:** Aggiunta di memoria e CPU a un singolo nodo

→ **Orizzontale:** Aggiunta di più nodi HW

### Client - Server , 2 o N-tier

Disponibilità	I server di ogni tier (ordine, fila) possono essere replicati, quindi se uno fallisce c'è solo una minor QoS.
Fault tolerance	Se un cliente sta comunicando con un server che fallisce, la maggior parte dei server reindirizza la richiesta a un server replicato in modo trasparente all'utente.
Modificabilità	Il disaccoppiamento e la coesione tipici di questa arch. favoriscono la modificabilità
Performance (efficienza)	Performance ok, ma da tenere sott'occhio: numero di threads paralleli su ogni server, velocità delle comunicazioni tra server, volume dati scambiato
Scalabilità	Basta replicare i server in ogni tier (quindi ok scale out). Unico collo di bottiglia l'eventuale base di dati che scala male orizzontalmente

### Pipes and Filters

Disponibilità	Avendo "pezzi di ricambio" (componenti e possibilità di connetterle) sufficienti a formare una catena.
Fault tolerance	Occorre riparare una catena interrotta usando componenti replica.
Modificabilità	Si, se le modifiche interessano una o comunque poche componenti
Performance (efficienza)	Dipende dalla capacità del canale di comunicazione e dalla performance del filtro più lento.
Scalabilità	Ok anche scale out.

### Publish - Subscribe

Disponibilità	Si possono creare clusters di dispatcher
Fault tolerance	Si cerca un dispatcher replica
Modificabilità	Si possono aggiungere publisher e subscribers a piacere. Unica attenzione al formato dei messaggi.
Performance (efficienza)	Ok. Ma compromesso tra velocità e altri requisiti tipo affidabilità e/o sicurezza.
Scalabilità	Ok scale out: con un cluster di dispatchers si può gestire un volume molto elevato di messaggi.

### P2P

Disponibilità	Dipende dal numero di nodi in rete, ma si assume sì.
Fault tolerance	Gratis
Modificabilità	Si, se dell'architettura interessa solo la parte di comunicazione
Performance (efficienza)	Dipende dal numero di nodi connessi, dalla rete, dagli algoritmi. Per esempio BitTorrent ottimizza scaricando per primo il file/pezzo più raro.
Scalabilità	Gratis

### Coordinatore di processi

Disponibilità	Il coordinatore è un punto critico: deve essere replicato se si vuole garantire disponibilità.
Fault tolerance	Occorre specificare compensazione: cosa fare se un server fallisce. Se fallisce il coordinatore: occorre ridirigere su un coordinatore replica.
Modificabilità	Posso modificare liberamente i servers purché non cambino le funzionalità esportate.
Performance (efficienza)	Il coordinatore deve essere in grado di servire più richieste concorrenti. La performance del processo è limitata dal server più lento. Se non tutti i server sono necessari, si usa un time-out.
Scalabilità	Replicando il coordinatore. Scala sia up che out.

Diagrammi di struttura composita  $\Rightarrow$  Diagramma che mostra la struttura di dettaglio

(implementazione) di:

- Classificatore strutturato
- Collaborazione

(componente vista)  
C & C

Panti: Ha un nome, un tipo e una molteplicità

nome: tipo [molt]

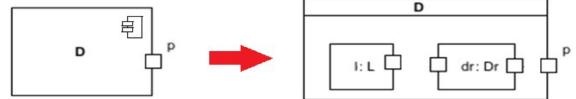
- una parte P:T descrive il ruolo che una istanza di T gioca all'interno del classificatore la cui struttura contiene P.
- La molteplicità indica quale insieme possono esserci in quel ruolo
- Una istanza di P è un'istanza di T



Porti: Gruppo di interfacce omogenee

La struttura composita che mostra la struttura di dettaglio del componente D ha

- tutti i porti di D sul proprio bordo,
- i porti associati alle parti che definiscono la struttura interna di D, che permettono le interazioni tra loro e con l'esterno.



Connettore:

1) Di assemblaggio: Connettore tra 2 parti che esprime un legame che permette la comunicazione tra due istanze dei ruoli spec. nella struttura

2) Di delega: Connettore tra una parte e un porto della componente e identifica l'istanza che realizza le comunicazioni attribuite a un porto

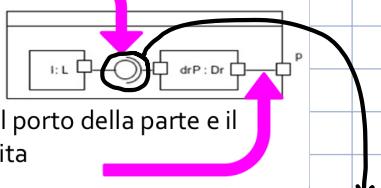
# Notazione connettori:

## Assemblaggio

- collega i porti delle due parti le cui istanze devono comunicare
- si usa la notazione lollipop

## Delega

- si usa una semplice linea tra il porto della parte e il porto della struttura composita



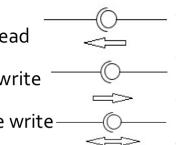
Lollipop

Il verso del lollipop non ha alcun legame con il verso in cui viaggiano i dati (frecce nei disegni): ha solo a che vedere con chi ha il controllo e con chi, interrogato, risponde

▪ un'interfaccia con solo operazioni di read

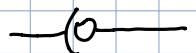
▪ un'interfaccia con solo operazione di write

▪ un'interfaccia con operazioni di read e write



• Se la logica prende dati e ci fa sopra dei conti

• Se invece la logica fa calcoli e poi ti deve inviare sul porto verso contrario



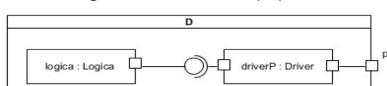
Un modo conveniente di strutturare una componente prevede di separare gli aspetti di comunicazione da quelli di realizzazione delle funzionalità richieste

- Favorisce modificabilità e comprensibilità della componente
- Risponde ai principi generali di progettazione
  - Coupling
  - Information hiding

