



UNIVERSITÀ DEGLI STUDI DI SALERNO

**Dipartimento dell'Ingegneria dell'Informazione ed
Elettronica e Matematica applicata**

Corso di Laurea in *Ingegneria Informatica* / L-8

Software Engineering

Appunti delle lezioni di *Gennaro Francesco Landi*

Docente: *Capuano Nicola*
A.A. 2024-2025

*"Aggiungere manodopera a un progetto software in ritardo
lo renderà ancora più in ritardo."
(F. Brooks, 1975)*

Indice

1	Introduzione	4
1.1	Scrivere un programma	4
1.2	Costruire un sistema	5
1.3	Ingegneria del Software	6
2	Il ciclo di vita del software	8
2.1	Processo Software	8
2.2	Modelli di Produzione del Software	9
2.2.1	Attività principali di un processo	9
2.3	Modelli di Processo tradizionali	10
2.3.1	Modelli di processo iterativi	11
3	Ingegneria dei Requisiti	12
3.1	Attività di Ingegneria dei Requisiti	12
3.1.1	Elicitazione dei requisiti	13
3.1.2	Analisi dei requisiti	14
3.2	Casi d'uso e UML	17
3.3	Introduzione a UML	17
3.4	Diagrammi dei casi d'uso - Use Case Diagram	19
3.5	Specifiche di un Componente Software	21
3.5.1	La metafora del contratto	22
3.6	Specifiche, Contratti e Documentazione	25
3.6.1	Contratti ed encapsulamento	25
3.6.2	Documentazione della specifica	26
4	Progettazione del software	28
4.1	Introduzione alla progettazione del software	28
4.2	Fasi della progettazione	29
4.3	Progettazione architetturale	31
4.4	Progettazione in dettaglio	34
4.4.1	Diagramma dei package e dei componenti	37
4.5	Esercizio sulla Progettazione - Tris	38
4.6	Progettazione Object-Oriented	39
4.6.1	39
4.6.2	Diagrammi delle Classi	40
4.6.3	Ereditarietà, contratti e dipendenze	47
4.7	Diagrammi di Interazioni	47
4.7.1	Diagrammi di interazione	48
4.7.2	Diagrammi di Sequenza	48
4.7.3	Diagrammi di Comunicazione	50

4.8	Diagramma delle attività - altri diagrammi UML	50
4.8.1	Diagrammi di macchina a stati	55
4.8.2	Diagrammi di distribuzione	58
5	Principi di Buona Progettazione	60
5.1	Principi generali di buona progettazione	62
5.1.1	Principi di buona progettazione orientata agli oggetti	63
6	Testing e debugging	65
6.1	Introduzione del testing	65
6.2	Le attività di testing	66
6.2.1	Concetti di test	66
6.2.2	Attività di test	69
6.3	Test unitari	70
6.3.1	Tecniche per l'individuazione dei casi di tests	70
6.4	Sviluppo guidato dai test (Test Driven Development)	72
7	Strumenti per il build ed il controllo delle revisioni	74
7.1	Debugging	74
7.2	Version Control	74
7.3	Git	75
7.3.1	Git: concetti fondamentali	75
7.3.2	Git: Comandi principali	76
7.3.3	Branch e Merge	77
7.4	Git e NetBeans	78
7.5	Git Remote	78
8	Build Automation	80
8.1	Build automation in C con make	80

Capitolo 1

Introduzione

Per realizzare un buon programma che soddisfa al meglio il problema reale, bisogna capire tutte le **richieste** e le **funzioni** imposte dal cliente sui design, e bisogna fare importanti decisioni tecniche. Successivamente bisogna progettarlo, realizzarlo e testarlo.

1.1 Scrivere un programma

Fasi per la *scrittura di un programma*:

1. Comprensione del problema:

Quindi, una **descrizione completa di un problema** deve specificare:

- i **requisiti**: necessari per assicurarsi che il programma prodotto risolva il *problema reale* per cui è stato prodotto.
Spesso i requisiti sono:
 - (a) **negoziabili**: alcune caratteristiche sono *necessarie* altre sono *desiderabili*;
 - (b) **Funzionali / Non-Funzionali**: sono *funzionali* (*cosa un programma deve fare*) se hanno una misura 'booleana' (soddisfatto / non soddisfatto). Sono invece *non funzionali* (*il modo in cui i requisiti devono essere raggiunti*) se si misurano su una *scala lineare*, ovvero possono avere un diverso grado di soddisfazione (es. il tempo di risposta di una funzione).
- i **design constraints**: vincolano il modo in cui il software deve essere progettato e realizzato. Spesso sono considerati come *requisiti non-funzionali*.

Alcuni esempi sono:

- Piattaforma
- Tecnologie da usare
- Integrazione con sistemi esistenti
- Dimensioni dei dati da trattare
- Tempi di realizzazione

Alcune di queste caratteristiche possono essere **vincoli di progettazione** mentre altre sono **scelte di progettazione**

2. Progettazione del problema:

- Identifichiamo i **componenti**

- Definiamo le **interazioni**
- Progettiamo la **user interface**
- Definiamo le altre **interfacce** (verso altri sistemi)
- Definiamo/scegliamo gli **algoritmi** da usare

Tutte queste scelte vengono fatte in base a *requisiti e design constraints*.

3. **Implementazione del programma:** traduciamo le decisioni di progettazione in **codice** vero e proprio
4. **Test del programma:**

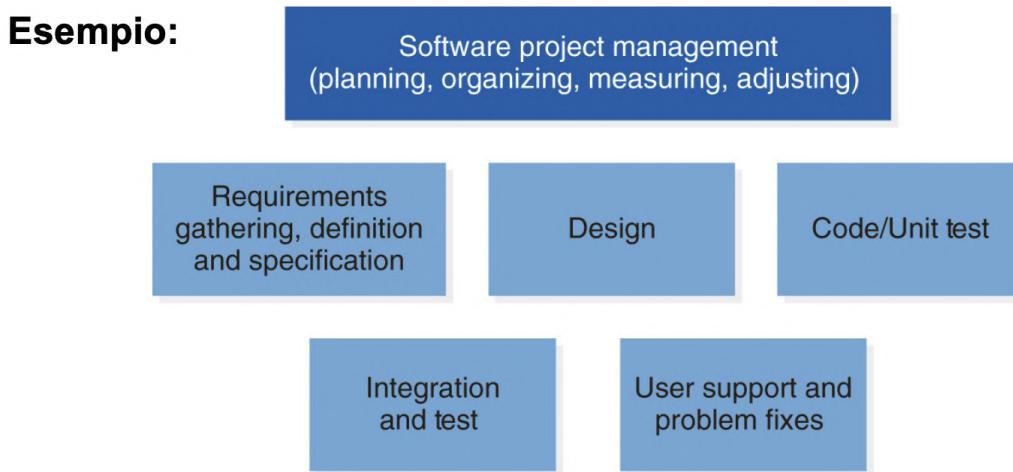
- Verifichiamo i **risultati** ottenuti usando dei dati di input noti (**case test**)
- I *casi di test* devono essere scelti in modo appropriato
- Se i risultati sono diversi da quelli attesi si esegue il **debug** (individuazione del problema), **fix** (correzione del problema) e **re-test**
- Ci fermiamo quando *tutti i casi di test* producono il *risultato atteso*

Un aspetto importante in un progetto software è la **stima della quantità di lavoro richiesta (effort)**.

1.2 Costruire un sistema

Un sistema software, a differenza di un programma, ha più funzionalità, più caratteristiche per ogni funzionalità, più utenti, più dati e più connessioni con altri sistemi.

Il **processo di Produzione del Software** è l'insieme delle *attività* necessarie per la costruzione di un sistema software.



Processo di Produzione del Software

- L'insieme delle *attività* necessarie per la costruzione di un sistema software
- La *sequenza* in cui le attività si susseguono
- Gli *input* e gli *output* di ciascuna attività
- I *partecipanti* di ciascuna attività

- Le *precondizioni* per stabilire quando un'attività può cominciare
- Le *postcondizioni* per stabilire quando un'attività può considerarsi completa

Però, l'aumento del personale aumenta in numero di comunicazione che induce a *maggiori rischi di errori di comunicazione*.

The Chaos Report - 1995

Studio realizzato da *Standish Group*, analizzati 300 progetti software di grandi dimensioni e solo il 16% di progetti è risultato completo nei **tempi** e nei **costi previsti**.

Principali cause di fallimento:

- Mancata comunicazione con gli utenti
- Requisiti e specifiche incompleti
- Requisiti e specifiche modificati nel corso del progetto

Attributi comuni ai progetti completati con successo:

- Coinvolgimento degli utenti
- Supporto da parte dei vertici aziendali
- Requisiti chiari
- Pianificazione adeguata

La legge di Brooks: aggiungere manodopera ad un progetto software in ritardo lo renderà ancora più in ritardo.

Questo perchè le aziende misurano il costo di un progetto in *mesi persona*. Un'assunzione implicita: **tempo e persone sono intercambiabili**.

1.3 Ingegneria del Software

Una disciplina ingegneristica che si occupa di tutti gli aspetti della **produzione** del software.

Ingegneria:

- uso appropriato di *modelli, metodi e strumenti*
- *soluzioni innovative* quando le soluzioni esistenti non sono adeguate
- rispetto dei *vincoli* ed *uso efficace ed efficiente* delle **risorse**

Produzione:

- la scrittura del *codice* è solo una delle **attività tecniche** necessarie
- oltre alle attività tecniche, ci sono anche le **attività non tecniche**

L'Ingegneria del Software è un ampio settore che tocca tutti gli aspetti dello sviluppo e del supporto di un sistema software, includendo le seguenti aree chiave:

- Processi tecnici e processi aziendali

- Metodologie e tecniche specifiche
- Caratterizzazione dei prodotti e metriche per misure quantitative
- Abilità delle persone e lavoro di squadra
- Coordinamento e gestione dei progetti

Conclusioni

La **produzione del software** è *complessa* e comporta *rischi* economici significativi. I fattori chiave non sono solo *tecnicici*, ma anche *umani*.

Per gestire queste sfide è necessaria una *metodologia strutturata*, fondata su un *approccio ingegneristico*.

Ogni progetto è **unico**, non esiste una *soluzione universale* che vada bene per tutte le sfide.

Capitolo 2

Il ciclo di vita del software

2.1 Processo Software

Un **Processo di Produzione del Software** fornisce un insieme di **regole** che consentono di *svolgere, coordinare e controllare* in maniera sistematica le *attività* necessarie per raggiungere gli *obiettivi* di un progetto software.

Un *processo software* definisce:

- un insieme di **attività (Task)** che devono essere svolte:
 - Unità di lavoro assegnate ad un ruolo, che ricevono degli input e producono degli output, che possono essere *documenti, codice, modelli da sviluppare*
 - Un'attività può cominciare solo quando sono soddisfatte delle specifiche condizioni (*entry condition*)
 - Un'attività si può considerare conclusa solo quando sono soddisfatte delle specifiche condizioni (*exit condition*)
- un insieme di **ruoli** che devono essere ricoperti dalle persone che partecipano al processo. Un ruolo definisce un insieme di *competenze/abilità* richieste alle persone che ricoprono quel ruolo, un insieme di *attività e responsabilità* per le persone con quel ruolo.
Esempi di ruoli: Project Manager, Software Architect, Developer, Tester, UX/UI Designer, Analyst.
- delle **linee guida** sullo svolgimento del progetto, per esempio delle buone pratiche, prioritò, errori comuni da evitare

Perché serve un *Processo Software* definito esplicitamente ?

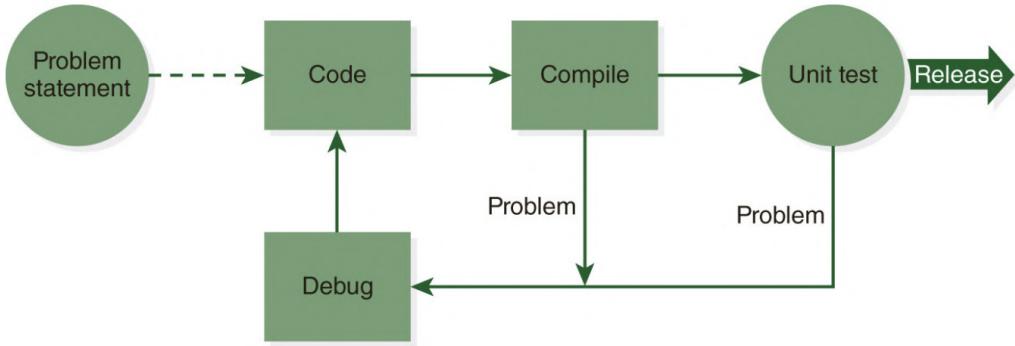
- Migliora la **comprendizione delle attività** da eseguire da parte dei membri del team
- Rende più chiare le **responsabilità, le aspettative ed i risultati** attesi dai membri del team
- Rende più facile **misurare e controllare** l'*avanzamento* del progetto

2.2 Modelli di Produzione del Software

Un **modello di processo software** è una *descrizione astratta* di un processo software, che non presenta tutti i dettagli, ma si concentra sui alcuni aspetti importanti come l'ordine in cui devono essere svolte determinate attività.

Il modello di processo più semplice è il *Code-and-fix model*:

Code-and-fix model:



2.2.1 Attività principali di un processo

- **Analisi dei requisiti:** l'obiettivo è determinare i requisiti *funzionali/non-funzionali*, ovvero **quali funzioni** deve svolgere il software e **quali vincoli** devono essere considerati. I requisiti devono essere *documentati e validati* dal committente
- **Progettazione (Design):** l'obiettivo è definire la **struttura del software** costruendo diversi *modelli software*. Si passa dai quali requisiti a come realizzarli.
- **Implementazione:** l'obiettivo è **trasformare il design** in un **programma funzionante**. Viene lasciata libertà al programmatore che dovrà prendere delle decisioni critiche per la meglio realizzazione commentando il codice, senza il bisogno necessario di modelli.
- **Integrazione e testing:** l'*integrazione* combina diversi componenti per formare un sistema completo (evitando diverse interpretazioni del design), il *testing* verifica sperimentalmente che il software funziona come dovrebbe.

Esistono diversi *livelli di testing*:

- **Unit testing:** test dei singoli componenti
- **Integration testing:** test per verificare che componenti diversi funzionino correttamente se usati insieme
- **System testing:** test del sistema completo, generalmente fatto da un gruppo di tester
- **Acceptance testing:** test finale, condotto dal committente

- **Rilascio e Manutenzione:**

- **Deployment:** installazione del software nell'ambiente reale (sul cloud, sullo store)
- **Attività Post-Deployment:** *Data Migration* (trasferimento dati da sistemi preesistenti), *Bug Fixing*, *Manutenzione adattiva* (modifiche per adattare il sistema a cambiamenti dell'ambiente in cui opera) o *evolutiva* (aggiunta di nuove funzioni o miglioramento delle funzioni esistenti).

Costo delle attività:

- Requirements: 15%
- Design: 25%
- Implementation: 20%
- Integration, testing, release and maintenance: 40%

2.3 Modelli di Processo tradizionali

Modello a cascata - Waterfall

Il primo modello di processo definito formalmente. Le attività si svolgono **sequenzialmente**, ciascuna fase riceve come input l'output dalla precedente. Consente di descrivere lo *stato di avanzamento* in maniera più precisa di "quasi finito".

Caratteristiche:

- ogni attività deve terminare *prima che la successiva cominci*
- gli output di ogni attività sono *documentati, verificati e approvati*
- ogni attività si basa *esclusivamente* sugli *output della precedente*

Vantaggi:

- *Struttura chiara* con fasi, input ed output ben definiti
- Semplifica la *divisione del lavoro, il monitoraggio* e la *gestione* del progetto

Svantaggi:

- *Feedback degli utenti* solo alla fine
- i *cambiamenti sono costosi*, specialmente sulle fasi iniziali
- se ci si ferma si rischia di *non avere risultati intermedi*

Applicabilità:

- progetti di *grandi dimensioni*, sviluppati da team altrettanto grandi
- progetti in cui è importante *poder verificare formalmente gli output* di ciascuna attività
- progetti in cui i *requisiti non cambiano* nel tempo

Varianti del modello a cascata:

- **Waterfall con feedback:** dopo il deployment si gestisce il feedback per modificare ogni fase
- **Waterfall con feedback ad ogni fase:** alla fine di ogni fase si ottengono feedback per la fase precedente, alla fine ho feedback per tutte le fasi.

2.3.1 Modelli di processo iterativi

Sviluppano il sistema in più **iterazioni**: ciascuna iterazione comprende le diverse attività come analisi dei requisiti, progettazione, implementazione, integrazione e testing.

Vantaggi:

- ideali per i progetti in cui *cambiano i requisiti nel tempo* oppure *non sono interamente definiti all'inizio*
- consentono di avere *ad ogni iterazione un feedback* dagli stakeholder (utenti ed altri soggetti interessati)
- eventuali problemi sono *individuati e risolti* prima

Esempi di processi iterativi: incrementale, a spirale, Unified Process.

Modello incrementale

Il progetto è diviso in **incrementi**, che vengono rilasciati in momenti successivi (*release*):

- ogni incremento contiene un **sottoinsieme** (crescente) delle funzionalità dell'intero sistema
- ogni incremento è un *sotto-progetto*, in cui si svolgono tutte le attività che fanno normalmente parte di un progetto.

Vantaggi:

- il cliente ottiene velocemente un **sottoinsieme funzionante e fondamentale** del sistema
- le prime release aiutano a *capire meglio i requisiti* per quelle successive
- *riduzione del rischio* di fallimento completo
- le *funzionalità più importanti sono realizzate per prime* e testate per più tempo

Svantaggi:

- è più difficile *sfruttare il parallelismo* in grandi team

Applicabilità:

- progetti di **dimensioni medio/piccole**
- progetti per cui è prevista una *lunga durata*
- *requisiti che cambiano* nel tempo

Capitolo 3

Ingegneria dei Requisiti

L'**ingegneria dei requisiti** è l'insieme delle attività necessarie alla definizione e alla comprensione:

- delle *funzionalità* che un cliente richiede ad un sistema
- dei *vincoli* sotto i quali il sistema viene sviluppato ed opera

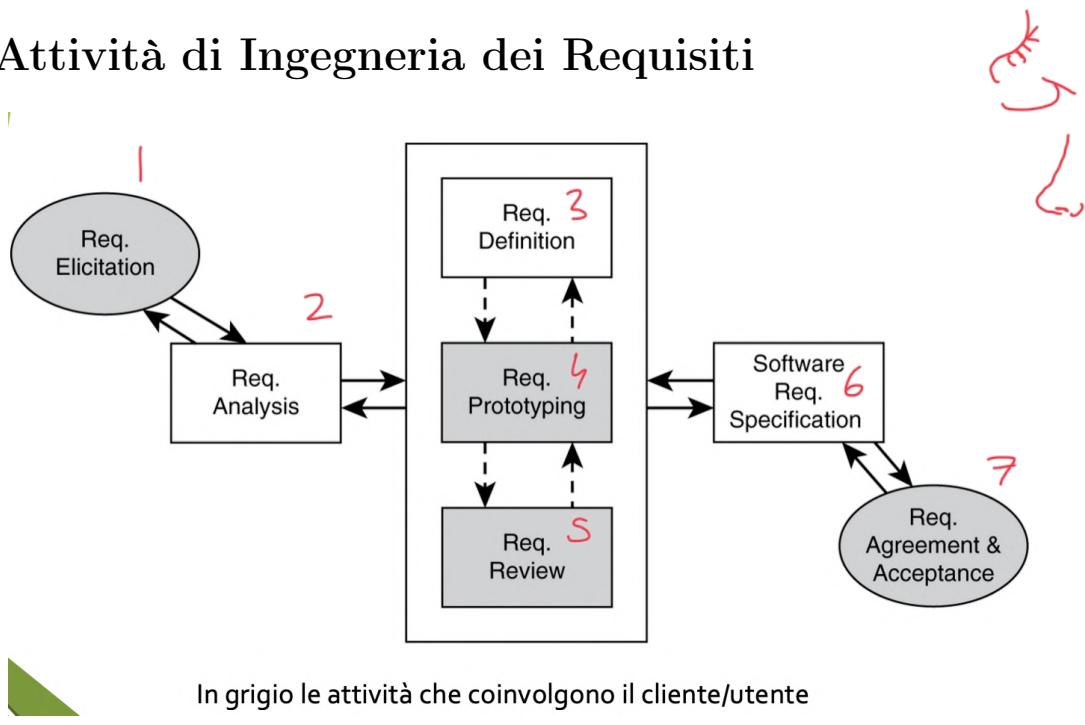
I **requisiti** sono le *descrizioni delle funzionalità e dei vincoli* del sistema:

- descrivono **cosa** dovrebbe fare il sistema software tramite una dichiarazione astratta di alto livello, fino ad una formale specifica matematica dettagliata
- *non descrivono come* esso dovrà essere realizzato

I requisiti possono avere una **duplice funzione**:

1. possono essere la **base per un'offerta** per un contratto: in questo caso devono essere *aperti all'interpretazione*
2. possono essere la **base del contratto** stesso: in questo caso devono essere *definiti in dettaglio*

3.1 Attività di Ingegneria dei Requisiti



Una delle **ragioni** del ***fallimento*** di un progetto software sono le specifiche dei requisiti incomplete, errate o mal comprese. Per questa ragione è necessario **documentare i requisiti**:

- sono necessari requisiti chiari per la *progettazione ed implementazione*
- serve a *creare casi e scenari di test*, soprattutto quando il team di test è separato dagli sviluppatori
- è necessario per creare *materiale formativo* per gli utenti e documenti per il *supporto* e la *manutenzione*



3.1.1 Elicitazione dei requisiti

I requisiti principali di un sistema software vengono solitamente *forniti dal cliente/utente* ma:

- gli utenti comprendono solo i requisiti relativi alle loro **specifiche mansioni** lavorative
- le **motivazioni e gli obiettivi aziendali** non sono sempre chiari ai singoli utenti
- i requisiti dichiarati dai utenti e dai clienti possono essere **contraddittori ed incompleti**

Dunque, molti requisiti vengono definiti dagli ingegneri del software, che devono **ottenere informazioni dai soggetti interessati, gli stakeholder** (*elicitazione*). Gli ingegneri del software che devono interagire con la direzione aziendale e gestire i requisiti sono talvolta chiamati ***analisti di business***.

Problematiche di elicitatione

- Gli stakeholder spesso *non sanno di cosa hanno bisogno* davvero
- esprimono i requisiti nella *loro terminologia*
- diversi stakeholder possono avere *esigenze contrastanti*
- i fattori organizzativi e politici dell'azienda possono *influenzare i requisiti* del sistema
- i *requisiti cambiano* durante il processo di analisi
- possono emergere *nuovi stakeholder* e l'ambiente di business può cambiare

Metodi di elicitatione

1. **Interviste**: utili per una *comprendizione generale* di ciò che fanno gli stakeholder e di **come potrebbero interagire** con il sistema.

Le interviste possono essere: **chiuse** (basate su un elenco predeterminato di domande) o **aperte** (vengono esplorate varie questioni con gli stakeholder).

2. **Analisi etnografica**: un esperto osserva accuratamente ed *analizza il modo in cui le persone lavorano*. Le persone non devono spiegare il loro lavoro, così si possono osservare *fattori sociale ed organizzativi* importanti.

I requisiti derivano dal modo in cui le persone lavorano realmente, quindi questo metodo è efficace per **comprendere i processi esistenti**, ma non è in grado di identificare *nuove funzionalità* da aggiungere al sistema.



3.1.2 Analisi dei requisiti

Dopo che i requisiti sono stati elicitati e raccolti, sono ancora un insieme *non organizzato di dati*.

Domanda d'Esame

L'**analisi dei requisiti** riguarda:

1. **Categorizzazione** dei requisiti: una categorizzazione di massima si può fare tra:

- **Requisiti funzionali**: dichiarazione sulle *funzionalità* che il sistema dovrebbe funzionare, su come dovrebbe reagire a particolari input e su come dovrebbe comportarsi in particolari situazioni.

Può indicare ciò che il sistema **non deve fare**.

- **Requisiti non funzionali**: *vincoli* sulle funzionalità offerte dal sistema (vincoli di tempo, processo di sviluppo, standard da adottare, ecc.).

Spesso si applicano al *sistema nel suo complesso* piuttosto che a singole caratteristiche o servizi.

Una categorizzazione più dettagliata può seguire le **sei dimensioni** dei requisiti (*5 funzionali, 1 non funzionale*):

- (a) **Funzionalità individuali**: i *servizi individuali che deve offrire il sistema*.

Alcuni *esempi* sono:

- *Funzionalità d'accesso*: il sistema deve consentire agli utenti di accedere utilizzando nome utente e password unici.
- *Gestione del conto*: gli utenti devono poter visualizzare il saldo del proprio conto, la cronologia delle transazioni e scaricare gli estratti conto
- *Stampa di certificati*: gli studenti devono poter ottenere un certificato che riporta il piano di studi corrente, ecc, e deve essere in formato PDF

- (b) **Business flow (scenari)**: descrivono le interazioni tra uno o più utenti ed il sistema per la realizzazione di uno specifico *processo aziendale*.

Ad esempio gestire l'*immatricolazione di uno studente*, la segreteria deve verificare il possesso dei requisiti, il sistema verifica il pagamento delle tasse, aggiornare il database, ecc.

Alcuni ingegneri del software partono dalla definizione delle **funzionalità individuali** per poi gestire il *flusso* (approccio bottom-top), oppure al contrario partendo dal **business flow** per tirare fuori le *funzionalità* (approccio top-down).

- (c) **Esigenze di dati ed informazioni**: descrivono i *dati che il sistema deve gestire*.

Ad esempio i *dati del profilo utente, sulle transazioni, ecc..*

- (d) **Interfacce utente**: si concentrano sulla *progettazione e sull'usabilità delle interfacce utente* del sistema.

Ad esempio la creazione di una *dashboard, design reattivo, messaggi d'errore*.

- (e) **Altre interfacce con sistemi/piattaforme esterne:** definiscono il modo in cui il sistema *interagisce con altri sistemi esterni*.
Ad esempio *integrazione con gateway di pagamento, uso di provider di autenticazione esterni, API*.
- (f) **Ulteriori vincoli come prestazioni, sicurezza ed affidabilità (unico requisito non funzionale):** qualificano i requisiti funzionali ed impongono *vincoli* al funzionamento del sistema.
Ad esempio prestazioni, sicurezza, affidabilità, scalabilità.

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

2. Definizione delle priorità dei requisiti: comporta la *stesura formale dei requisiti*. Si possono utilizzare *diversi approcci*, spesso in combinazione tra loro:

- *linguaggio naturale:* i requisiti sono scritti come un elenco di descrizioni naturali. Il *vantaggi* sono che è espressivo, intuitivo ed universale, i requisiti possono essere compresi da utenti e clienti.
Gli *svantaggi*, manca di chiarezza, troppo accorpamento dei requisiti
- *linguaggio naturale strutturato:* i requisiti sono descritti seguendo una struttura fissa (template).
Un esempio sono le *Tabelle Input-Process-Output (IPO)*

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

- *casi d'uso:* sono utilizzati per identificare le *funzionalità* principali, le *precondizioni* e le *post-condizioni* di tali funzionalità, il *flusso di attività* e le *condizioni di errore* ed i *flussi alternativi*.
- *diagrammi ed altre notazioni.*

La ricerca di **coerenza e completezza** dei requisiti: l'analisi dei requisiti deve anche garantire la coerenza e la completezza dei requisiti.



- *Coerenza*: non devono esserci contraddizioni o conflitti nelle descrizioni delle funzionalità del sistema.
- *Completezza*: i requisiti devono includere la descrizione di tutte le funzionalità necessarie di un sistema

A causa della complessità del sistema e dell'ambiente, spesso questo diventa impossibile. Un aspetto fondamentale è quindi la **tracciabilità dei requisiti**, la capacità di seguire il percorso dei requisiti durante il processo di sviluppo e di verificare che siano stati tutti *sviluppati e testati*.

Ogni requisito deve essere:



- **univocamente identificabile**
- **collegato alla fonte documentale** o alla **persona** che lo ha definito
- ogni **artefatto** di progettazione ed implementazione e test deve essere collegato al requisito che lo ha originato

La matrice di tracciabilità aiuta a tenere traccia dei requisiti durante il processo di sviluppo

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

Revisione dei requisiti: checklist

- *verificabilità* → ESANZI
- *comprendibilità* → È STATO COMPRENSO?
- *tracciabilità*
- *adattabilità*



Specifiche ed accettazione dei requisiti

Una volta elicitati, analizzati, definiti e revisionati, i requisiti devono essere inseriti in un *documento di Specifica dei Requisiti del Software (SRS)*. Il *livello di dettaglio* di questo documento dipende dalla dimensione e complessità del progetto, e della conoscenza ed esperienza degli sviluppatori nell'area tematica.

Il documento **SRS** deve essere *accettato dal cliente*:

- costituisce un *accordo tra cliente e fornitore*
- può costituire parte di un *contratto formale*
- serve per *verificare i risultati* intermedi e finali del progetto di sviluppo software
- serve come base per le *richieste di cambiamento ed integrazione*



3.2 Casi d'uso e UML

Un **Caso d'Uso** (*Use case*) rappresenta un insieme di *scenari di funzionamento* del sistema collegati tra loro per il fatto di essere finalizzati al raggiungimento di uno specifico *obiettivo* dell'utente. La *descrizione di un caso d'uso* consiste in:

- un **nome** univoco: una tipologia di utente che interagisce con il sistema. Gli attori partecipano ai casi d'uso.
- **Attori** partecipanti
- **Pre-condizioni**
- **Post-condizioni**
- Flusso di **eventi** normale
- Eventuali flussi di eventi **alternativi**

Perché sono utili i casi d'uso ?

1. *elicitare o descrivere i requisiti*: spesso la caratterizzazione dei casi d'uso porta alla scoperta di nuovi requisiti man mano che il progetto prende forma
2. *comunicare con i clienti*: la semplicità di notazione è un buon modo per gli sviluppatori di comunicare con i clienti
3. *scegliere i test cases*: la caratterizzazione di un caso d'uso può fornire indicazioni su come definire i casi di test che ne verificano l'implementazione

3.3 Introduzione a UML

UML **Unified Modeling Language**), linguaggio di *modellazione* per specificare, progettare e documentare sistemi software. *Unified Modeling Language*:

- *linguaggio*: serve ad esprimere concetti ed idee in maniera precisa e non ambigua
- *modellazione*: descrivere un sistema software ad un altro livello d'astrazione
- *unificato*: UML è diventato uno standard riconosciuto a livello globale

Caratteristiche di UML:

- è un **linguaggio grafico**, che definisce un insieme di diagramma e di notazioni grafiche
- particolarmente utile per la *progettazione orientata agli oggetti*
- *indipendente dal linguaggio di programmazione*
- **Non è** una metodologia né un processo di sviluppo software

Perché usare UML:

- *Standard aperto, con notazione grafica per specificare, visualizzare, costruire e documentare sistemi software.*
- *Migliora la comprensione/comunicazione* del prodotto da parte di clienti e sviluppatori. Può essere più *chiaro del linguaggio naturale* (impreciso) e del codice (tropo dettagliato). Può contribuire ad acquisire una *visione d'insieme* di un sistema.

Nome: AcquistoBiglietto

Attore partecipante: Passeggero

Precondizioni:

- Passeggero in piedi davanti al distributore di biglietti
- Passeggero ha denaro sufficiente per acquistare il biglietto

Postcondizioni:

- Passeggero ha il biglietto

Flusso di eventi:

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

Figura 3.1: Descrizione testuale ma strutturata del caso d'uso

Flusso di eventi alternativi:

3a. Passeggero annulla l'operazione

3a.1 L'esecuzione riprende dal passo 1.

3b. Trascorrono 60 secondi senza che Passeggero inserisca alcuna moneta

3b.1 Il distributore restituisce l'eventuale denaro già inserito

3b.2 L'esecuzione riprende dal passo 2.

Figura 3.2: Flussi di eventi alternativi

Come utilizzare UML ?

GLi sviluppatori utilizzano UML per **comunicare** alcuni aspetti di un sistema:

- *forward engineering*: si disegnano i diagrammi UML prima di scrivere il codice
- *reverse engineering*: si disegnano diagramma UML a partire da un codice esistente per aiutarne la comprensione
- **UML come schizzo (*sketch*):**
 - lo scopo è la *comunicazione di alcuni aspetti* piuttosto che la *completezza*, non si deve scrivere tutto il codice con tutti i dettagli
 - ci si concentra solo sulle *questioni importanti* (incompletezza intenzionale)
- **UML come disegno tecnico (*blueprint*):**
 - l'obiettivo è **costruire uno schema dettagliato** che un programmatore possa direttamente tradurre in codice
 - dovrebbe essere sufficientemente *completo* e chiarire tutte le decisioni progettuali
- **UML come linguaggio di programmazione - unused:**
 - gli sviluppatori disegnano diagrammi UML che vengono *tradotti automaticamente* in codice eseguibile
 - parzialmente supportato da alcuni IDE con plugin specifici
 - i diagramma devono essere *completi ed estremamente dettagliati*, rappresentando anche le decisioni più banali
 - gli sviluppatori disegnano diagrammi UML che vengono *tradotti automaticamente* in codice eseguibile.
 - Ha senso solo se la creazione dei diagrammi con il livello di dettaglio richiesto richiede *meno tempo* della scrittura in un linguaggio di programmazione convenzionale.

3.4 Diagrammi dei casi d'uso - Use Case Diagram

Un *diagramma dei casi d'uso* rappresenta una *vista d'insieme* dei casi d'uso di un sistema:

- evidenzia le *relazioni tra attori e casi d'uso*
- evidenzia le *relazioni tra casi d'uso collegati*

Un *errore comune* è sostituire un diagramma dei casi d'uso con la descrizione dei casi d'uso, è utile solo se è accompagnato dalle descrizioni vere e proprie.

Elementi di un diagramma dei casi d'uso

- la **relazione «extend»**: rappresenta *casi eccezionali* o raramente invocati.

I flussi di eventi eccezionali sono *eliminati dal flusso dell'evento principale* per motivi di chiarezza.

La direzione di una relazione «extend» è verso il caso d'uso esteso.

Quando usarla ?

B estende A quando, al verificarsi di determinate condizioni (quindi, non sempre), l'esecuzione di A comporta anche l'esecuzione di B

- la **relazione «include»**: indica che un caso d'uso è "incluso" come passo di un altro caso d'uso (nella direzione del verbo).

La fattorizzazione ha lo scopo di:

1. *semplificare la descrizione* del caso d'uso principale
2. *riutilizzare il caso d'uso* incluso

la direzione di una relazione «include» è verso il caso d'uso utilizzato (nella direzione del verbo).

Quando usarla ?

A include B quando l'esecuzione di A comporta sempre anche l'esecuzione di B.

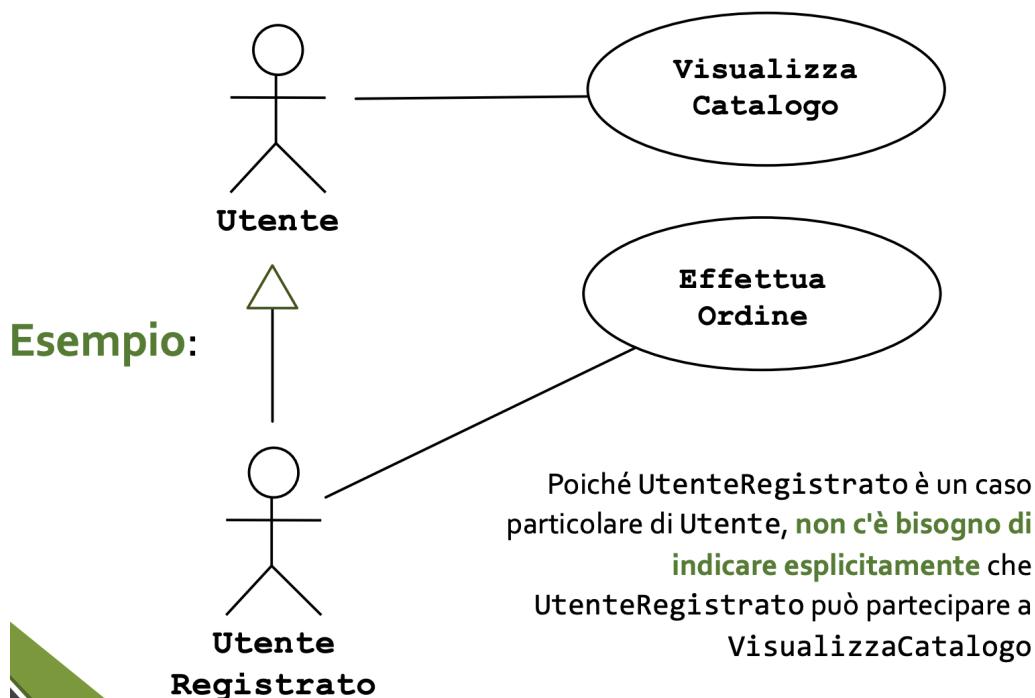
- la **relazione di specializzazione** (detta anche, se "letta al contrario", di *generalizzazione*): indica che un elemento è un caso particolare di un altro elemento.