## Come strutturare una componente:

1) Supponiamo di avere una componente D con un porto p  
La struttura di D dovrà avere almeno due parti

- driverP, che realizza la parte di comunicazione richiesta per implementare il porto
- logica, che realizza la funzionalità richiesta alla componente e due connettori
- di delega tra driverP e P
- di assemblaggio tra driverP e logica (il verso del lollipop non è fissato a priori)



2) In generale, data una componente con n porti, la sua struttura minima dovrà prevedere

- n parti col ruolo di driver
  - collegate ognuna a un porto, con un connettore di delega
- una parte che realizza la logica della componente
  - collegata a tutti i driver con connettori di assemblaggio

I nomi "driver" e "logica" per le parti di una componente sono usati in questo pattern, per aiutare a distinguere le loro responsabilità

3) È ovviamente possibile, e normalmente necessario, dettagliare maggiormente la struttura di una componente

- raffinando la "logica" in un insieme di parti interconnesse
  - con connettori di assemblaggio
  - con dipendenze
- introducendo dei "proxy", per realizzare comunicazioni con sistemi remoti chiamate al sistema operativo
  - connessi alle parti che descrivono la logica, con connettori di assemblaggio
  - non sono collegati ai porti, perché non realizzano la comunicazione con altre componenti del sistema che si sta progettando

I nome "proxy" è introdotto in questo pattern, per distinguere il ruolo da quello di un driver: non è corretto pensare sia simile al proxy di RMI

## Riassumendo:

Assumiamo C abbia 2 porti verso le componenti C1 e C1 e comunichi con un sistema esterno S

La struttura di C dovrà avere almeno 4 parti

- : DriverC<sub>1</sub> e : DriverC<sub>2</sub>, drivers che realizzano la comunicazione con C<sub>1</sub> e C<sub>2</sub>, rispettivamente
- : Logica, che realizza la funzionalità richiesta alla componente
- : ProxyS, proxy che realizza la comunicazione con S e un certo numero di connettori
- 2 di delega tra i driver e i porti che realizzano
- 3 di assemblaggio tra driver / proxy e logica (il verso del lollipop non è fissato a priori ma dipende dal sistema che si sta progettando)

**Testing**  $\Rightarrow$  Suppongo che tutto ciò che non riesco a testare sia corretto

Problema della terminazione: Non posso creare un algoritmo che dato un input e un secondo

**(Halting problem)**

algoritmo, determini se l'algoritmo termina o no per quel dato input

Dim:

Si assume che esista:

```
boolean C(a, d) { return halts(a(d)); }
```

dove halts((a(d)) restituisce true il programma a con input d (d è una sequenza di caratteri) termina, false altrimenti

Dato che un programma è sua volta una sequenza di caratteri, si può invocare C(a,a). Si può quindi definire K(a) come segue

```
boolean K(a) {  
    if C(a,a) while(true) { skip; }  
    else return false;  
}
```

Cosa succede alla terminazione del programma K(K)?

L'algoritmo K(K) termina, restituendo il valore false, solo se l'algoritmo K con input K non termina.

K(K) termina se e solo se K(K) non termina. Contraddizione!

Non può esistere l'algoritmo C!

**Undecidibilità**: la non possibilità di determinare su un prog. termina o no

## Verifica e validità

I progettisti della fase di verifica devono:

- scegliere e programmare la giusta combinazione di tecniche
  - per raggiungere il livello richiesto di qualità
  - entro i limiti di costo
- progettare una soluzione specifica che si adatta
  - al problema
  - ai requisiti
  - all'ambiente di sviluppo

Domande da porsi:

1) Quando iniziano verifiche e valutazioni? Quando sono complete?

a) Uniscono non appena decidiamo di creare un SW o anche prima

b) Durano oltre la consegna del prodotto

a) Studio di fattibilità

Lo studio di fattibilità di un nuovo progetto deve tener conto delle qualità richieste e dell'impatto sul costo complessivo

In questa fase, le attività correlate alla qualità comprendono

- analisi del rischio
- definizione delle misure necessarie per valutare e controllare la qualità in ogni stadio di sviluppo
- valutazione dell'impatto di nuove funzionalità e nuovi requisiti di qualità
- valutazione economica delle attività di controllo della qualità: costi e tempi di sviluppo

b) Dopo il rilascio

Le attività di manutenzione comprendono:

- analisi delle modifiche ed estensioni,
- generazione di nuove suite di test per le funzionalità aggiuntive,
- riesecuzione dei test per verificare la non regressione delle funzionalità del software dopo le modifiche e le estensioni
- rilevamento e analisi dei guasti

## 2) Quali tecniche applicare?

Nessuna singola tecnica di Analisi e testing è suff. per tutti gli scopi.

Le principali ragioni per combinare diverse tecniche sono:

- Efficacia per diverse classi di errori: analisi statica invece di test per le race conditions
- Applicabilità in diverse fasi del processo di sviluppo, per esempio: ispezione per la convalida dei requisiti iniziali
- Differenze negli scopi. Esempio: test statistico per misurare l'affidabilità
- Compromessi in termini di costo e affidabilità: usare tecniche costose solo per requisiti di sicurezza

Tecniche diverse in fasi diverse

## 3) Come poniamo valutare se un prodotto è pronto per essere rilasciato

Alcune misure di **dependability**:

- La **disponibilità** misura la qualità di un sistema in termini di tempo di esecuzione rispetto al tempo in cui il sistema è già
- Il **tempo medio tra i guasti** (MTBF) misura la qualità di un sistema in termini di tempo tra un guasto e il successivo
- **L'affidabilità** indica la percentuale di operazioni che terminano con successo

Organizzazione di alfa e beta test:

- **Alfa**: test da parte da sviluppatori o utenti in ambiente controllato  
orientati dall'organizzazione di sviluppo
- **Beta**: test eseguiti da utenti reali nel loro ambiente, eseguendo attività reali senza interferenze o monitoraggio raffinato

## 4) Come posso controllare la qualità delle release successive?

Attività dopo la consegna

- test e analisi del codice nuovo e modificato
- riesecuzione dei test di sistema
- memorizzazione di tutti i bug trovati
- test di regressione
  - Quasi automatico
- distinzione tra "major" e "minor" revisions
  - 2.0 VS 1.4
  - 1.5 VS 1.4

## 5) Come può essere migliorato il processo di sviluppo?

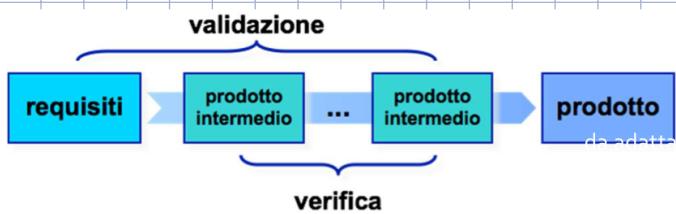
Si incontrano gli stessi difetti progetto dopo progetto

- identificare e rimozione dei punti deboli nel processo di sviluppo
  - Per esempio cattive pratiche di programmazione
- identificare e rimuovere i punti deboli del test e dell'analisi che consentono loro di non essere individuati

## Verifica vs Convalida

**Convalida**: Controlla se stiamo costituendo il sistema che serve all'utilente

**Verifica**: Controlla se stiamo costituendo un sistema che rispetta le specifiche



## Terminologia logica IEEE

**Mal funzionamento**: Il SW non si comporta secondo le aspettative o le specifiche.

Viene causato da un difetto.

**Difetto (bug o fault)**: Un difetto appartiene alla struttura statica del programma (codice).

Non sempre manifesta un malfunzionamento: **latente**

### Esempio:

```
int raddoppia (int x){  
    return x*x;  
}
```

- Con input 3, restituisce 9
  - malfunzionamento del metodo raddoppia
- Un malfunzionamento è causato dalla presenza di un difetto:
  - In questo caso l'operatore \* invece di +

es: Difetto coperto da un altro difetto

es: Difetto contenuto in un cammino che non viene praticamente mai eseguito

**Emone:** Incomprensione umane nel tentativo di comprendere o risolvere un problema, o nell'uso di strumenti.

### Limi<sup>t</sup>i teorici e prati<sup>c</sup>i nel testing

#### limite teorico

Una prova formale di correttezza di un programma corrisponderebbe all'esecuzione del sistema con tutti i possibili input.  $\Rightarrow$  impossibile

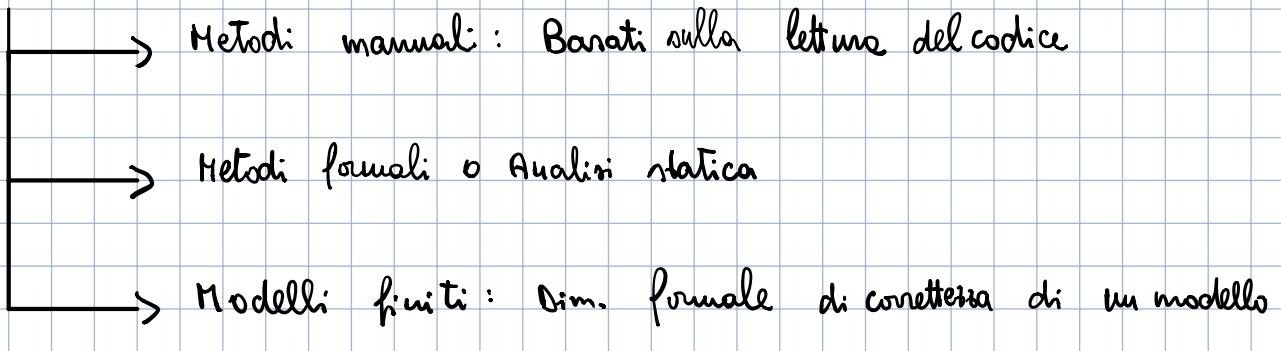
#### limite prati<sup>c</sup>o

Eseguire e provare ogni possibile input del programma (Testing Esauritivo)

- Richiede tempo infinito, se gli input sono infiniti (limi<sup>t</sup>i anche di mem. fisica)
- Tempo troppo lungo, per domini di input finiti ma molto grandi

tesi di **Dijkstra**  $\Rightarrow$  Il test di un programma puo' rilevare la presenza di difetti, ma non dimostrare l'assenza

**Verifica statica:** Verifica senza esecuzione del codice



- Pratica
- Induttiva
- Economica

## Metodi di lettura del codice

1) Inspection: lettura minata del codice guidata da una lista di controllo

↓  
Si legge il codice riga per riga  
Facilitare la ricerca su aspetti ben definiti (non generic)

Nou viene fatto dai programmatori  
(ispettori)

Fasi:

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

- Le liste di controllo sono aggiornate ad ogni iterazione di inspection
- Contengono solo aspetti che non possono essere controllate in maniera automatica
- Sono frutto dell'esperienza degli ispettori

2) Walkthrough: Percorrere il codice simulando l'esecuzione

Venne fatto sia da ispettori che sviluppatori insieme

Fasi:

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

- Affinità
  - controlli statici basati su desk-test
  - programmatore e verificatore contrapposti
  - documentazione formale

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

Inspection vs walkthrough ⇒

Verifica Dinamica

Proprietà

- Ripetibilità
- Verifica di componenti vs verifica di sistema
- Test di integrazione
- Vari tipi di test sul Sistema
- Test di accettazione (o collaudo)

Fasi

Si compone di più fasi:

- Progettazione (input, output atteso, ambiente di esecuzione ...)
- Definizione ambiente di test
- Esecuzione del codice
- Analisi dei risultati (output ottenuto con l'esecuzione vs output atteso)
- Debugging

1) Ripetibilità della prova: Viene effettuata in un ambiente definito con casi di prova definiti (in/out attesi) e procedure definite

Caso di prova (test case)

<input, output, ambiente>

atteso

## Progettazione casi di test

Si parte dal presupposto che l'adeguatezza di un insieme di casi di test è indiscutibile.