Nei diagrammi dei casi d'uso è **usata principalmente per gli attori**.

La direzione di una relazione di specializzazione è *verso l'elemento più generale*.

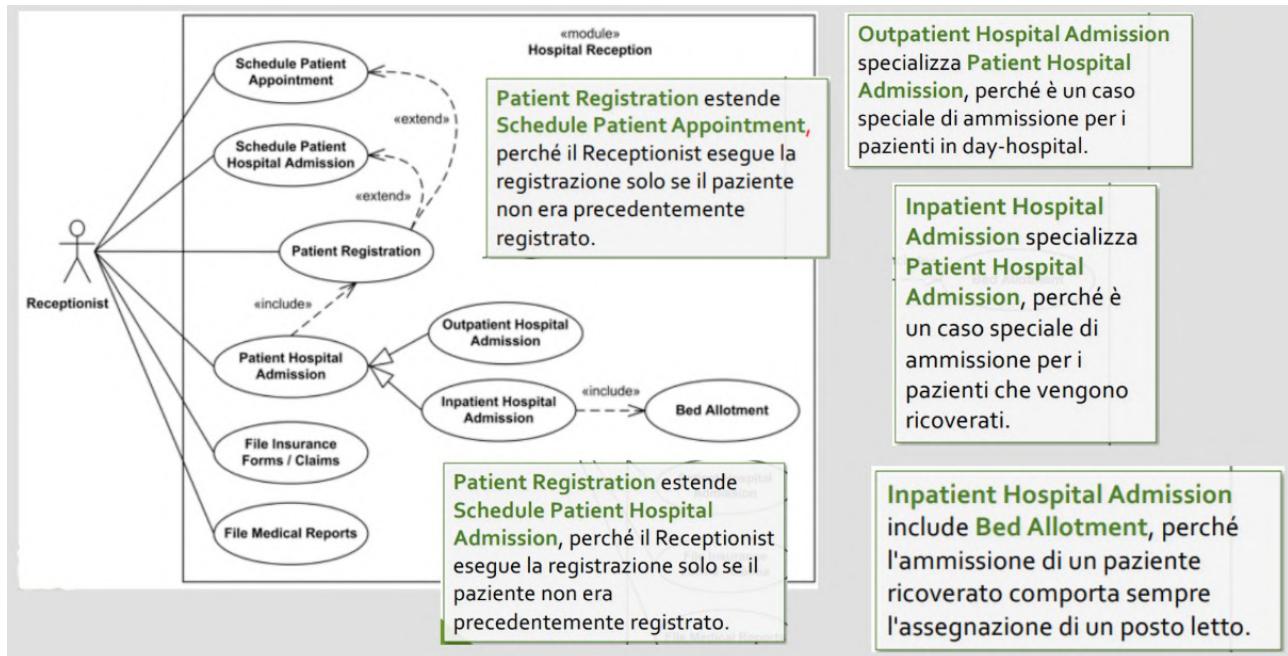
Quando usarla ?

B specializza A quando B si può considerare un caso particolare di A; l'esecuzione di B sostituisce in determinate situazioni l'esecuzione di A.



Riepilogo:

- i *casi d'uso* rappresentano il comportamento esterno di un sistema
- i diagrammi dei casi d'uso sono utili *come indice dei casi d'uso*
- le *descrizioni dei casi d'uso* forniscono le informazioni più importanti, non i diagrammi



3.5 Specifica di un Componente Software

Un **componente software** è un'unità di software che può essere sviluppata e testata singolarmente, e che può essere utilizzata da altre parti di un sistema software per risolvere uno specifico problema.

- aiuta a dividere un sistema completto in parti più semplici (**modularità**)
- un componente può essere utilizzato in più sistemi software (**riusabilità**)
- un componente può essere realizzato da un'azienda diversa da quella che lo ha sviluppato

Per *progettare* e *realizzare* un singolo componente software, la descrizione di **cosa** deve fare il componente (*requisiti funzionali*) e di quali **vincoli** deve rispettare (*requisiti non funzionale*) ci viene fornita attraverso una **specific**a del componente software.

Rispetto ai requisiti di un sistema software, la **specific**a di un componente è:

1. più formale e precisa
2. normalmente espressa in un linguaggio tecnico
3. più dettagliata

Le **informazioni** che devono far parte della specifica di un componente software sono:

- **tutte** le informazioni che sono necessarie agli sviluppatori che dovranno **usare** il componente software

- **nessuna** delle informazioni che servono soltanto agli sviluppatori che dovranno **realizzare** il componente software

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

Alcune delle informazioni della specifica sono ricavabili dalla **dichiarazione del componente** (es. *prototipo della funzione*). Ma la maggior parte delle informazioni non lo è.

Cosa *dobbiamo inserire* nella **documentazione** del componente?

- *Tutte le informazioni della specifica che non sono direttamente ricavabili dalla dichiarazione del componente*
- è una buona idea inserire queste informazioni in un **commento**
- esistono tecniche per **estrarre automaticamente** la **documentazione dai commenti**

3.5.1 La metafora del contratto

Un contratto è un *accordo* tra cliente e fornitore.

Caratteristiche di un contratto:

- Ciascuna parte si aspetta alcuni **benefici** del contratto
- Ciascuna parte è disposta a sostenere alcuni **obblighi** per ottenerli
- il beneficio del cliente è un obbligo per il fornitore e viceversa
- il contratto **documenta** questi benefici ed obblighi

Vantaggi del contratto:

1. **Protezione del cliente:** definisce chiaramente cosa deve fare il cliente per ricevere il servizio concordato, il fornitore non può richiedergli di **più** o dagli di **meno**.
2. **Protezione del fornitore:** definisce chiaramente il servizio considerato accettabile, il cliente non può richiedergli di **più** o dagli di **meno**.

Se una delle due parti non rispetta i propri **obblighi**, perde il diritto ai relativi *benefici* e non vi è nessuna clausola nascosta.

Design by Contract

Il **Design by Contract** (*D_bC*) è una metodologia per la specifica di componenti software in cui le caratteristiche dei componenti sono descritte immaginando di definire una sorta di "contratto" che regolamente la collaborazione tra due parti.

Ruoli della DbC:

- *Client:* la parte software che **usa** i servizi offerti da un componente software
- *Provider:* il componente software, che **offre** dei servizi che sono descritti nella specifica di cui stiamo parlando.

Impostando la specifica come se fosse un contratto tra Client e Provider, si identificano chiaramente quali sono gli obblighi e le responsabilità delle due parti

- Ciò rende il lavoro più chiaro sia per chi dovrà realizzare il componente software che per chi dovrà usarlo
- La metafora del contratto inoltre ci aiuterà a capire meglio alcuni aspetti della progettazione e della programmazione orientata agli oggetti

Precondizioni, postcondizioni, invarianti

- **Servizi:** la specifica di un componente software prevede una o più operazioni che il client può richiedere al componente.

Ad esempio, ogni metodo ***pubblico*** di una classe Java è un'operazione prevista dal contratto della classe.

I servizi possono essere specificati tramite la definizione di *Precondizioni* e *Postcondizioni*.

- **Precondizioni:** sono condizioni che devono essere verificate nel momento in cui viene effettuata la richiesta di un'operazione al componente software.

- Le precondizioni sono **obbligazioni** per il *client*, che ha la responsabilità di garantire che esse siano soddisfatte quando richiede l'operazione
- Le precondizioni sono **benefici** per il *provider*, che può sfruttare nella sua implementazione per la realizzazione della funzionalità richiesta

Le precondizioni riguardano:

- condizioni, limitazioni o vincoli sui **parametri** che vengono forniti come input
- condizioni, limitazioni o vincoli sullo **stato** di altre variabili o risorse

- **Postcondizioni:** sono condizioni che, a **patto che le precondizioni siano soddisfatte**, devono essere verificate nel momento in cui il componente completa l'operazione richiesta.

- le precondizioni sono **benefici** per il *client*, che può sfruttarle per raggiungere i suoi scopi (il client richiede l'operazione proprio perché ha bisogno che sia realizzata)
- le precondizioni sono **obbligazioni** per il *provider*, che ha la responsabilità di garantire che esse siano soddisfatte quando completa l'operazione

Le postcondizioni riguardano:

- condizioni, limitazioni o vincoli sul **valore di ritorno** dell'operazione
- condizioni, limitazioni o vincoli sullo **stato** di altre variabili o risorse che sono accessibili dal componente software
- eventuali **effetti collaterali** (*side effects*): ad esempio una *printf*

- **Invarianti:** quando un componente software contiene una o più **strutture dati**, generalmente ogni operazione prevista dal contratto prevede che al momento della **richiesta** le **strutture dati siano in uno stato "coerente"**.

Analogamente, il *contratto* prevede che ogni operazione mantenga le strutture dati in uno stato "coerente".

Le condizioni che caratterizzano uno stato "coerente" delle strutture dati vengono dette **invarianti**:

- le invarianti possono essere considerate allo stesso tempo come *precondizioni e postcondizioni* presenti in ogni operazione prevista dal contratto. Devono essere quindi valide prima e dopo ma non è necessario durante.
- le invarianti quindi sono obblighi sia per il client che per il provider
- nella descrizione del contratto le invarianti vengono specificate una sola volta, quindi sono *precondizioni e postcondizione* presenti implicitamente.

- **Inizializzazioni:** hanno il compito di portare le strutture dati in uno stato iniziale che soddisfi tutte le invarianti.

Le operazioni di inizializzazione sono speciali, perché **non hanno le invarianti come precondizioni ma solo come postcondizioni**

Precondizioni e postcondizioni, evita il **defensive programming**, quando i controlli sulle condizioni vengono effettuate sia dal chiamante che dalla funzione.

Asserzioni

Un'asserzione è un'**espressione booleana** che il programmatore si *aspetta sia vera in un punto specifico*.

- Aiuta nella **convalida delle ipotesi**, formalizzando invarianti, precondizioni e postcondizioni
- forniscono una chiara **documentazione** delle condizioni e dei vincoli previsti
- **Non** deve **influire sulla logica** del programma, ma serve a facilitare la *comprendizione* del comportamento del codice
- Se un'asserzione fallisce, viene lanciata un'**eccezione**
- Utilizzata per il **debug** per rilevare violazioni delle condizioni previste in fase di esecuzione. Sono implementate in fase di sviluppo e progettazione, e possono essere disabilitate nelle versioni in produzione.

Nell'**implementazione** di un componente software le asserzioni possono essere usate come "rete di sicurezza" per aiutare gli sviluppatori a controllare se il contratto viene rispettato.

- **All'inizio di un'operazione:** si asseriscono le *precondizioni* e le *invarianti* per controllare che il client abbia svolto correttamente la sua parte del contratto
- **Al termine di un'operazione:** si asseriscono le *postcondizione* e le *invarianti* per controllare che il provider abbia svolto correttamente la sua parte del contratto.

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

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

3.6 Specifiche, Contratti e Documentazione

Asserzioni in Java

Per impostazione predefinita, le *asserzioni sono disabilitate in Java*. Per abilitarle è necessario eseguire il programma con il flag:

`-ea` o `-enableassertions`

Sintassi di base:

```
assert condition : "Error message";
```

3.6.1 Contratti ed encapsulamento

Domanda: i metodi e le strutture dati private *fanno parte del "contratto" della classe?*

Risposta: Il contratto regolamenta i **rapporti tra la classe ed i suoi client**, poiché i client non possono accedere ai membri privati della classe, quindi **non fanno parte del contratto**.

Tuttavia, la metafora del contratto **può essere usata**, oltre che descrivere la specifica di una classe, anche per aiutare a *garantire la correttezza della sua realizzazione*.

La classe deve rispettare **due contratti**:

1. un contratto "**pubblico**" verso i *clienti* della classe, che deve essere definito al momento della specifica della classe
2. un contratto "**privato**", che serve a garantire che le varie parti della classe lavorino correttamente tra loro, che deve essere definito al momento della **progettazione in dettaglio**

Se le **invarianti riguardano solo la parte privata** della classe, il meccanismo dell'encapsulamento aiuta a *garantire che un client non possa erroneamente violare le invarianti*.

- se i *costruttori* garantiscono la verifica delle invarianti al momento della creazione dell'oggetto
- ogni *metodo pubblico* garantisce che se le invarianti erano soddisfatte all'inizio del metodo, saranno soddisfatte alla fine
- allora, il *client non può portare l'oggetto in uno stato che non soddisfa le invarianti!!*
- in questo caso, il rispetto delle invarianti diventa una responsabilità esclusivamente del provider.

Domanda: il "*contratto privato*" di una classe **deve essere documentato?**

Risposta: sì per l'utilizzo *interno*. Una documentazione separata da quella esterna, resa disponibile agli sviluppatori che devono usare la classe.

Modifica di un componente software

La metafora del contratto può essere utile a capire l'*impatto di una modifica di un componente software*. Possiamo domandarci *come la modifica cambia i benefici e gli obblighi per il client*:

- *gli obblighi dei client restano uguali o diminuiscono*: la modifica non ha impatto sul client e quindi ha un *impatto contenuto*
- *gli obblighi dei client aumentano*: oltre al codice del componente, bisogna verificare e forse modificare anche il codice del client. La modifica è *potezialmente rischiosa*

Domanda: quando si verifica che gli *obblighi del client aumentano*?

Risposta: quando le **precondizioni** di un'operazione diventano più "restrittive":

- viene aggiunta una precondizione
- una precondizione viene modificata, in modo che la nuova versione possa essere falsa in alcune situazioni in cui la vecchia versione era vera

Domanda: quando si verifica che i benefici del client diminuiscono?

Risposta: quando viene **eliminata una delle operazioni previste dal contratto** oppure quando le **postcondizioni** di un'operazione diventano *meno "restrittive"* (rimossa o modificata in modo che la nuova versione possa essere vera in alcune condizioni in cui prima era falsa).

Quanti più sono gli elementi di una classe che rientrano nel suo "*contratto pubblico*" tanto maggiore sono le **probabilità che una modifica abbia un impatto elevato**.

Per questo è una buona idea **Mantenere nel contratto pubblico solo gli elementi strettamente necessari** (vantaggio dell'incapsulamento).

3.6.2 Documentazione della specifica

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

- Gli sviluppatori che **usano** il componente (gli sviluppatori del client) potrebbero **non avere accesso al codice sorgente** del componente software
- gli sviluppatori che **realizzano** il componente dovranno garantire che la progettazione e l'implementazione del componente **rispetti quello che dice la specifica**
- gli sviluppatori che devono **manutenere** il componente software, per loro è importante che il codice sorgente e la documentazione siano "**allineati**"

Un modo efficace per favorire l'allineamento tra documentazione e codice sorgente è di inserire la **descrizione della specifica nel codice sorgente** (*commenti di documentazione*) e usare degli **strumenti automatici** (*Doxxygen*) per estrarre dal codice sorgente un *documento* che descriva la specifica.

Cosa inserire nei commenti?

Nei commenti della documentazione è **inutile** aggiungere le informazioni che possono essere ricavate dai prototipi delle operazioni. Invece, è importante inserire le *informazioni sul contratto* che **non possono essere ricavate dal prototipo**.

1. Cosa fa la funzione
2. Precondizioni/Postcondizioni

Modi dei parametri

Bisogna fare differenza tra le **modalità di comunicazione** tra il *chiamante* e la *funzione* (direzioni differenti).

- parametri in **input**: informazione *definita* dal *chiamante* prima della chiamata e viene *usata* dalla funzione
- parametri in **output**: informazione definita dalla funzione, e *usata* dal *chiamante* dopo la chiamata

Questa differenza fa **parte del contratto** della funzione, ma **non** fa parte *della sintassi del linguaggio*, perciò deve essere **evidenziata nei commenti di documentazione**.

Un modo efficace di descrivere la comunicazione, è indicare la "**direzione**" in cui viaggiano le informazioni:

- **Input** (*In*)
- **Ouput** (*Out*)
- **Input/Output** (*In/Out*)

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

Documentazione automatizzata

Alcuni strumenti:

- **Doxxygen**: supporta più linguaggi
- **Javadoc**: specifico per *Java*
- **Shinx**: specifico per *Python*

Capitolo 4

Progettazione del software

4.1 Introduzione alla progettazione del software

I **requisiti** affrontano il "*Cosa?*". Cosa *dovrebbe fare* il sistema, quali sono i *vincoli*.

La **progettazione** (*design*) affronta il "*Come?*"?

- Come il sistema viene **scomposto** in componenti
- Come questi componenti si **interfacciano** ed **interagiscono**
- Come **funziona** ogni singolo componente

Lo scopo della **progettazione** è prendere un insieme di decisioni su *come verrà costruito il sistema* che aiutino a garantire:

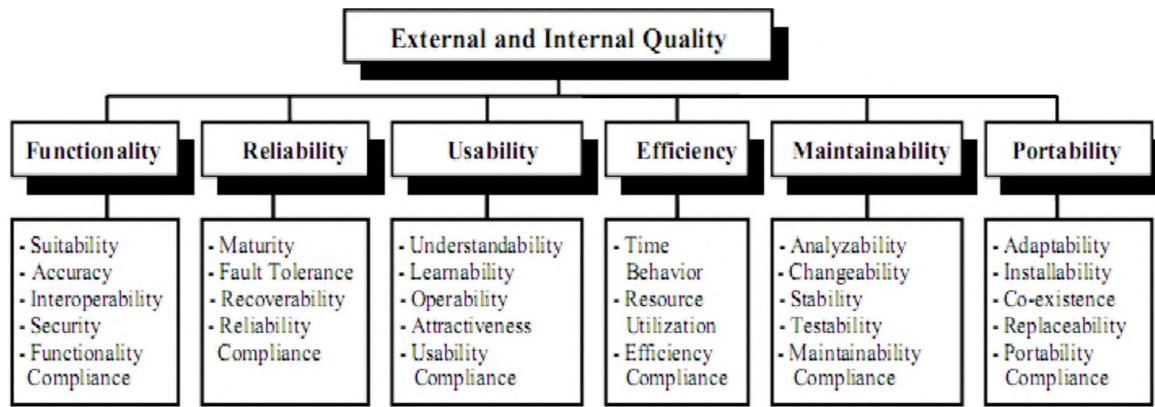
- che il sistema realizzato soddisfi i **requisiti funzionali e non funzionali** identificati
- che il sistema abbia delle **caratteristiche di "qualità"**: anche se non specificate come requisiti, hanno un impatto sui **costi** per la realizzazione, manutenzione ed evoluzione del sistema e sui **rischi** che il sistema possa non riuscire a soddisfare i suoi requisiti

Un **Attributo di Qualità** (*Quality Attribute, QA*) è una *proprietà misurabile o testabile* di un sistema che ha un impatto sulla capacità del sistema di soddisfare le esigenze degli stakeholder.