Quindi bisogna definire dei criteri che identificano inadeguatezza nei casi di test.

Esempi: Ci sono n istruzioni, i casi ne testano  $k < n$  oppure i casi di test non testano 2 casi differenti trattandoli differentemente

Un criterio di adeguatezza = numero di test obligations

Test Obligation: Descrizione di casi di test che richiede proprietà

Riferimenti importanti per il testing  $\Rightarrow$  Quale forma devono avere i test case

Esempio:

- specifica del test case: un input formato da due parole e un input formato da tre
  - i casi di test con i valori di input
    - "alpha beta"
    - "Milano Pisa Roma"
- sono due tra i tanti test che soddisfano la specifica



Come definirli:

- dalle funzionalità (a scatola chiusa, black box): dalla specifica sw
    - basati sulla conoscenza delle funzionalità
    - mirati a evidenziare malfunzionamenti relativi a funzionalità
    - Es: se la specifica richiede una procedura di recovery nel caso di mancanza di corrente, le test obligations dovranno includere la simulazione del fenomeno
  - dalla struttura (a scatola aperta, white box): dal codice
    - basati sulla conoscenza del codice
    - mirati a eseguire il codice indipendentemente dalle funzionalità
    - Es: passare da ogni loop almeno una volta
  - dal modello del programma: dal modello del sistema
    - Modelli utilizzati nella specifica o nella progettazione, o derivati dal codice
    - Es: Esercizio di tutte le transizioni nel modello di protocollo di comunicazione
  - da fault ipotetici
    - Cercano difetti ipotizzati (bug comuni)
    - Es: check per la gestione del buffer overflow testando con input molto grandi
- $\Rightarrow$  test che simulano le funzionalità a prescindere dal codice
- $\Rightarrow$  test sul codice (es: testare if ecc...)

## Criterio di Adeguatezza

Un insieme di test soddisfa un criterio di adeguatezza se:

- 1) Tutti i test hanno successo
- 2) Ogni test obbligatorio è soddisfatto da almeno un caso di test

(Nell'insieme di casi di test scelto)

Esempio: il criterio di adeguatezza della copertura delle istruzioni è soddisfatto da un insieme di test S per il programma P se ogni istruzione eseguibile in P è esercitata da almeno un test in S, e il risultato dell'esecuzione corrisponde a quello atteso.

## Progettazione di casi di test

- 1) White-box: Si basa sul presupposto di testare un prodotto conoscendo come è fatto e avendo accesso al codice.  
I vantaggi: Si possono fare dei test (unit test) più minimi rispetto agli altri approcci e che quindi potrebbero portare a rilevare problemi che difficilmente potrebbero essere rilevati altremodo.
- 2) Black-box: Si basa sull'assunto di non sapere come è realizzata una funzionalità, interessa solo cosa fa, per cui il punto di vista è (quasi) lo stesso dell'intento finale focalizzandosi sul cosa fa il sistema e sui requisiti forniti dalle specifiche. Lo vantaggio è che porta spesso ad una copertura di tutti gli scenari piuttosto scarsa per cui potrebbero esservi lasciati latenti dei bugs che potrebbero poi rilevarsi a seguito di sviluppi successivi.

## Elementi di una prova

- 1) Batteria di prove (test suite):insieme di casi di prova
- 2) Procedura di prova: procedure per eseguire, registrare, analizzare e valutare i risultati di una batteria di prove

### Condizione di una prova

- Definizione dell'obiettivo della prova
  - è importante definire l'obiettivo
- Progettazione della prova
  - la progettazione consiste soprattutto nella scelta e nella definizione dei casi di prova (della batteria di prove)
- Realizzazione dell'ambiente di prova
  - ci sono driver e stub da realizzare, ambienti da controllare, strumenti per la registrazione dei dati da realizzare
- Esecuzione della prova
  - l'esecuzione può richiedere tempo
- Analisi dei risultati
  - l'esame dei risultati alla ricerca di evidenze di malfunzionamenti
- Valutazione della prova

Test scaffolding: codice aggiuntivo per eseguire un test.

Può includere:

→ Codice fittizio che richiama le funzionalità da testare

1) Driver di test: sostituiscono un programma principale o di chiamata

2) Test harness: sostituiscono parti dell'ambiente di distribuzione

Testare una funzione con valori fittizi che simulano dei valori dati da un'altra funzione (es: db) → lo stub ritorna questi valori fittizi del db che simulano dati reali del db

3) Stub: sostituiscono funzionalità chiamate o utilizzate dal software in prova

4) Tool per gestire l'esecuzione del test

5) tool per registrare i risultati

# Metodi Black-Box per generare valori di input

- Strategia:
- 1) Separare le funzionalità da testare
  - 2) Derivare un insieme di casi di test per ogni funzionalità

Per fare ciò:

- 1) Per ogni parametro di input: Si individuano dei valori da testare
- 2) Per l'insieme dei parametri: Si usano tecniche dette "testing combinatorio" per ridurre le combinazioni

1) **Metodo Random:** Generare in modo automatico un insieme grande a piacere di valori