- **Attributi di qualità esterni**: proprietà *visibili* agli utilizzatori del sistema, ad esempio la *Performance*.
Spesso questi attributi di qualità esterni diventano **requisiti non funzionali**
- **Attributi di qualità interni**: proprietà visibili solo agli sviluppatori del sistema, ad esempio la *Manutenibilità*

Domanda: se gli attributi di qualità interno non sono visibili agli utilizzatori del sistema, perchè dobbiamo preoccuparcene ?

Risposta: perchè la capacità di soddisfare le esigenze degli utilizzatori dipende anche dagli attributi di qualità interni.



Alcuni **QA importanti**:

1. QA esterni:

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

2. QA interni:

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

4.2 Fasi della progettazione

Le attività di progettazione sono solitamente suddivise in:

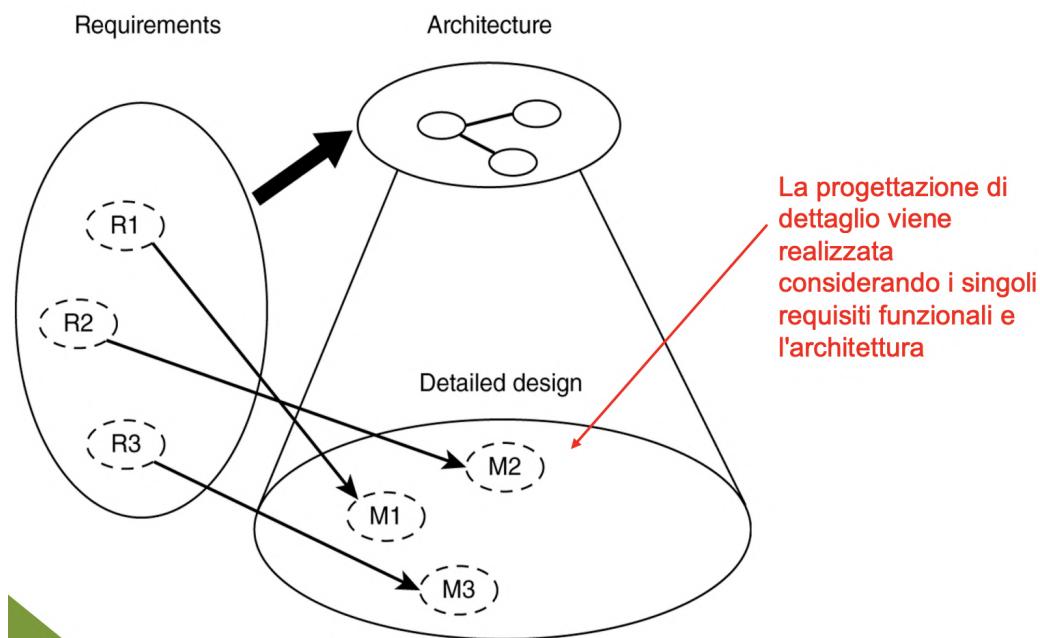
- Fase di **progettazione architettonale**:

- viene definita una *vista di alto livello* del sistema
- i *componenti* principali sono identificati
- le principali *relazioni* tra i componenti sono definite
- ***guidata dai requisiti (principalmente non funzionali) e dai QA***

- fase di **progettazione dettagliata**:

- i componenti vengono **scomposti** ad un livello di dettaglio molto più fine
- **Guidata dai requisiti funzionali e dall'architettura, oltre che dai QA**

L'**architettura** viene definita considerando una vista d'insieme dei requisiti. La **progettazione** di dettaglio viene realizzata *considerando i singoli requisiti funzionali e l'architettura*.



Fasi della progettazione:

- **nei piccoli sistemi**: una definizione **esplicita** dell'architettura è talvolta assente. Si passa direttamente alla **progettazione di dettaglio**.
- **In diversi processi tradizionali**: il progetto del sistema è il più **dettagliato** possibile. I programmati si limitano a **tradurre il progetto in codice**
- **Negli approcci più moderni**: la progettazione si ferma ad un **livello più alto di astrazione**. Il programmatore affronta direttamente la progettazione di dettaglio senza produrre documentazione specifica (*metodi agili*).

Output della progettazione Le **decisioni** prese durante la fase di progettazione possono essere rappresentate attraverso un insieme di **modelli** del sistema:

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

4.3 Progettazione architetturale

Insieme di **modelli** che descrivono il software da sviluppare, tra cui:

- i principali **elementi** del software (moduli, componenti)
- le loro **proprietà** visibili all'esterno (interfacce)
- le **relazioni** tra loro (interazioni)

Ogni sistema software ha un'architettura (anche implicita).

Un sistema può essere descritto da *più modelli*: molti modi di organizzare gli elementi, a seconda della *prospettiva*.

Viste architettoniche

Il concetto di viste architettoniche si riferisce a un modo di rappresentare l'architettura di un sistema software attraverso diverse prospettive o "viste". Queste diverse viste permettono di descrivere aspetti differenti del sistema, focalizzandosi su particolari dettagli che sono importanti per i vari stakeholder (ad esempio, sviluppatori, utenti, amministratori di sistema). Il modello delle viste architettoniche proposto da **Kruchten** (Architectural Views) nel 1995, noto anche come il modello 4+1, distingue le seguenti viste:

- **Vista logica:** definisce i moduli del software, i componenti e le loro relazioni (*gerarchia dei moduli, diagrammi delle classi*). Lista Statica.
- **vista di processo:** come i componenti del sistema interagiscono e come sono distribuiti nei diversi processi (*diagrammi di sequenza, diagrammi di attività*) Lista Dinamica.
- **Vista di sviluppo:** descrivere l'organizzazione dei team di sviluppo e come viene gestito il processo di sviluppo (*script di compilazione, controllo delle versioni*)
- **Vista fisica:** informazioni sui componenti hardware, sulla topologia della rete e sulla distribuzione dei componenti software tra i server (*diagrammi di distribuzione*)
- il '+1' descrive come il *sistema risponde a specifici casi d'uso*. Funzione come una vista trasversale, perché verifica che tutte le altre viste siano coerenti e che il sistema funzioni come previsto nei casi d'uso importanti.

Per Bass, Clements e Kazman (2021), esistono queste altre viste:

- **Viste modulari:** rappresentano la decomposizione del sistema in moduli e sottosistemi ed il modo in cui i moduli dipendono l'uno dall'altro
- **Viste di runtime:** indica come i moduli o i processi in esecuzione comunicano tra loro
- **Viste di allocazione:** mappano i moduli software ad altri sistemi (strutture hardware, file sorgente, persona o team responsabile)

Conoscenza meta-architettonica: diverse architetture hanno caratteristiche comuni.

Gli ingegneri del software hanno definito: *stili o pattern architettonici* (punto di partenza per le *attività di progettazione architettonica*), e *tattiche architettoniche*. Utilizzabili come **punto di partenza** per la definizione dell'architettura del sistema, e il **meccanismo di comunicazione** efficace per fornire un'idea rapida della struttura di alto livello di un sistema.

Esempio: Pipeline.

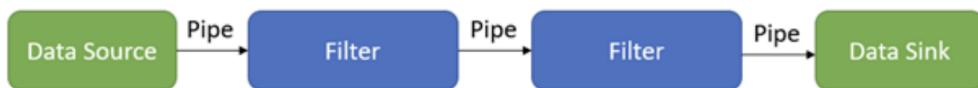
Architettura incentrata sui dati, strutturata sul modo *in cui i dati fluiscono attraverso l'applicazione*.

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

Costituisce una **serie di processi** collegati da "*pipes*": l'output di un processo è l'input del processo successivo (*cat books.txt / sort / uniq*).

Componenti:

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



Pipeline: *Benefici*

- ogni *filtro* incapsula una funzionalità specifica, semplificano lo sviluppo, il test e la manutenzione dei componenti.

In questo modo è possibile sostituire o estendere i filtri senza influire sul sistema

- una volta creato un filtro, è possibile **applicarlo a più pipeline** o persino a sistemi diversi
- è possibile **riorganizzare i filtri** o introdurne di nuovi per modificare il comportamento del sistema o aggiungere nuove funzionalità *senza una riconfigurazione del sistema*
- i filtri possono essere *eseguiti in parallelo*, sfruttando i processori multi-core e l'elaborazione distribuita.

Ovviamente, favorire la modularità, diminuisce l'efficienza. Magari con uno stile più parallelo si favorisce l'esecuzione di più operazioni insieme.

Altri stili architetturali:

- Event-Driven
- Client-Server
- Model-View-Controller (MVC)
- Startificato
- Orientato ai servizi (SOA)

Tattiche architetturali

Risolvono **problemi architetturali specifici**, che riguardano più componenti, senza influire sulla struttura complessiva.

Esempio:

- vogliamo evitare che un *componente possa bloccarsi* senza che il sistema lo rilevi
- introduciamo un nuovo componente responsabile del **rilevamento dei guasti**

Tattica:

1. **Heartbeat**: ogni componente invia un messaggio al rilevatore di guasti ad intervalli prestabiliti
2. **Ping/Echo**: il rilevatore di guasti invia ad intervalli prestabiliti un messaggio a tutti gli altri componenti ed attende una risposta
3. **Caching**: memorizzazione e riutilizzo dei dati a cui si accede di frequente per ridurre la latenza ed aumentare le prestazioni
4. **Bilanciamento del carico**: distribuisce le richieste su più server o risorse
5. **Compressione**: riduce le dimensioni dei dati
6. **Crittografia**: protegge dati sensibili
7. **Connection Pooling**: riutilizzare le connessioni al database per evitare l'overhead della creazione di nuove connessioni per ogni richiesta

Achitettura e QA

Le decisioni architetturali possono avere un impatto significativo sugli attributi di qualità di un sistema. Ad esempio:

1. **Efficienza**: può essere aumentata riducendo la necessità di trasferire dati tra componenti diversi
2. **Security**: può essere aumentata avendo componenti che eseguono controlli sulle informazioni scambiare con il mondo esterno
3. **Disponibilità**: può essere aumentata introducendo componenti ridondanti nel sistema
4. **Manutenibilità**: può essere aumentata se ciascun componente deve svolgere poche operazioni e non è collegato a molti altri componenti

Sfortunatamente, spesso questi attributi di qualità sono in **confitto tra di loro**. Ad esempio:

- avere *componenti di piccole dimensioni* aumenta la manutenibilità, ma potrebbe ridurre l'efficienza
- un **unico punto di comunicazione** con il mondo esterno aumenta la security, ma potrebbe ridurre la disponibilità

Quindi, bisogna scegliere un **compromesso accettabile** tra le varie esigenze.

4.4 Progettazione in dettaglio

- Nella progettazione **architetturale**, il sistema da realizzare è *suddiviso in componenti*
- Nella progettazione **di dettaglio**, questi componenti possono essere suddivisi in *unità più piccole*
- La suddivisione può essere **ripetuta a più livelli**, suddividendo le unità precedentemente ottenute fino ad arrivare a **unità sufficientemente semplici da realizzare**

Apporcci alla *Scomposizione*

1. *Scomposizione funzionale* (o procedurale): si basa sulle **operazioni/funzionalità** che devono essere realizzate, ad ogni livello di scomposizione, operazioni complesse sono suddivise in operazioni più semplici
2. *Scomposizione object-oriented*: si basa sulla suddivisione in **oggetti** che descrivono una **parte del dominio del problema**; ad ogni livello di scomposizione, gli oggetti/classi del livello precedente vengono resi ancora più piccoli

I due approcci possono essere **combinati**. Ad esempio:

- al livello più alto si esegue una scomposizione in componenti con l'**approccio funzionale**, mentre ai livelli successivi ciascun componente è scomposto in classi
- oppure, al contrario, all'interno di una classe, le *operazioni della classe* sono divide in **metodi** con *approccio funzionale*

Scomposizione in moduli. Le informazioni da definire sono:

- **Dipendenze** tra i moduli individuati
- **Interface**
- **Aspetti dinamici**: comportamento dei moduli ed interazioni tra essi

Modi di scomporre un sistem

Dato un sistema da realizzare, ci sono **molti modi diversi di scomporlo**, anche usando lo stesso approccio (funzionale o object-oriented).

Due caratteristiche che aiutano a valutare la qualità di una scomposizione:

1. **Coesione**: misura quanto le parti che sono incluse nello stesso modulo sono *legate tra di loro*. Caratteristica *intra-modulo*
2. **Accoppiamento**: misura il grado di *interdipendenza tra moduli diversi*. Caratteristica *inter-modulo*

Il generale, una buona decomposizione cerca di ottenere **alta coesione e basso accoppiamento**.

Livelli di coesione

I livelli **più alti** sono i *"migliori"*:

1. **Funzionale:** il modulo include funzionalità che lavorano insieme per **realizzare un singolo compito ben definito**.

Esempio: un modulo contiene le operazioni fondamentali su una struttura dati.

```
public class Stack{
    public boolean isEmpty;
    public void push(int x);
    public int pop;
}
```

2. **Sequenziale:** il modulo include *funzionalità che lavorano insieme, e gli output sono usati come input da un'altra*.

Esempio: un modulo contiene l'operazione che carica un file audio e ne riproduce il contenuto

```
public class AudioPlayer{
    public Clip loadClip(String fileName);
    public void playAudio(Clip c);
}
```

3. **Comunicazionale:** il modulo include funzionalità che *lavorano sugli stessi dati*.

Esempio: un modulo contiene l'operazione che calcola la media ponderata di un elenco di esami sostenuti e l'operazione che verifica se l'elenco di esami rispetta le propedeuticità

```
public class CarrieraStudente{
    public boolean rispettaProped;
    public double mediaPonderata;
}
```

4. **Procedurale:** il modulo include funzionalità che vengono spesso **usate insieme come parte di una stessa operazione**.

Esempio: un modulo contiene l'operazione che controlla l'esistenza di un file di configurazione e quella che ne carica il contenuto

```
public class ConfigurazioneFile{
    public boolean esisteFile;
    public Configurazione caricaFile;
}
```

5. **Temporale:** il modulo include funzionalità che vengono usate *nella stessa fase di esecuzione del programma*.

Esempio: un modulo contiene tutte le operazioni svolte all'inizio del programma

```
public class Inizio{
    public Partita creaPartita;
    public void svuotaCache;
}
```

6. **Logica**: il modulo include funzionalità che **svolgono operazioni simili**, ma in contesti o su *dati non correlati tra loro*.

Esempio: un modulo contiene tutte le operazioni di stampa

```
public class Stampa{
    public void stampaArray;
    public void stampaStudente;
}
```

7. **Coincidentale**: il modulo include funzionalità che **non hanno nessun collegamento tra loro**.

Esempio: le funzionalità stanno insieme solo perché implementate dallo stesso autore

```
public class Giovanni{
    public void ordinaArray(int a []);
    public double eqSecondoGrad(double a, double b);
}
```

Livelli di accoppiamento

I livelli **più bassi** sono i "*migliori*":

1. **Per contenuti** (*Content coupling*): un modulo accede agli **aspetti implementativi dell'altro**.

Esempio: un modulo accede direttamente alla struttura dati di un altro modulo, *senza usare getter e setter*.

2. **Per aree comuni** (*Common coupling*): due moduli si scambiano informazioni attraverso un'area dati comune.

Esempio: utilizzo di una variabile globale, una variabile **static** si comporta come una variabile globale!

3. **Per controllo** (*Control coupling*): un modulo passa *informazioni* di "*controllo*" ad un altro, in base alle quali viene **cambiata la logica di funzionamento** di quest'ultimo.

Esempio: il flusso di esecuzione di un *metodo* dipende da un flag passato da un'altra classe.

4. **Per timbro** (*Stamp coupling*): un modulo passa una struttura dati ad un altro, la struttura passata contiene anche **informazioni che non sono necessarie all'altro modulo**.

5. **Per dati** (*Data coupling*): un modulo passa all'altro solo le *informazioni strettamente necessarie per il suo funzionamento*.

6. **Nessuno** (*No Direct coupling*): due moduli non dipendono (*direttamente*) l'uno dall'altro. E' estremamente improbabile che un modulo non sia accoppiato con nessun altro modo, tuttavia è importante ridurre il numero di moduli con cui ciascun modulo è accoppiato.

4.4.1 Diagramma dei package e dei componenti

Diagramma dei package

Il diagramma dei package è un *diagramma UML* che può essere usato per descrivere la scomposizione in moduli di un sistema (a diversi livelli di dettaglio) e le dipendenza tra i moduli.

Un **package** (in UML) è un costrutto che rappresenta un gruppi di altri elementi (es, classi).

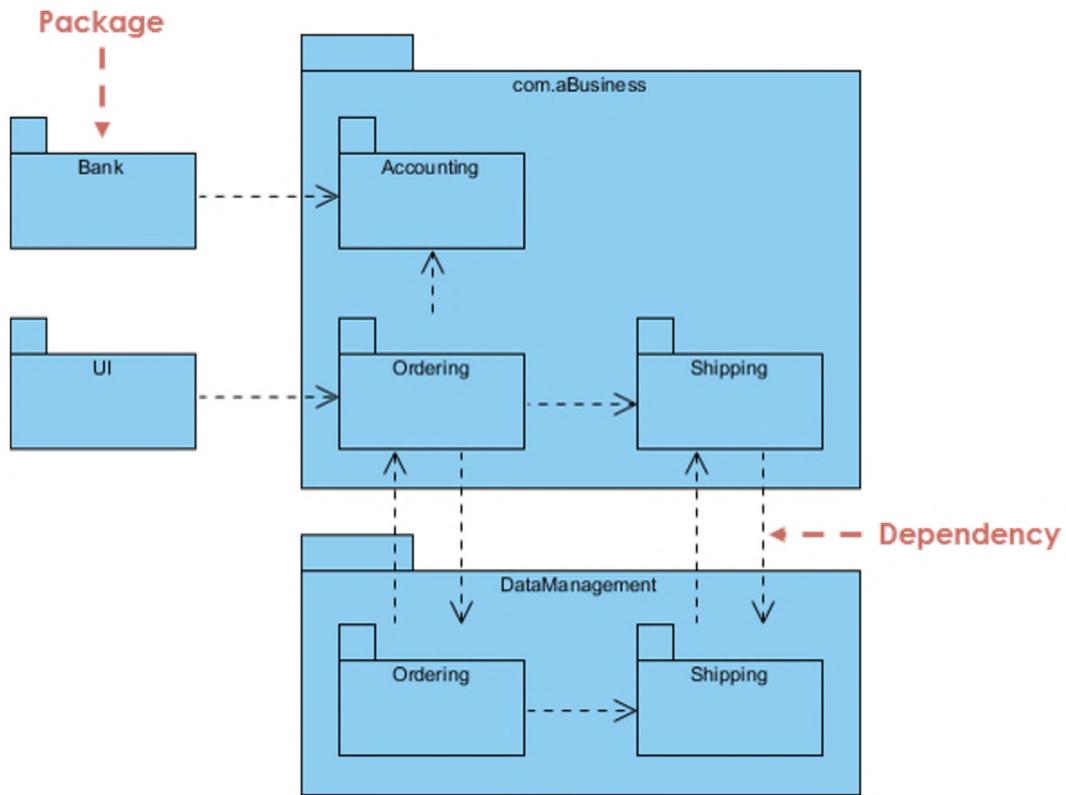
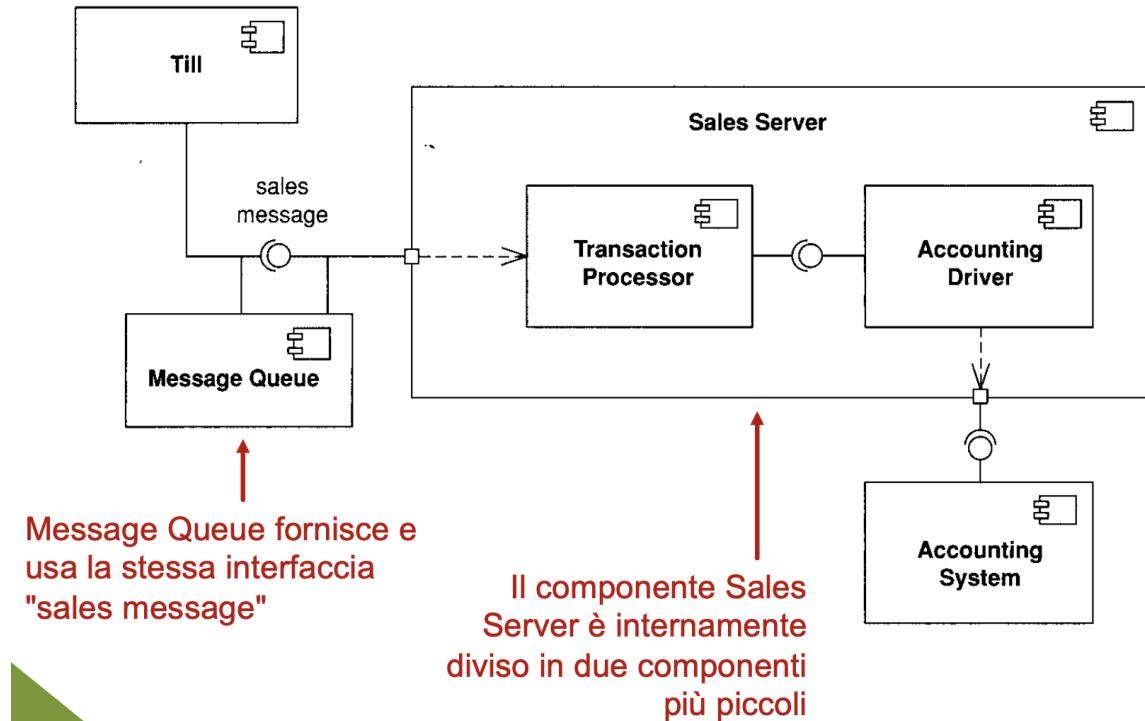


Diagramma dei componenti

Un diagramma UML usato per indicare la **suddivisione di un sistema in componenti software**, specialmente nel caso in cui i componenti possano essere *realizzati o acquistati separatamente*.

Può essere usato per **identificare esplicitamente le interfacce** che ciascun componente usa e quelle che mette a disposizione.

- un'interfaccia rappresenta un **servizio** o un **insieme di servizi collegati**
- L'idea è che un **componente** possa essere *sostituito da un altro componente che offre le stesse interfacce*.



Le **interfacce usate e fornite** da un componente sono rappresentate con i simboli, pallino aperto / pieno.

4.5 Esercizio sulla Progettazione - Tris

Il problema:

vogliamo realizzare un programma che consente di giocare a *Tic-Tac-Toe*. Abbiamo come **vincolo** di progettazione che il programma deve essere *realizzato in C*.

Il programma deve consentire ad un umano di *giocare contro il computer*.

1. Scomposizione in moduli.

Il primo passo nella progettazione di dettaglio è la *scomposizione in moduli*.

Decidiamo di seguire un **approccio funzionale**, quindi partiamo dalle operazioni che il programma deve svolgere:

- *Gestione della scacchiera* (inizializzazione, aggiornamento, ecc.)
- *Visualizzazione di informazioni all'utente*
- *Acquisizione di input dall'utente*
- *Calcolo della mossa del computer*
- *Un main che richiama le altre funzioni nell'ordine giusto*

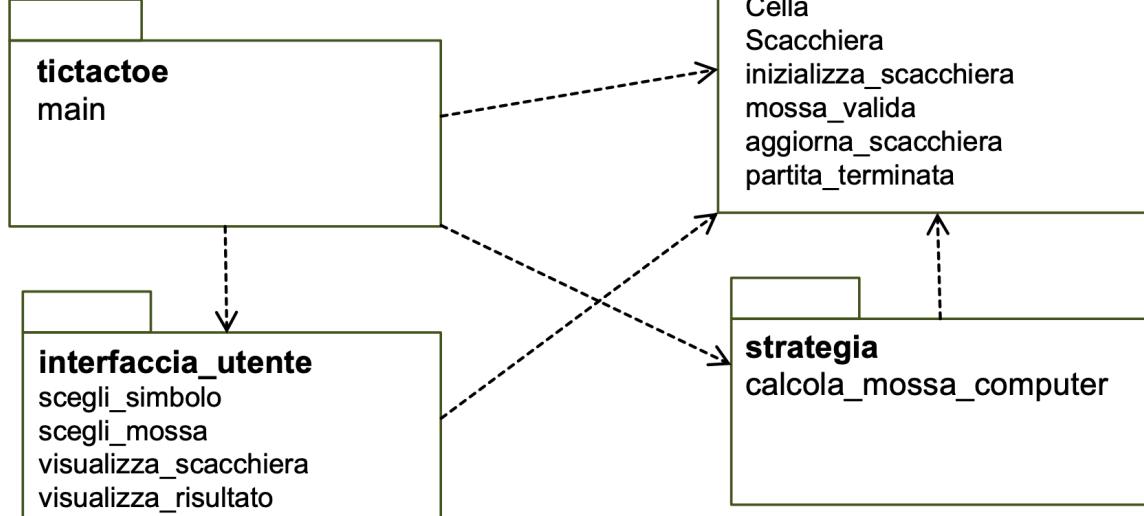
2. Definizione delle interfacce.

Coninciamo ad individuare le *operazioni di ciascun modulo*.

- Ci serve una *struttura dati* per rappresentare una scacchiera
- Ci serve un *tipo* per rappresentare il contenuto di una *cella* della scacchiera.

3. Valutazione di coesione ed accoppiamento.

Proviamo a valutare coesione e accoppiamento dei moduli progettati



Possiamo migliorare i problemi di accoppiamento individuati aggiungendo due operazioni al modulo scacchiera:

```

Cella prossimo_giocatore(Scacchiera *s){
    return s->prossimo;
}
    
```

```

Cella contenuto_cellula(Scacchiera *s, int posizione){
    return s->cella [ posizione ];
}
    
```

4.6 Progettazione Object-Oriented

4.6.1

Decomposizione in classi Il primo passo è l'*individuazione delle classi* del programma. Le fasi fondamentali sono:

1. **Analisi grammaticale:** si inizia da una *descrizione in linguaggio naturale* del sistema da realizzare.
 - I sostantivi (*nomi*) usati nella descrizione sono candidati a diventare classi, oppure attributi di classi
 - I *verbi* usati nella descrizione sono candidati a diventare metodi
2. **Entità del dominio del problema:** si parte dalle *descrizioni dei requisiti*. Sono candidati a diventare classi:
 - Oggetti fisici (es. Prodotto, Fattura)
 - Ruoli (es. UtenteRegistrato)

- Eventi (es. Consegna)
- Unità organizzative (es. Reparto)
- altre "entità" significative

3. **Analisi del comportamento:** si parte dalle *descrizioni del comportamento del sistema*, che vengono suddivise in operazioni più semplici. Si individuano come classi:

- Chi o cosa fa partire un'operazione
- Parti del sistema che possono partecipare allo svolgimento di un'operazione (perchè hanno accesso a tutte le informazioni necessarie)

Le tencinche citate foniscono un'**ipotesi iniziale** delle classi del sistema, ma questa ipotesi va **raffinata**, aggiungendo o rimuovendo classi, per arrivare alla decomposizione vera e propria.

Una volta individuate le classi è importante **stabilire come esse sono collegate tra loro** (come collaborano per lo svolgimento delle loro funzioni).

Infine bisogna specificare l'**interafaccia** (il "contratto") di ciascuna classe.

4.6.2 Diagrammi delle Classi

Descrivono un sistema attraverso:

- le **classi** che lo compongono
- i loro **attributi** e le loro **operazioni**
- le **relazioni** tra loro

I diagrammi delle classi sono **modelli statici**:

- mostrano *ciò che interagisce* nel sistema ma non quello che succede *durante le interazioni*

Notazioni UML

Una classe è rappresentata con un rettangolo diviso in tre parti:

- **Nome** della classe (unica obbligatoria)
- **Attributi** della classe, detti anche *Priorietà*.

Sintassi:

visibilità nome: tipo[molteplicità] = default proprietà

Esempio:

-nome: String[1] = "No Title" readOnly

- *nome*: unico obbligatorio
- *tipo*: che l'oggetto può contenere
- *molteplicità*: quanti oggetti contiene
- *default*: il valore per un oggetto appena creato
- *proprietà aggiuntive*: readOnly, ordered, unique
- *visibilità*: public, private, protected, package ()

- **Operazioni** della classe, *Metodi*.

Sintassi:

visibilità nome(lista-parametri): tipo-di-ritorno proprietà

Esempio:

- + balanceOn (data: Date): Money
 - *nome*: obbligatorio
 - *tipo-di-ritorno*: valore restituito
 - *proprietà aggiuntive*: ad esempio query (il metodo non modifica gli attributi della classe, fa solo lettura)
 - *lista-parametri*)
 - *direzione*: indica se il parametro è in ingresso (in), uscita (out) o entrambi (inout)

Per gli attributi ed i metodi è *possibile* specificare:

- **Informazioni di tipo** dell'attributi, del valore di ritorno. Nomi e tipi dei parametri
- **Modificatori:**

- Privato : -
- Pubblico : +
- Protetto:
- static : sottolineato
- abstract : *corsivo*

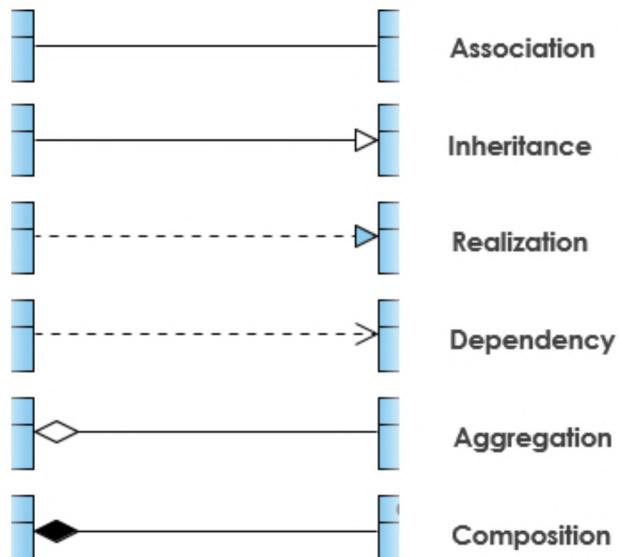
I diagrammi di classe sono utili da due punti di vista:

1. **Concettuale**: rappresentano i concetti del dominio
2. **Implementativo**: descrivono come le classi saranno implementate

La *prospettiva* influenza la quantità di dettagli da mostrare.

Relazioni

Le linee o le frecce tra le classi indicano le **relazioni** tra le classi



L'**ereditarietà** è una relazione tassonomica tra una classe più generale (superclasse) e una sua sottoclasse. Indicata anche come la relazione "**is-a**".

Chiamato anche **specializzazione** o, se letta dall'altro lato, *generalizzazione*. Indicata dal *triangolo* che punta alla superclasse.

Associazione

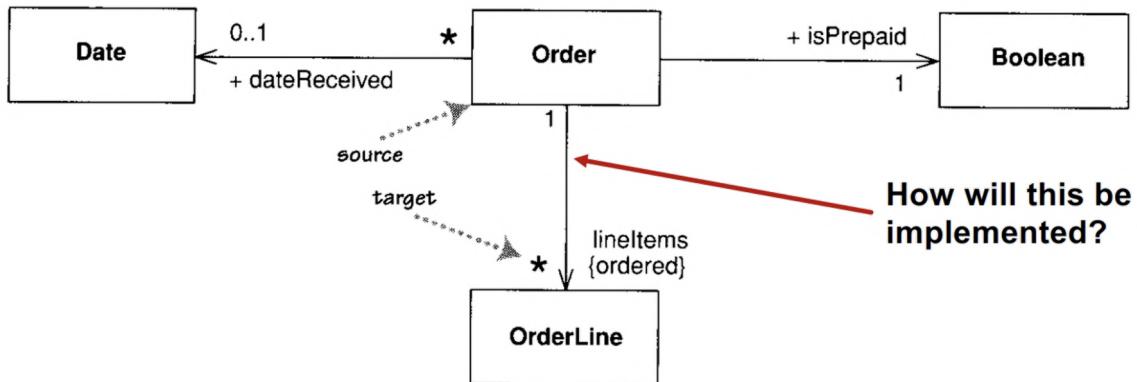
Un oggetto di una classe mantiene un riferimento ad un oggetto dell'altra per poter svolgere il proprio lavoro. Indicata anche come la relazione "**has-a**".

- Indicata da una linea retta
- Può avere associato un **ruolo** che spiega il significato dell'associazione (nome della proprietà)
- Può essere annotata con la *moltiplicità*

L'associazione può essere:

1. **bidirezionale** entrambe le entità "si conoscono". Hanno entrambe un attributo dell'altra classe.
2. **unidirezionale:** *A* mantiene un riferimento ad un oggetto della classe *B*, ma *B* non mantiene un riferimento ad un oggetto della classe *A*.

Un'associazione con un'altra classe equivale ad avere una proprietà che abbia come tipo un valore dell'altra classe.



Usiamo una forma o l'altra a seconda di cosa vogliamo mettere in evidenza

Order
+ dateReceived: Date [0..1] + isPrepaid: Boolean [1] + linelitems: OrderLine [*] {ordered}

Una classe può avere un'associazione con se stessa (*self-association*).

- Questo non vuol dire che un oggetto della classe mantiene un riferimento a se stesso
- Il significato di una *self-association* è che un oggetto della classe mantiene un riferimento ad un oggetto della stessa classe.

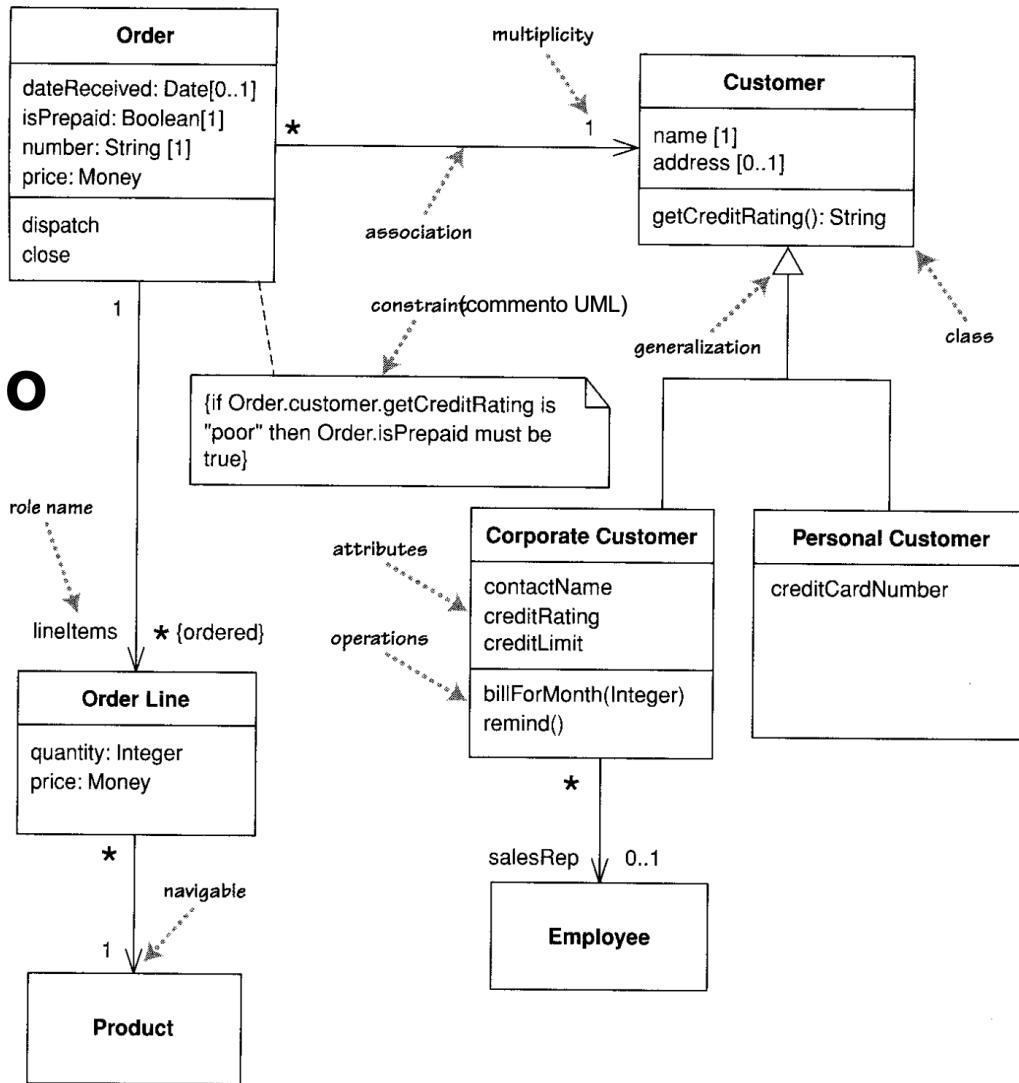
Per esempio:

Persona -> self-association -> coniuge.

Perchè coniuge fa riferimento ad un oggetto della classe Persona

Molteplicità	Significato
<i>n..m</i>	da <i>n</i> a <i>m</i> istanze
0..1	zero o un'istanza
0..* oppure *	nessun limite al numero di istanze (anche nessuna)
1	esattamente un'istanza
1..*	almeno un'istanza

Esempio



Aggregazione

Un caso *speciale di associazione* in cui una classe rappresenta un insieme di parti. Indica una relazione di "parte di", "is-made-of". Indicato da un diamante vuoto sul lato della collezione. L'aggregazione è una *relazione asimmetrica*.