- Costo zero la generazione
- Non ripetibile e può essere difficile trovare l'output atteso
- Applicabile se costa poco l'esecuzione
- Difficilmente considera i casi limite
  - Esempio: trovare le radici di un'equazione di secondo grado
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
  - Quasi impossibile che il caso  $b=0, a=0$  sia generato in modo casuale

2) **Metodo Statistico:**

- I casi di test sono selezionati in base alla distribuzione di probabilità dei dati di ingresso del programma
- Il test è quindi progettato per eseguire il programma sui valori di ingresso più probabili per il suo utilizzo a regime
- Il vantaggio è che, nota la distribuzione di probabilità, la generazione dei dati di test è facilmente automatizzabile
- Non sempre corrisponde alle effettive condizioni d'utilizzo del software
- È oneroso calcolare il risultato atteso

**Esempio:**

Si consideri l'input "età il giorno della laurea":

- Il tipo è int
- In questo caso è ragionevole usare il metodo statistico e dare le specifiche di test:
  - tutti i valori compresi tra 20 e 27
  - Il 40% dei valori tra 27 e 35
    - Questi possono essere scelti in modo random
  - Il 5% dei valori tra 36 e 100
    - Questi possono essere scelti in modo random

3) **Partizione dei dati di ingresso**  $\Rightarrow$  Testo in base a come mi aspetto il funzionamento (suddivisione) delle classi di equivalenza

↓  
Un test per ogni classe di equivalenza

↓  
Raggruppamento di valori  
che mi aspetto abbiano lo stesso  
comportamento

**Esempio:**

- Il dominio dei dati di ingresso è ripartito in classi di equivalenza (categories nel libro Pezzé Young)
  - due valori d'ingresso appartengono alla stessa classe di equivalenza se, in base ai requisiti, dovrebbero produrre lo stesso comportamento del programma
- Il criterio è economicamente valido solo per quei programmi per cui il numero dei possibili comportamenti è sensibilmente inferiore alle possibili configurazioni d'ingresso
  - per come sono costruite le classi, i risultati attesi dal test sono noti e quindi non si pone il problema dell'oracolo
- Il criterio è basato su un'affermazione generalmente plausibile, ma non vera in assoluto
  - la deduzione che il corretto funzionamento sul valore rappresentante implica la correttezza su tutta la classe di equivalenza dipende dalla realizzazione del programma e non è verificabile sulla base delle sole specifiche funzionali

C. E.	Scaglioni di reddito	Aliquote
1°	Fino a € 15.000	23%
2°	Oltre a € 15.000 e fino a € 28.000	27%
3°	Oltre a € 28.000 e fino a € 55.000	38%
4°	Oltre a € 55.000 e fino a € 75.000	41%
5°	Oltre a € 75.000	43%

- metodo int calcolaTasse(int reddito)
- Proof obligation: un caso di test per aliquota
- Casi di test che soddisfano le test obligations:
  - <10.000, 2300, \_>, <20.000, 4800, \_>, ...

## 4) Valori di frontiera $\Rightarrow$ Testare su i casi limite delle classi di equivalenza

### Esempio

- Basato su una partizione dei dati di ingresso
  - le classi di equivalenza realizzate o in base all'egualanza del comportamento indotto sul programma o in base a considerazioni inerenti il tipo dei valori d'ingresso
- Dati di test: valori estremi di ogni classe di equivalenza
- Questo criterio ricorda i controlli sui valori limite tradizionali in altre discipline ingegneristiche per le quali è vera la proprietà del comportamento continuo
  - in meccanica, ad esempio, una parte provata per un certo carico resiste con certezza a tutti i carichi inferiori
- Questa proprietà però non è applicabile al software: i valori limite sono frequentemente trattati in modo particolare

Scaglioni di reddito	Aliquote
Fino a € 15.000	23%
Oltre a € 15.000 e fino a € 28.000	27%
Oltre a € 28.000 e fino a € 55.000	38%
Oltre a € 55.000 e fino a € 75.000	41%
Oltre a € 75.000	43%

- metodo int calcolaTasse(int reddito)
- Proof obligation: provare tutti gli intorni degli estremi degli intervalli
- Casi di test che soddisfano le test obligations:
  - <14.990, 3.447, ... <15.000, 3450, ..., <15.010, 3452, ..., ...
- (Per questa specifica poco significativo questo criterio: sui punti di frontiera non è derivabile ma è comunque continua)

## 5) Casi non validi

- Per ogni input si definiscono anche i casi non validi (che devono generare un errore):
  - Età inferiori a 20 o superiori a 120 per la laurea
  - Reddito negativo per il calcolo delle aliquote
  - ...

## 6) Test basato sul catalogo

- Nel tempo un'organizzazione può essersi costruita un'esperienza nel definire casi di test
- Collezionare questa esperienza in un catalogo può rendere più veloce il processo e automatizzare alcune decisioni riducendo l'errore umano
- I cataloghi catturano l'esperienza di coloro che definiscono i test elencando tutti i casi che devono essere considerati per ciascun possibile tipo di variabile

### Esempio:

- Assumiamo che una funzione usi una variabile il cui valore deve appartenere ad un intervallo di interi, il catalogo potrebbe indicare i casi seguenti come rilevanti:
  1. The element immediately preceding the lower bound of the interval
  2. The lower bound of the interval
  3. A non-boundary element within the interval
  4. The upper bound of the interval
  5. The element immediately following the upper bound
- Di fatto, si stanno considerando:
  - l'intervallo in cui è definita la funzione come se fosse un'unica classe di equivalenza
  - la sua frontiera

## 7) Testing Combinatorio : Tecniche da applicare al crescere del numero dei parametri in input

$$f(x, y, z, \dots)$$

In presenza di più dati di input, se si prende il prodotto cartesiano dei casi di test individuati, facilmente si ottengono numeri non gestibili

Esplorazione Combinatoria :



Tecniche per ridurre l'esplosione combinatoria

- Vincoli
- Pairwise testing

**Vincoli:** servono per ridurre le possibili combinazioni

## 1) Vincoli di errore

- Immaginiamo 5 parametri di input:  $\langle x_1, x_2, x_3, x_4, x_5 \rangle$
- Dominio di  $x_1$  e  $x_2$  ripartibile in 8 classi (di cui una di valori non validi  $\rightarrow$  errore)
- Dominio di  $x_3$  e  $x_5$  ripartibile in 4 classi (di cui una di valori non validi  $\rightarrow$  errore)
- Dominio di  $x_4$  ripartibile in 7 classi (di cui una di valori non validi  $\rightarrow$  errore)
- Un rappresentante per classe:  $8 \times 8 \times 4 \times 7 \times 4 = 7.168$  casi di test

testo con  $x_1 = \text{err}$  o gli altri non err.  
testo con  $x_2 = \text{err}$  e gli altri non err.  
:  
:

- $\langle x_1, x_2, x_3, x_4, x_5 \rangle$ , come prima
- Un rappresentante per classe:  $8 \times 8 \times 4 \times 7 \times 4 = 7.168$  casi di test

■ Viene preso un solo caso, per ogni posizione, con input non valido  
■  $5 + 7 \times 7 \times 3 \times 6 \times 3 = 2.651$   
■ Da 7.168 a 2.651

per testare le classi di errori

tutti per ogni classe di errore classi di errori non errori

## 2) Vincoli di Proprietà $\Rightarrow$ Mettiamo insieme solo i casi di test di classi compatibili:

- $\langle x_1, x_2, x_3, x_4, x_5 \rangle$ , avevamo  $5 + 7 \times 7 \times 3 \times 6 \times 3 = 2.651$ 
  - $x_1$ :  
classe 1, classe 2, classe 3, classe 4 [property negativi]  
classe 5, classe 6, classe 7 [property positivi]  
(classe 8 [errore])
  - $x_2$ :  
classe 1, classe 3, classe 5, classe 7 [if negativi]  
classe 2, classe 4, classe 6 [if positivi]  
(classe 8 [errore])
- $5 + (4 \times 4 + 3 \times 3) \times 3 \times 6 \times 3 = 5 + 1350 = 1.355$
- da 7.168 a 2.651 a 1.355

Metto insieme quelli compatibili:

## 3) Vincoli (single)

- Per uno (o più) parametri si può decidere di testare un solo valore (come per error)
- per esempio  $x_4$  [single]
- $5 + (4 \times 4 + 3 \times 3) \times 3 \times 1 \times 3 = 5 + 225 = 230$
- da 7.168 a 2.651 a 1.355 a 230

al posto di testare tutte le classi di equiv, ne testo solo 1 se quel parametro non è rilevante rispetto alla funzione

# Painwise Testing : Combinazione di test basato su coppie

- La tecnica basata su vincoli vista precedentemente permette di introdurre vincoli che limitino il numero di test ottenuti dalla generazione di tutte le combinazioni di valori possibili.
- Funziona bene se i vincoli che imponiamo sono reali vincoli del dominio e non se li aggiungiamo al solo scopo di limitare le combinazioni
- Nel caso in cui il dominio non contenga in sé questi vincoli è preferibile optare per la generazione di tutte le combinazioni solo per i sottoinsiemi di k variabili, con  $k < n$  (pairwise con  $k=2$ )
- L'idea: generare tutte le possibili combinazioni solo per  $k$  variabili
- se  $k=2$  genero tutte le combinazioni per tutte le possibili coppie di variabili
- Quanto si risparmia?

Esempio:

<b>Display Mode</b>	<b>Language</b>	<b>Fonts</b>
full-graphics	English	Minimal
text-only	French	Standard
limited-bandwidth	Spanish	Document-loaded
Portuguese		
<b>Color</b>		
Monochrome	<b>Screen size</b>	
Color-map	Hand-held	
16-bit	Laptop	
True-color	Full-size	

- Se volessimo generare tutte le combinazioni per *Display mode*, *Screen size* e *Fonts* avremmo

$$3^3 = 27$$

- Se invece generiamo tutte le combinazioni solo per la coppia *<Display mode, Screen size>* abbiamo

$$3^2 = 9$$

- il valore del terzo parametro può essere aggiunto in modo da coprire tutte le combinazioni di *Fonts*x*Screen size* e *Fonts*x*Display mode*

<b>Display Mode</b>	<b>Language</b>	<b>Fonts</b>
full-graphics	English	Minimal
text-only	French	Standard
limited-bandwidth	Spanish	Document-loaded
Portuguese		
<b>Color</b>		
Monochrome	<b>Screen size</b>	
Color-map	Hand-held	
16-bit	Laptop	
True-color	Full-size	

<i>Display mode</i> × <i>Screen size</i>		<i>Fonts</i>
Full-graphics	Hand-held	Minimal
Full-graphics	Laptop	Standard
Full-graphics	Full-size	Document-loaded
Text-only	Hand-held	Standard
Text-only	Laptop	Document-loaded
Text-only	Full-size	Minimal
Limited-bandwidth	Hand-held	Document-loaded
Limited-bandwidth	Laptop	Minimal
Limited-bandwidth	Full-size	Standard

- La generazione di combinazioni che in maniera efficiente coprono tutte le coppie è impossibile da fare a mano per molti parametri con molti valori ma può essere fatta con euristiche.

L'idea è quindi diminuire il numero di test: vorremmo esercitare tutte le coppie di input (che sono 27) tuttavia sono troppe. Voglio una combinazione delle 3 (*Display Mode*, *Screen Size* e *Fonts*) e dunque mi restringo a creare combinazioni solo di due per volta. Le combinazioni tra *Display Mode* e *Screen Size* sono un semplice prodotto cartesiano tuttavia il terzo parametro, *Fonts*, deve essere scelto in modo da non generare coppie inutili. Dunque troveremo tutte le combinazioni tra *Display Mode* e *Screen size* ma non con tutte le combinazioni di *Fonts*.