L'**aggregazione** può essere spesso *sostituita* da un'*associazione uno-a-molti*. In sintesi, la frase sottolinea che l'aggregazione, in alcuni casi, può essere semplificata o sostituita da una più generica relazione di associazione uno-a-molti, riducendo complessità di modellazione.

Composizione

Forma più forte di aggregazione in cui:

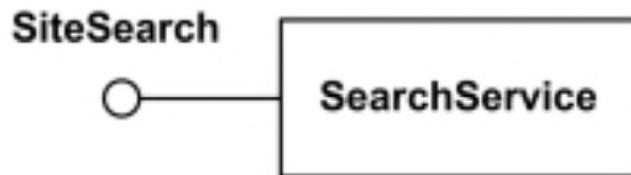
- una parte può essere inclusa in un solo insieme
- una parte non può esistere se non è inclusa nell'insieme. Quando l'insieme viene rimosso, deallocated, vengono rimosse anche tutte le sue parti
- Indicato da un diamante pieno sul lato della collezione



Implementazione

Una relazione tra una **specifica** e la sua *implementazione*.

- Utilizzato per mettere in relazione una classe con un'*interfaccia* che è implementata dalla classe
- Specifica che la classe è conforme al contratto specificato dall'interfaccia

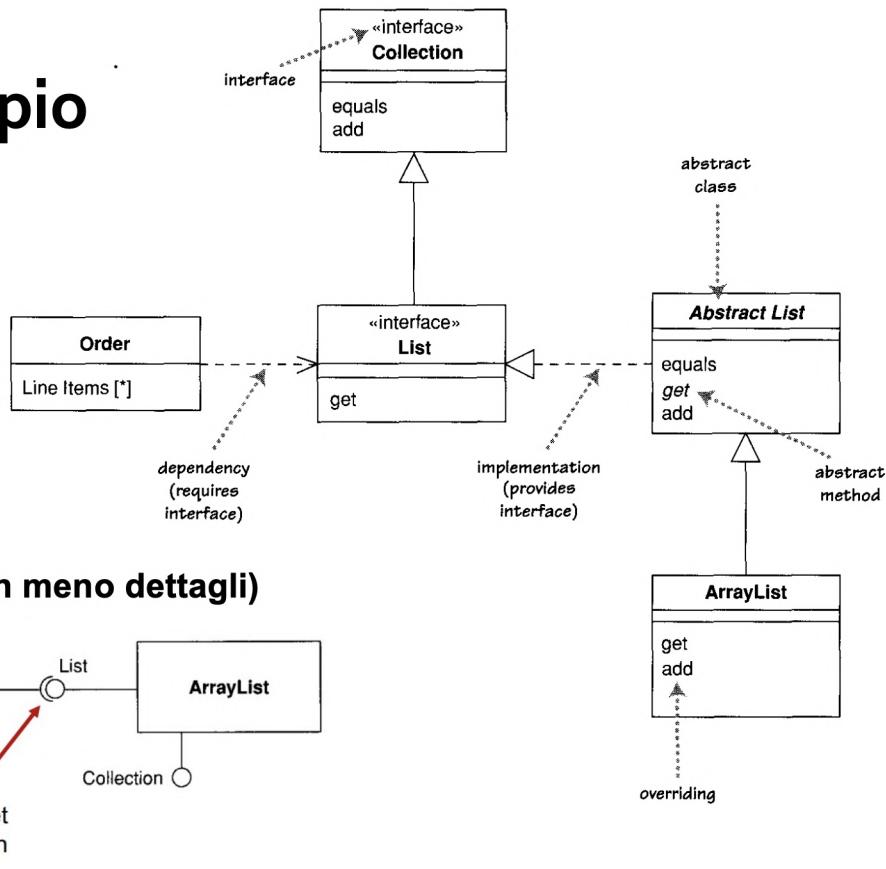


Dipendenza

Una relazione di dipendenza indica che una classe dipende da un'altra per un motivo diverso dalle altre relazioni viste in precedenza.

- Ad esempio, una classe A usa una classe B in una variabile locale, in un paramtro o nel valore di ritorno di un metodo
- Altro esempio, la classe A richiama un metodo static della classe B

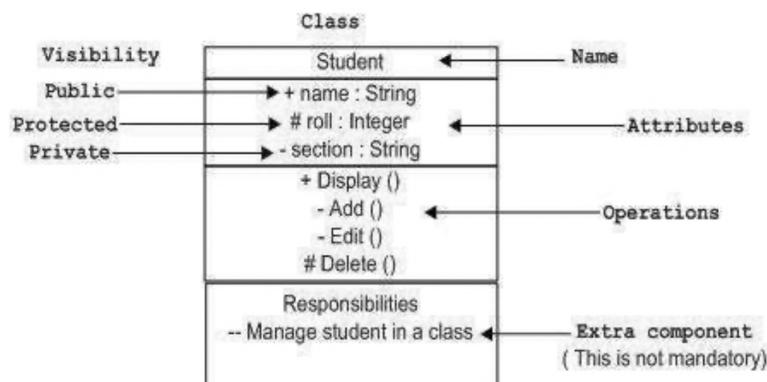
Esempio



Responsabilità della classe

Spesso è utile indicare esplicitamente le responsabilità di una classe

- Esiste un quarto scomparto opzionale per questo
- le responsabilità possono essere descritte come testo semplice



Diagrammi degli oggetti

Qualche volta può essere utile, per chiarire il funzionamento di una classe, mostrare lo stato di un insieme di oggetti in un determinato momento dell'esecuzione.

Un *diagramma degli oggetti* (o diagramma delle istanze) serve a questo scopo, usando una notazione simile al diagramma delle classi.



4.6.3 Ereditarietà, contratti e dipendenze

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

Un oggetto della classe *derivata* **deve** poter essere usato in tutti i contesti in cui viene usato un oggetto della classe base. **Principio di sostituzione di Liskov.**

Quali sono le conseguenze del *principio di sostituzione* sul contratto della classe derivata ?

- La sostituzione **non deve** "danneggiare" il client
- Quindi:
 1. i *benefici per il client* devono restare uguali o aumentare
 2. gli *obblighi per il client* devono restare uguali o diminuire.

La classe derivata può **aggiungere attributi** o **metodi** al contratto, ma non può rimuoverne.

La classe derivata può **ridurre** (rendere meno stringenti) le *precondizioni* di un metodo, ma non può aumentarle.

La classe derivata può **aumentare** (rendere più stringenti) le **postcondizioni** di un metodo, ma non può ridurle.

Il tool EasyUML

Un plugin NetBeans per i diagrammi di classe:

- Forward Engineering: Genera codice Java da diagrammi di classe
- Reverse Engineering: Genera diagrammi di classe dal codice Java

* INTERFAZIA: USAΓΑ PER DEFINIRE UN CONTRATTO SENZA FORNIRE UN'IMPLEMENTAZIONE DEL CONTRATTO
PER NEΓΕ DI INTERROGRERE CONCRETI PER DIPENDENZE INDIREΓΓΕ

4.7 Diagrammi di Interazioni

I diagrammi UML visti finora rappresentano **modelli statici** del sistema.

- Descrivono l'organizzazione del sistema, e le relazioni tra le sue parti, ma non il loro comportamento

Aspetti del comportamento del sistema possono essere descritti mediante altri diagrammi UML, che rappresentano *modelli dinamici*.

4.7.1 Diagrammi di interazione

I diagrammi di interazione descrivono il **modo in cui gruppi di oggetti collaborano tra loro.**

UML definisce diversi tipi di diagrammi di interazione:

- **Diagrammi di sequenza:** si concentrano sulla sequenza dei messaggi tra gli oggetti
- **Diagrammi di comunicazione:** descrivono l'organizzazione strutturale degli oggetti che inviano e ricevono messaggi

Altri diagrammi di interazione (che non vedremo):

- *Diagrammi temporali:* enfatizzano i vincoli temporali
- *Diagrammi di interazione:* mix di diagrammi di sequenza e di attività

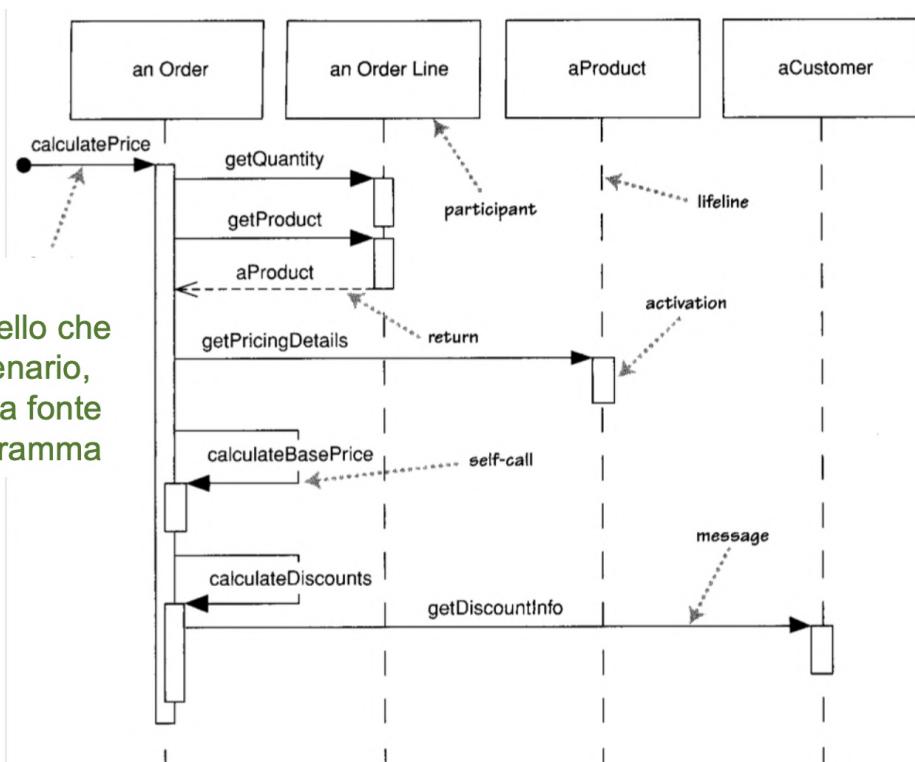
4.7.2 Diagrammi di Sequenza

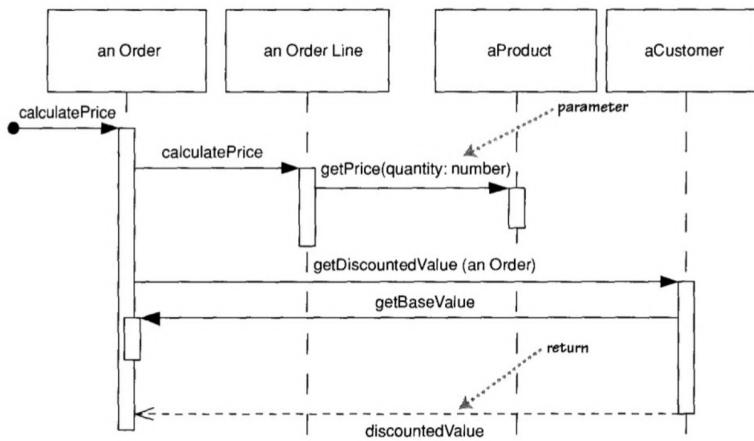
Un diagramma di sequenza cattura il comportamento del sistema (o di un sottoinsieme) in un **singolo scenario**. E' il più comune tra i diagrammi.

Mostra gli oggetti coinvolti ed i messaggi che vengono scambiati tra questi oggetti all'interno del caso d'uso.

In un diagramma di sequenza:

- Il tempo avanza dall'alto verso il basso.
- Gli oggetti coinvolti (partecipanti) sono elencati da sinistra a destra.
- I messaggi tra gli oggetti sono rappresentati mediante frecce che indicano la direzione in cui viaggia il messaggio.





Alcuni partecipanti possono essere creati e/o distrutti durante un'interazione.

Es: in un ambiente che implementa un *garbage collector* non si deallocano esplicitamente gli oggetti, ma può essere utile usare la **X** per indicare quando un oggetto non è più necessario.

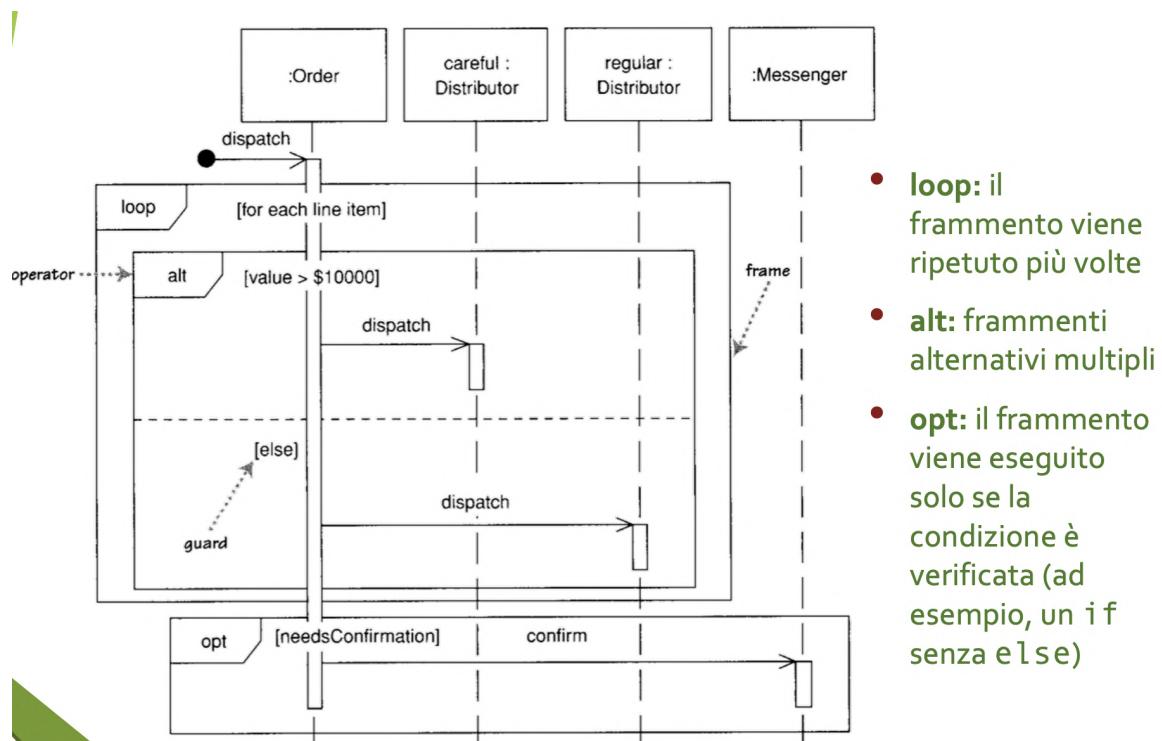
I *nomi* dei **Partecipanti** possono specificare solo la classe dell'oggetto.

Una sintassi più ricca è: "nome:*classe*", il nome è *opzionale* (e.g., *order: Order* ; *:OrderLine*).

I partecipanti possono essere non solo oggetti ma anche altre entità, come gli **attori** dei casi d'uso.

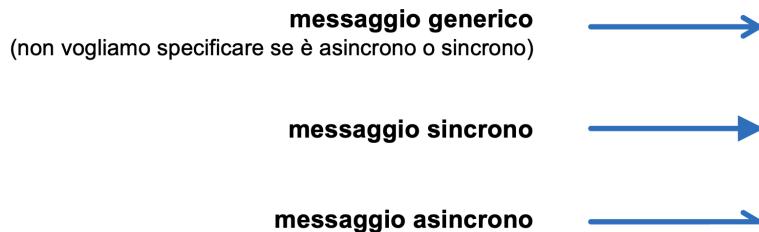
Cicli e condizioni

I diagrammi di sequenza **non sono adatti** a mostrare cicli e comportamenti condizionati. Tuttavia possiamo utilizzare i **frame di interazione**.



Messaggi sincroni / asincroni

- **Messaggio sincrono:** il chiamante deve attendere che l'elaborazione del messaggio sia completata prima di continuare (ad esempio, quello che succede quando si invoca un metodo)
- **Messaggio asincrono:** il chiamante non deve attendere una risposta e può continuare subito l'elaborazione (e.g., applicazioni multithread)

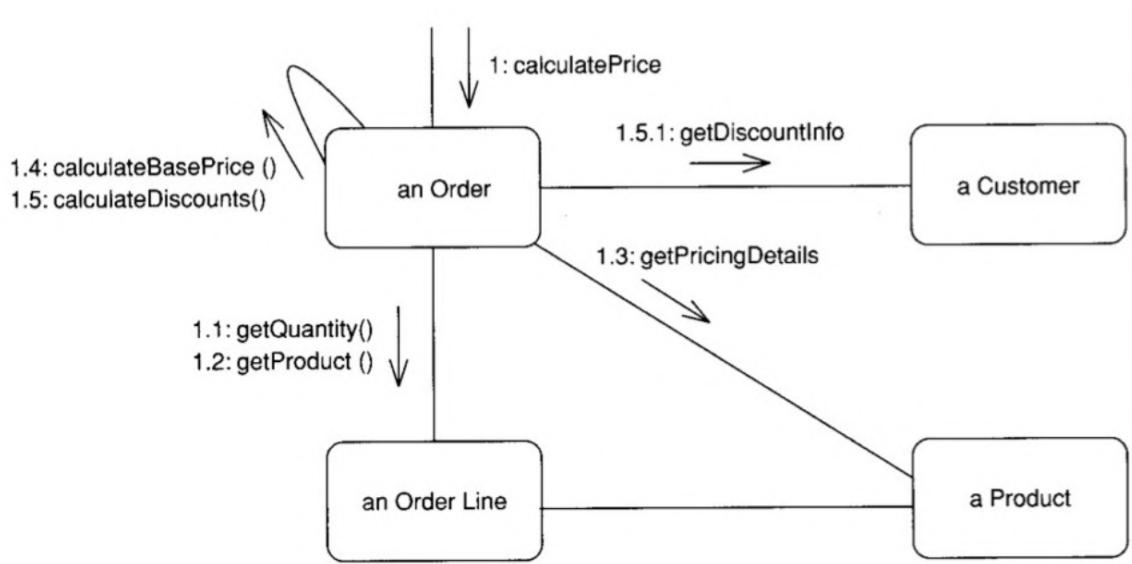


4.7.3 Diagrammi di Comunicazione

I diagrammi di comunicazione (o **collaborazione**) mostrano informazioni simili ai diagrammi di sequenza in un formato differente.