Dunque ho la garanzia di avere tutte le coppie di *Display Mode* e *Screen Size* nonostante non coprano tutti i valori (ogni riga) non abbia tutte le combinazioni di *Fonts*.

## Metodi white-Box per generare valori di input

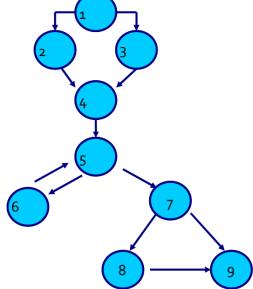
Usati per eseguire e coprire tutte le parti di un programma

- ↓
- Comandi
- Branches (decisioni)
- Condizioni
- Cammini

GRAFO DI FLUSSO: Grafo di esecuzione del programma cioè come il flusso di esecuzione passa attraverso i vari step del programma

### CODICE

```
double eleva(int x, int y) {  
    1. if (y<0)  
    2.     pow = 0-y;  
    3.     else pow = y;  
    4.     z = 1.0;  
    5.     while (pow!=0)  
    6.         { z = z*x; pow = pow-1 }  
    7.     if (y<0)  
    8.         z = 1.0 / z;  
    9.     return(z);  
}
```



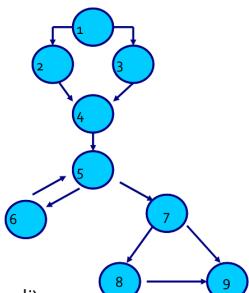
1) Copertura dei comandi ⇒ Esegui tutte le istruzioni del programma

```
double eleva(int x, int y) {  
    1. if (y<0)  
    2.     pow = 0-y;  
    3.     else pow = y;  
    4.     z = 1.0;  
    5.     while (pow!=0)  
    6.         { z = z*x; pow = pow-1 }  
    7.     if (y<0)  
    8.         z = 1.0 / z;  
    9.     return(z);  
}
```

Insieme di valori per x e y che esercitino i comandi

Esempi:

- {(x=2,y=2), (x=0,y=0)} (non esercita tutti i comandi)
- {(x=-2,y=3), {(x=4,y=0), (x=0, y=-5)}} (esercita tutti i comandi)

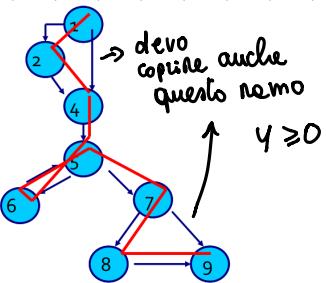


Misura di copertura =  $\frac{\text{numero di comandi eseguiti}}{\text{numero di comandi totali}}$

- Non è detto che aumentando il numero dei test, aumenti la % di copertura cioè 1 test può avere più copertura rispetto a 2 test
- Non sempre vale lo stesso cercare a tutti i costi un insieme minima che dia copertura al 100 %.

## 2) Copertura delle decisioni

```
double eleva(int x, int y){
    pow=y;
    if (y<0)
        pow = -pow;
    z = 1.0;
    while (pow!=0)
        { z = z * x; pow = pow-1}
    if (y<0)
        z = 1.0 / z;
    return(z);
}
```



Posso esercitare tutti i comandi con  $(x=2, y=-1)$   
ma non mi accorgo del fatto che manca il ramo else!  
Devo avere casi di test che esercitino entrambi i rami di una condizione.

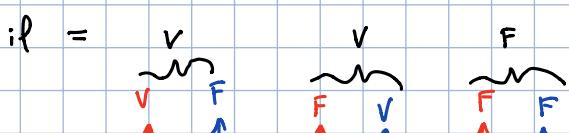
Misura di copertura =  $\frac{\text{numero di archi esercitati}}{\text{numero di archi totali}}$

Per ogni condizione(if) e while devo copiare sia il caso dove entra  
sia quello dove la condizione non è verificata

Caso condizioni composte:

voglio testarle entrambe

```
if (x>1 || y==0) {comando1}      => Non voglio testare solo che l'if sia
else {comando2}                       vero o falso, ma anche le singole
                                          decisioni
```



Il test  $\{x=2, y=1\}, \{x=0, y=0\}, \{x=1, y=1\}$  esercita tutti i valori di verità delle due condizioni e tutte le decisioni

$x=2$ $\downarrow$ $V$	$y=0$ $\uparrow$ $V$	$\rightarrow \text{if} = V$
$x=-1$ $\downarrow$ $F$	$y=1$ $\uparrow$ $F$	$\rightarrow \text{if} = F$

} altro test che esercita tutti i valori di verità delle due condizioni e dell'if

es:  $x=0$

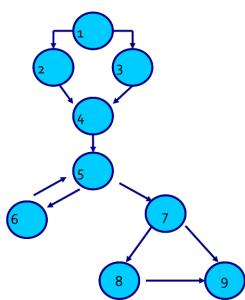
- Copertura delle basic conditions =  $\frac{\text{n. di valori di verità assunti dalle basic conditions}}{2 * \text{n. di basic conditions}}$

## Multiple condition coverage: tabella di verific per le condizioni composte

- si consideri il codice  
if ( $x > 1 \&& y == 0 \&& z > 3$ ) {comando1}  
else {comando2}
- La multiple condition coverage richiede di testare tutte le possibili combinazioni ( $2^n$  con n condizioni semplici)
- Nell'esempio sarebbero  $2^3$  casi, ma (semantica Java di  $\&\&$ ) ci si può ridurre da a 4:
  - vero, vero, vero
  - vero, vero, falso
  - vero, falso, -
  - falso, -, -

## 3) Copertura dei cammini

- Richiede di percorrere tutti i cammini
- In presenza di cicli il numero di cammini è potenzialmente infinito
- Per limitare il numero di cammini da attraversare si richiedano casi di test che esercitino il ciclo
  - o volte,
  - esattamente una volta
  - più di una volta
- Alcuni cammini impossibili (1245679)



Si seguono i cammini possibili e si vede se si fanno tutti.