Elementi principali del diagramma:

- un **oggetto** che partecipa ad uno scenario
- i **collegamenti tra oggetti**: linee solide tra gli oggetti che interagiscono
- i **messaggi scambiati tra gli oggetti**; frecce con uno o più nomi che indicano la direzione ed i nomi dei messaggi inviati tra gli oggetti.



4.8 Diagramma delle attività - altri diagrammi UML

Visualizzano il *flusso di azioni* coinvolte in un processo.

- Enfasi sulla sequenza e sulle condizioni del flusso: simili ai diagrammi di flusso e alle reti di Petri

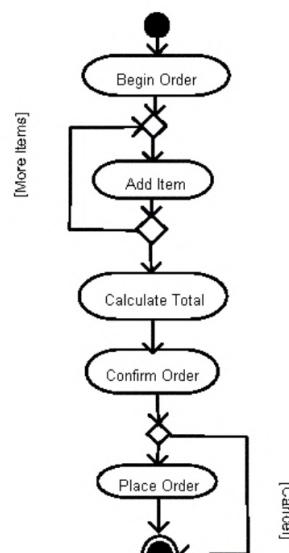
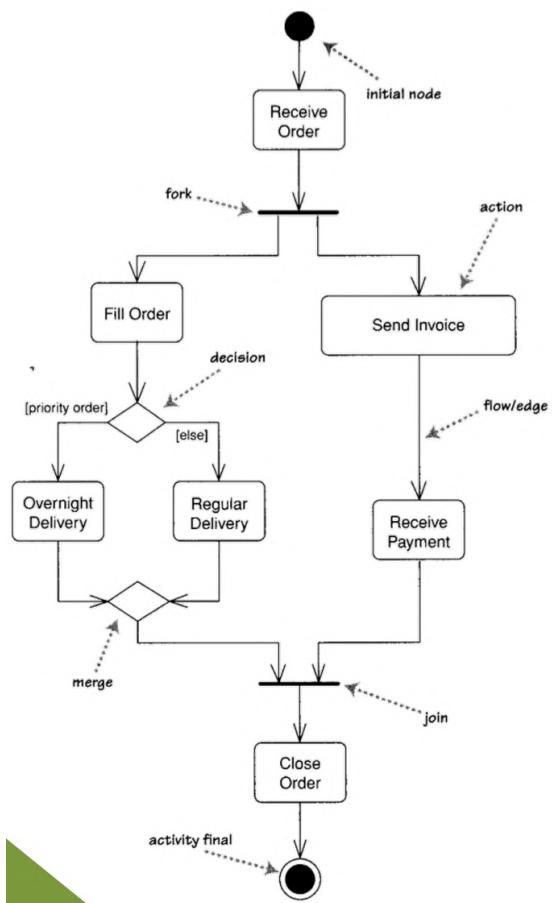
Utilizzabili per modellare il comportamento associato a:

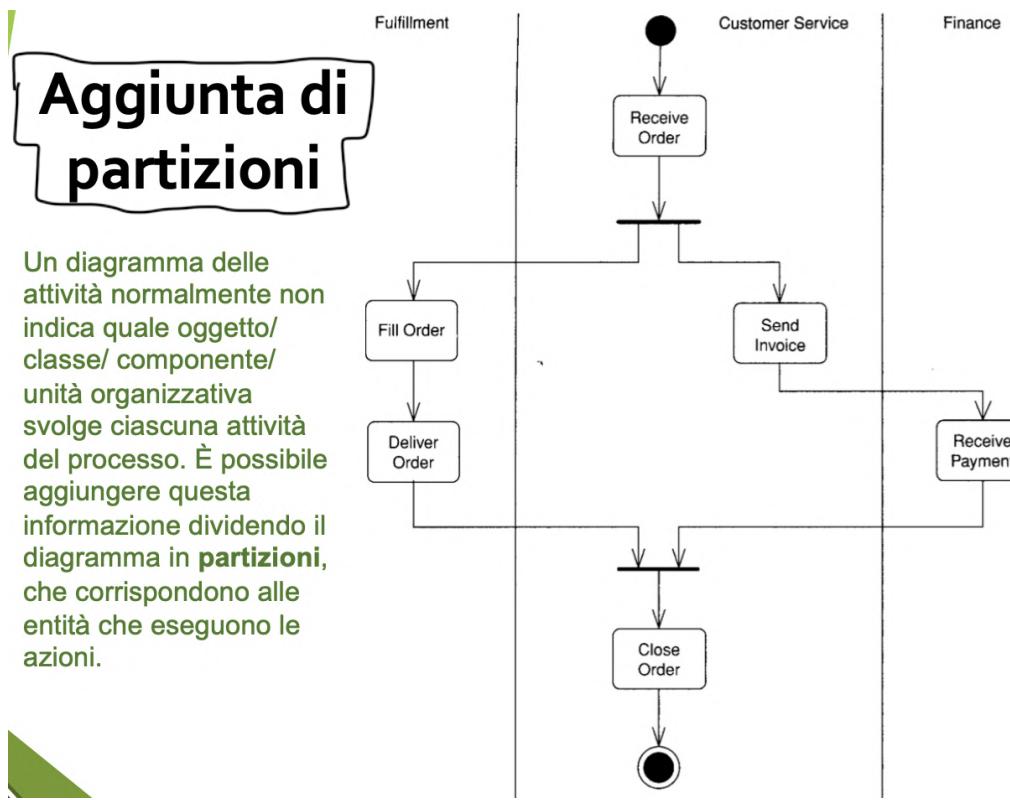
- Processi aziendali
- Casi d'uso
- Metodi
- Algoritmi

Elementi principali:

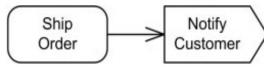
1. **Azione**: l'esecuzione di un comportamento
2. **Flusso (edge)**: collega le azioni in una sequenza
3. **Decisione (branch)**: dirama il flusso di input testando una condizione - terminato da un **merge**
4. **Fork**: trasforma un singolo flusso in ingresso in più flussi paralleli in uscita - terminato da un **join**
5. **Partizioni (swimlanes)**: indicano quale oggetto è responsabile di quali azioni
6. **Nodi iniziali e finali**

Esempi di diagrammi delle attività

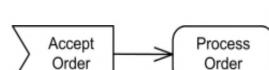




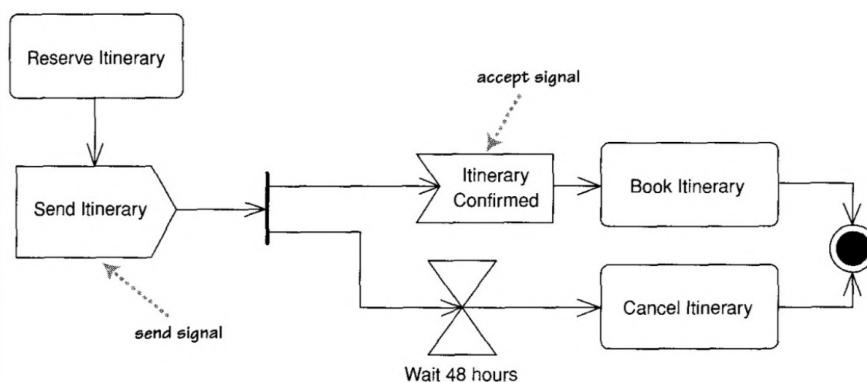
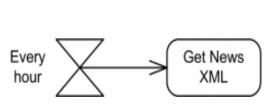
- Inviare un segnale a un processo esterno



- Attendere un segnale da un processo esterno

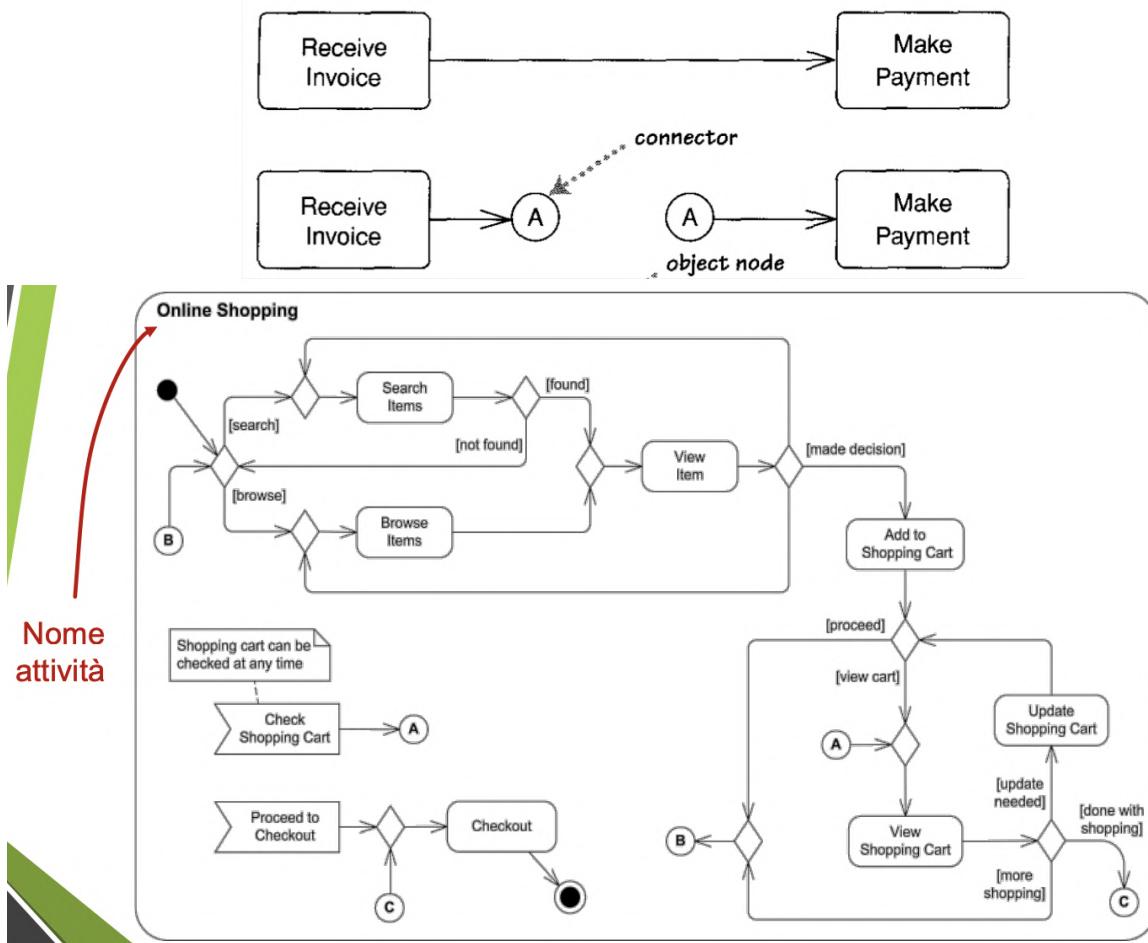


- Segnale temporale (si verifica a causa del trascorrere del tempo)



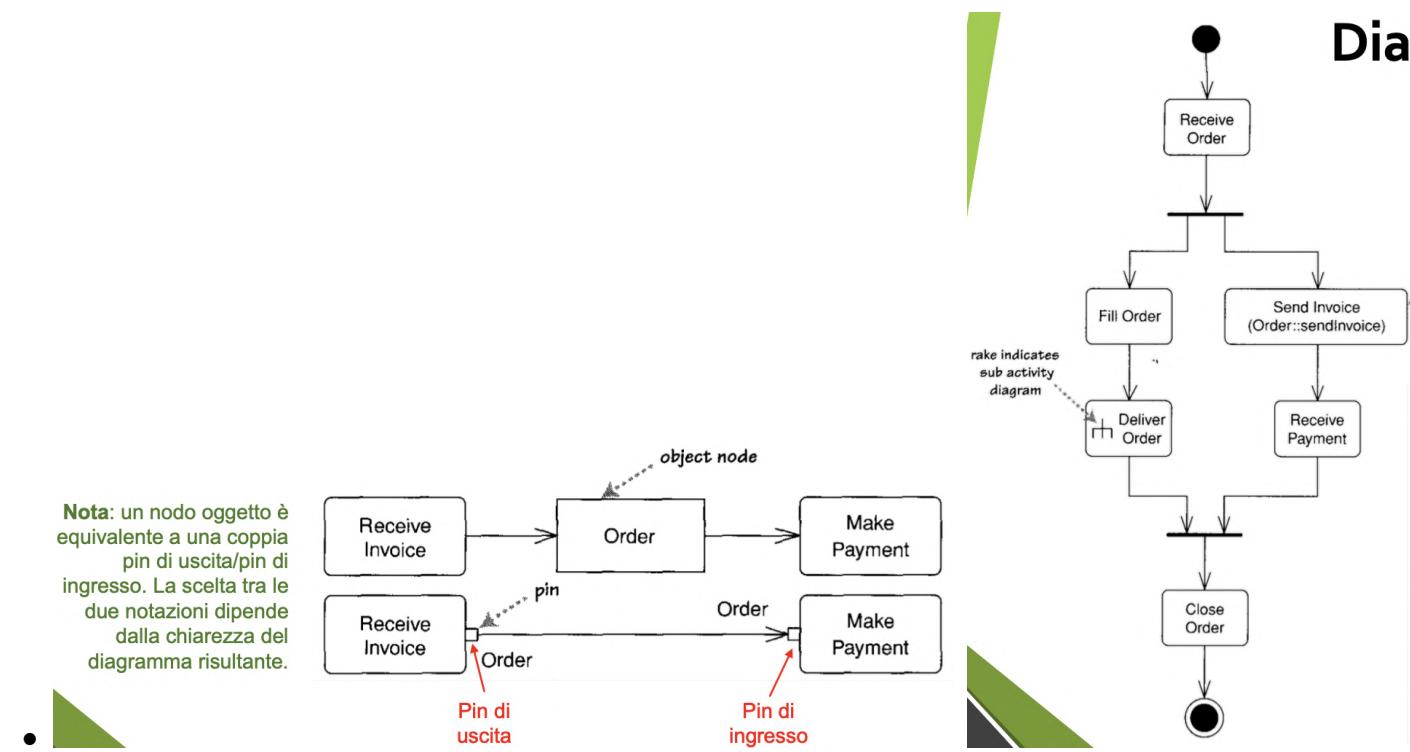
Rappresentano un salto nel diagramma

- Utilizzato quando il diagramma diventa troppo complesso



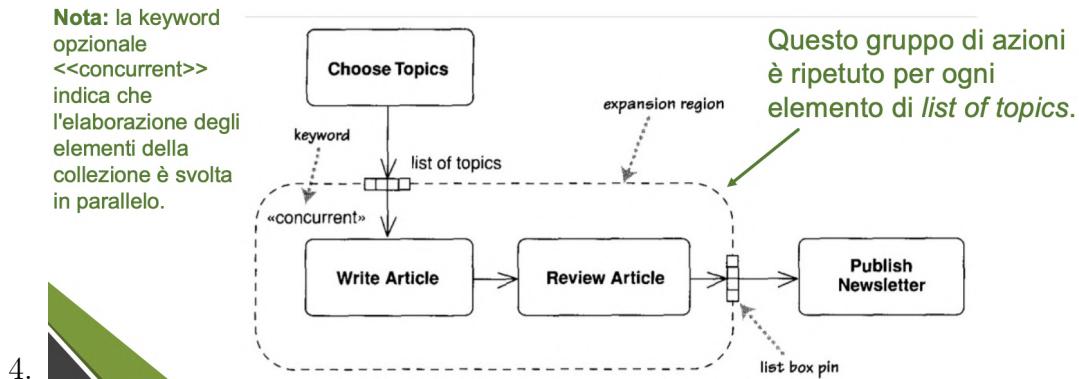
Nodi oggetto vs. pin: è possibile (ma non obbligatorio) indicare le informazioni scambiate tra le azioni in una sequenza.

- Un **nodo oggetto** rappresenta la classe di un oggetto scambiato tra le azioni
- I **pin di uscita** rappresentano le informazioni prodotte da una azione, che saranno usate da altre azioni
- I **pin di ingresso** rappresentano le informazioni usate da un'azione, che saranno prodotte da altre azioni.

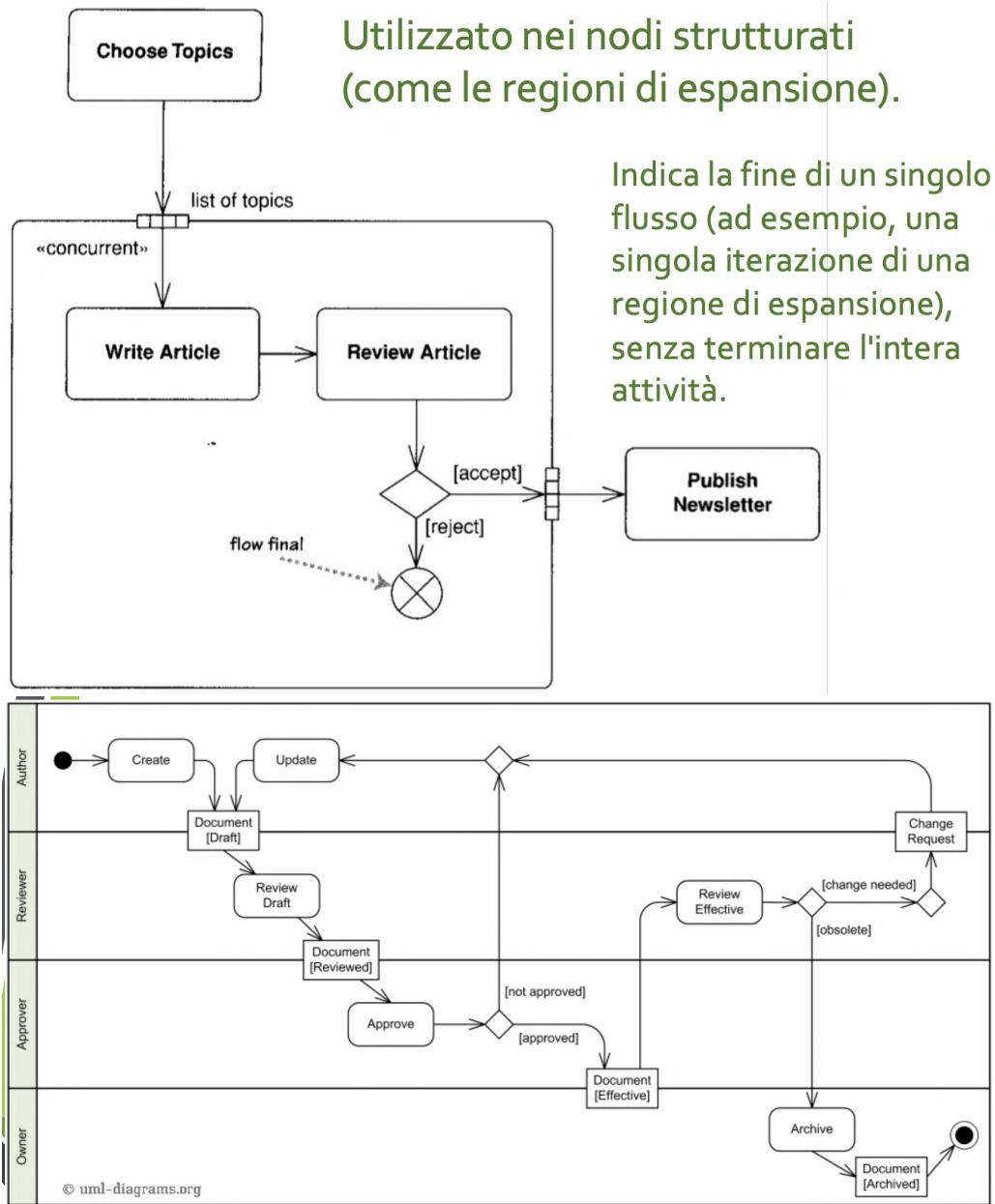


Regioni di espansione: sono *nodi strutturati* che:

1. prendono in input una *collezione di oggetti* (es. array)
2. agiscono su ogni elemento delle collezioni individualmente
3. producono elementi che confluiscono in una collezione in output



Finale di flusso



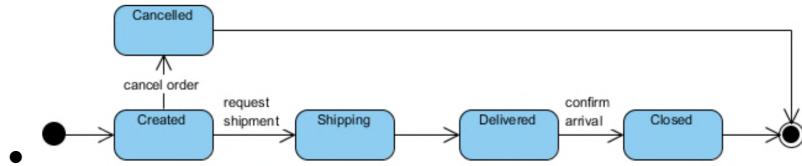
4.8.1 Diagrammi di macchina a stati

Detti anche "diagramma degli stati".