↓  
perché l'if è  $y < 0$   
per entrambe le decisioni

Y metodi white-box e blackbox non sono in alternativa ma vengono usati entrambi!

## Fault Based testing :



### Validation della batteria di test

- Ipotizza dei difetti potenziali nel codice sotto test

- Crea o valuta una test suite sulla base della sua capacità di rilevare i difetti ipotizzati

- La più nota tecnica di fault based testing è il **test mutazionale**

- Si iniettano difetti modificando il codice

## TEST MUTAZIONALE :



Andiamo a modiificare il codice per vedere se la batteria di testing usata finora ad ora fallisce nella versione modificata.

- Dopo aver eseguito un programma  $P$  su una batteria di test  $T$ , si verifica  $P$  corretto rispetto a  $T$ .
- Si vuole fare una verifica più profonda sulla correttezza di  $P$ : si introducono dei difetti (piccoli, dette **mutazioni**) su  $P$  e si chiama il programma modificato  $P'$ . Questo  $P'$  viene detto **mutante**.
- Si eseguono su  $P'$  gli stessi test di  $T$ . Il test dovrebbe manifestare dei malfunzionamenti.
- Se il test non rileva questi difetti, allora significa che la batteria di test non era abbastanza buona
- Se li rileva, abbiamo una maggior fiducia nella batteria di test.
- Questo è un metodo per valutare la capacità di un test, e vedere se è il caso di introdurre test più sofisticati.



- mutazione**: cambiamento sintattico (un bug inserito nel codice)
- Esempio: modifica  $(i < 0)$  in  $(i \leq 0)$
- Un mutante viene **ucciso** se fallisce almeno in un caso di test
- efficacia di un test** = quantità di mutanti uccisi
- La tecnica si applica in congiunzione con altri criteri di test
- Nella sua formulazione è prevista infatti l'esistenza, oltre al programma da controllare, anche di un insieme di test già realizzati.

Specifico: la funzione foo restituisce  $x+y$  se  $x \leq y$  e  $x*y$  altrimenti

```
\original
int foo(int x, int y)
{ if(x <= y)
    return x+y;
  else return x*y; }
```

```
\versione modificata (mutante)
int foo(int x, int y)
{ if(x < y)
    return x+y;
  else return x*y; }
```

Consideriamo la seguente batteria di test

$\{(0,0), 0\}, \{(2,3), 5\}, \{(4,3), 12\}$

Non fallisce né nell'originale né nella versione modificata

→ si dice che il mutante non viene ucciso

→ la batteria è poco efficace e va riprogettata

(anche se copre: criteri strutturali : tutte le decisioni, tutte le istruzioni, criteri funzionali : le classi di equivalenza e la frontiera)



Una test suite che uccide i mutanti  
è capace di trovare difetti reali nel programma



### Esempi di mutationi

- crp**: sostituzione (replacement) di costante per costante
  - ad esempio: da  $(x < 5)$  a  $(x < 12)$
- ror**: sostituzione dell'operatore relazionale
  - ad esempio: da  $(x \leq 5)$  a  $(x < 5)$
- vie**: eliminazione dell'inizializzazione di una variabile
  - cambia  $int x = 5;$  a  $int x;$
- lrc**: sostituzione di un operatore logico
  - Ad esempio da  $&$  a  $|$
- abs**: inserimento di un valore assoluto
  - Da  $x$  a  $|x|$

- Attenzione:**
- Mutante invalido : Mutante che non porta la compilazione
  - Mutante valido : Mutante che supera la compilazione

- Un mutante è utile se valido
- Un mutante è utile se non è equivalente alla porzione di codice modificata!

### Perche utilizzare questa strategia:

Questa strategia è adottata con obiettivi diversi

- favorire la scoperta di malfunzionamenti ipotizzati: intervenire sul codice può essere più conveniente rispetto alla generazione di casi di test ad hoc.
- valutare l'efficacia dell'insieme di test, controllando se "si accorge" delle modifiche introdotte sul programma originale.
- cercare indicazioni circa la localizzazione dei difetti la cui esistenza è stata denunciata dai test eseguiti sul programma originale

### Come trovare l'output atteso degli input trovati:

- Risultati ricavati dalle specifiche
  - specifiche formali
  - specifiche eseguibili
- Inversione delle funzioni
  - quando l'inversa è "più facile"
  - a volte disponibile fra le funzionalità
  - limitazioni per difetti di approssimazione
  - Partire dall'output e trovare l'input
  - Per esempio per testare un algoritmo di ordinamento prendere un array ordinato (output atteso) e rimescolarlo per ottenere un input

- Semplificazione dei dati d'ingresso
  - provare le funzionalità su dati semplici
  - risultati noti o calcolabili con altri mezzi
  - ipotesi di comportamento costante

- Semplificazione dei risultati
  - accontentarsi di risultati plausibili
  - tramite vincoli fra ingressi e uscite
  - tramite invarianti sulle uscite

- Versioni precedenti dello stesso codice
  - disponibili (per funzionalità non modificate)
  - prove di non regressione
- Versioni multiple indipendenti
  - programmi preesistenti (back-to-back)
  - sviluppate ad hoc
  - semplificazione degli algoritmi
  - magari poco efficienti ma corrette

Y test normalmente sono automatizzati