Mostrano i possibili **stati** di un oggetto e le **transizioni** che causano un cambiamento di stato.

- Basati sugli *automati a stati finiti deterministici (DFA)*
- Possono essere usati per mostrare il comportamento di un oggetto durante il suo ciclo di vita.
- *NOTA: non sempre il comportamento di un oggetto può essere descritto adeguatamente da un diagramma di macchina a stati.*

- Lo **stato** di un oggetto può essere visto come una combinazione dei valori delle sue proprietà
- Tuttavia, lo stato è una notazione più astratta: stati diversi implicano un modo diverso di reagire agli eventi.



Transizioni

Una transizione indica il passaggio da uno stato ad un altro.

Ha un'etichetta il cui formato è: **trigger[guardia]/attività** (tutti gli elementi sono facoltativi).

- **Trigger:** un evento che innesca un potenziale cambiamento di stato
- **Guardia:** una condizione che deve essere vera affinché la transizione venga eseguita
- **Attività:** un comportamento che viene eseguito durante la transizione.

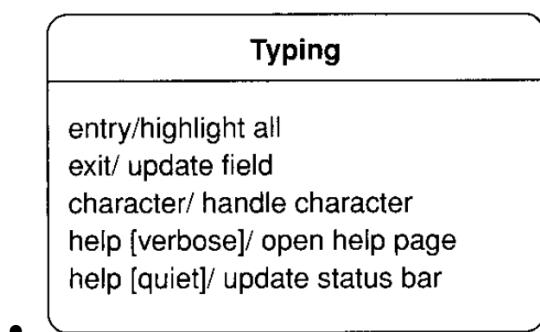
Attività interne

Gli stati possono reagire ad alcuni eventi senza effettuare transizioni, utilizzando le **attività interne**:

- *Trigger, guardia ed attività* sono indicati all'interno della casella di stato
- Un'attività interna è *quasi* equivalente ad un'auto-transizione (una transizione da uno stato verso se stesso)

Attività speciali

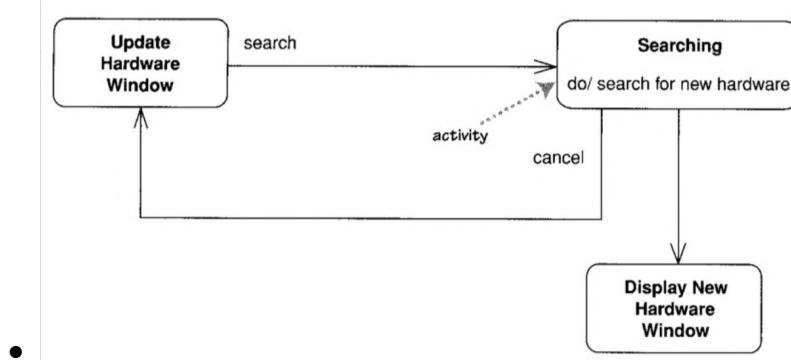
- L'attività associata al trigger **entry** viene eseguita ogni volta che si entra nello stato. Analogamente, l'attività associata al trigger **exit** viene eseguita quando si esce dallo stato.
- Le attività interne non innescano le attività di *entry* e *exit* (a differenza delle auto-transizioni esplicite).



Stati attività

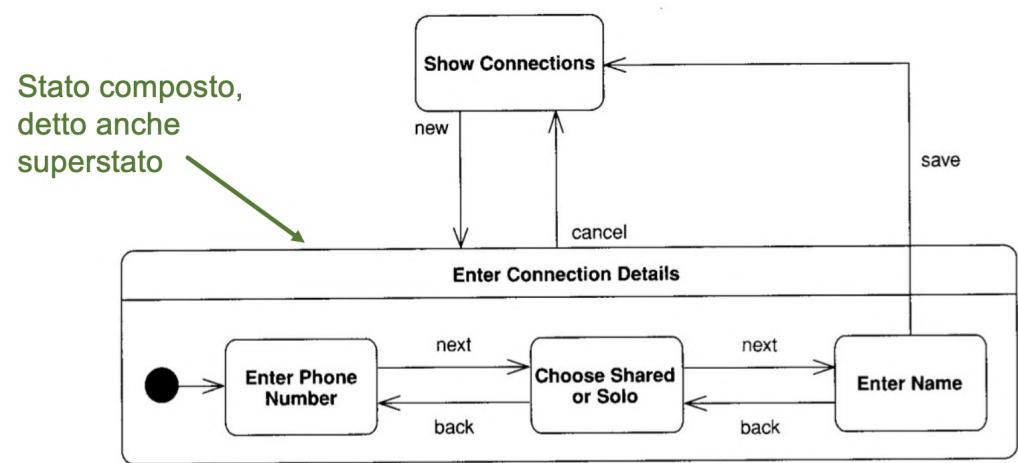
Stati in cui l'oggetto si ferma a svolgere un lavoro.

- Il lavoro associato allo stato è contrassegnato dal trigger **do/**
- Una volta completato il lavoro, viene effettuata la transizione senza trigger che esce dallo stato

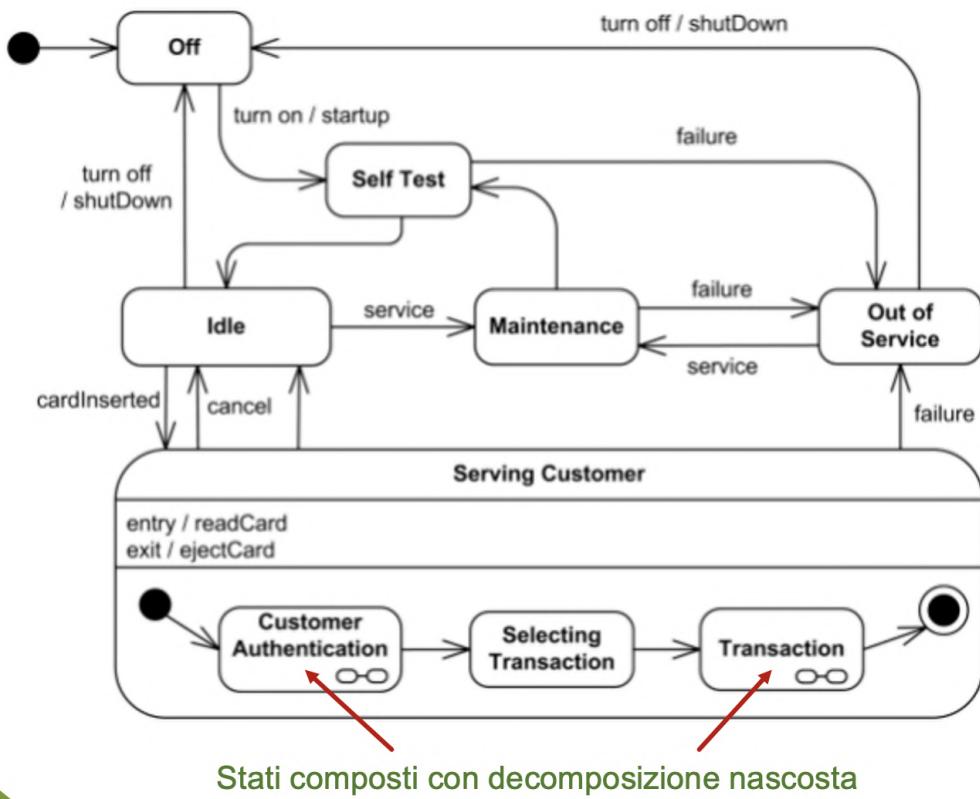


Stati composti (superstati)

Raggruppano gli stati che condividono transizioni e/o attività interne comuni.



Da ciascuno dei sotto-stati dello stato composto si passa allo stato "Show Connections" quando si verifica l'evento "cancel".



4.8.2 Diagrammi di distribuzione

Correlano l'**architettura fisica** dell'hardware con il sistema software implementato

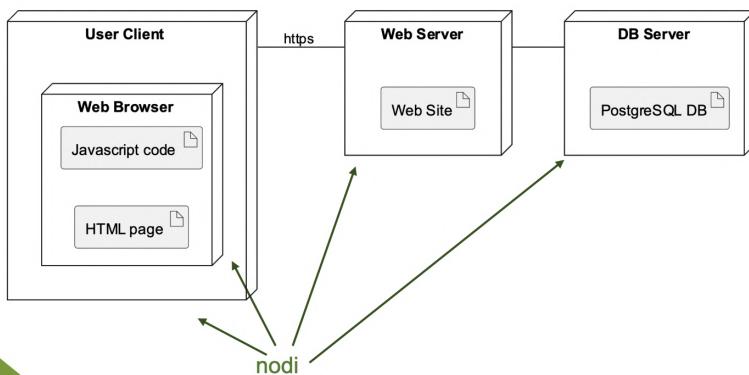
- Quale componente software funziona su quale dispositivo software ?
- E' costituito da **nodi** collegati da *percorsi di comunicazione*

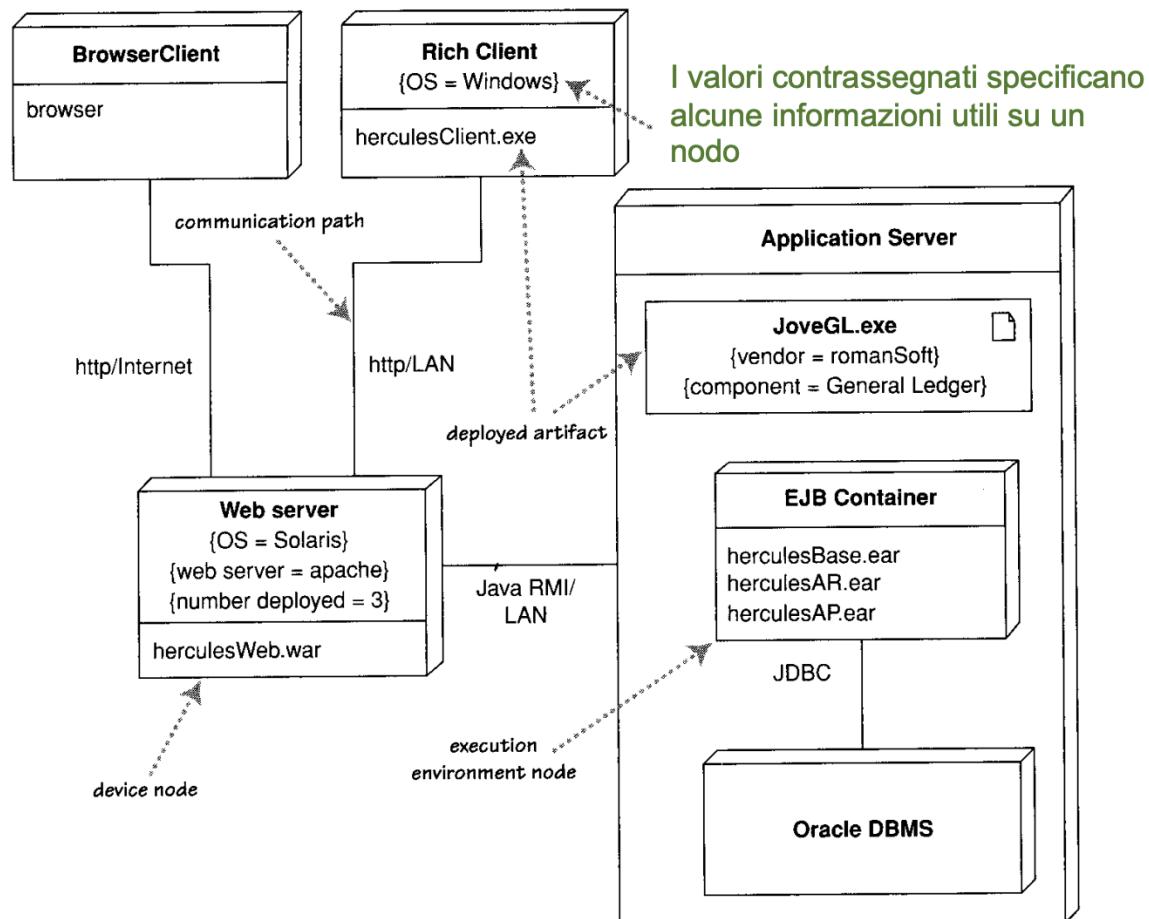
Nodi: qualcosa che può ospitare software

- *Dispositivo*: nodo hardware, es. computer
- *Ambiente di esecuzione*: software che contiene altro software (OS, container, hypervisor)

I *nodi* contengono **artefatti**:

- *file eseguibili*
- *file non eseguibili*: dati, file di configurazione, pagine HTML





Capitolo 5

Principi di Buona Progettazione

Vale la pena progettare bene il software ?

Fraintendimenti comuni

Non bisogna mai cofondere:

- 1. la **qualità della progettazione** del software
- 2. **con la qualità della documentazione** della progettazione del software

E' possibile avere una **pessima** progettazione: *prendere decisioni completamente sbagliate rispetto alle loro conseguenze sugli attributi di qualità che ci si era prefissi di raggiungere.*

Accompagnata da una **documentazione di progettazione perfetta**: una documentazione chiara e dettagliata di tutte le scelte progettuali.

Viceversa, è possibile avere una **ottima progettazione** ma non lasciare *nessuna documentazione* delle decisioni progettuali effettuate:

- come e quanto documentare le decisioni progettuali dipende dal processo software che abbiamo deciso di seguire
- in diversi processi software moderni, i modelli usati per la progettazione di dettaglio **non** vengono conservati come parte della documentazione

La progettazione non è un'attività "binaria" (fatta o non fatta), ma ci sono *diversi gradi di effort* che possono essere "investiti" in questa attività.



Debito tecnico

In un progetto software si usa il concetto di **debito tecnico** per indicare la cattiva progettazione di qualche aspetto. Avevre una parte del software *progettata male*:

- mi permette di *rilasciare il software più velocemente*
- ma l'effort che dovrò spendere alla fine sarà maggiore di quello necessario per una buona progettazione (**mi servirà più effort per correggere, aggiornare ed estendere le funzionalità**).
- In casi estremi, se i debiti tecnici si accumulano, l'effort per la correzione, aggiornamento ed estensione del software porterà al **fallimento**.

Tipi di debito tecnico: il *quadrante del debito tecnico*, proposto da Martin Fowler, **categorizza** le motivazioni del debito tecnico.

La tabella ha due dimensioni:

1. **l'attitudine:** che può essere *sconsiderata* o *prudente*
2. la **consapevolezza:** nell'assunzione del debito (il debito è *deliberato* oppure *accidentale*)

	Sconsiderato	Prudente
Deliberato	"Non abbiamo tempo per fare la progettazione!"	"Dobbiamo fare il rilascio ora, e poi affronteremo le conseguenze di questa scelta."
Accidentale	"Che cos'è la divisione del software in Layer?"	"Adesso abbiamo capito come avremmo dovuto farlo."
3. ▶		

Refactoring:

- Un *debito tecnico non è un bug* (il software funziona correttamente)
- tuttavia, è utile dedicare una parte del lavoro a ridurre i debiti tecnici
- Questa attività prende il nome di **refactoring**: modifco del codice già funzionante, senza aggiungere nuove funzionalità, per migliorare la qualità della progettazione.

5.1 Principi generali di buona progettazione

- **KISS:** *Keep It Simple, Stupid!*.

Semplicità significa preferire:

- classi semplici, con poche operazioni
- metodi brevi
- algoritmi e strutture dati non complicati
- una struttura chiara, facile da comprendere

- **SINE:** *Simple Is Not Easy.*

La soluzione più semplice **non** è la più facile da realizzare, o quella che viene in mente prima, o quella che una persona inesperta capisce più facilmente.

La semplicità richiede lavoro, esperienza, competenza.

- **DRY:** *Don't Repeat Yourself.*

Ogni "idea" deve essere implementata **in un solo punto** del codice sorgente:

- se un pezzo di codice è utilizzato in due punti, va creata un'astrazione di quell'algoritmo e richiamarla due volte
- scoprire delle ripetizioni è uno dei segnali che dovrebbero far sospettare una cattiva progettazione

La duplicazione può portare ad una manutenzione molto complessa e a contraddizioni logiche.

- **YAGNI:** *You Aren't Going to Need It.*

Spesso gli sviluppatori cedono alla tentazione di aggiungere funzionalità che non sono necessaria. Questa tendenza porta a rendere più generale del dovuto una certa funzionalità.

Il principio YAGNI raccomanda di resistere a questa tentazione

1. dedicare tempo alla realizzazione di funzionalità che *forse* saranno necessarie domani, significa togliere tempo alla realizzazione di funzionalità che oggi sono *sicuramente* necessarie.

- **Separazione delle preoccupazioni** (*separation of concerns*).

Aspetti diversi del sistema devono essere gestiti da moduli distinti e non sovrapposti:

- semplifica lo sviluppo e la manutenzione delle applicazioni software;
- quando i problemi sono ben separati, le singole parti possono essere riutilizzate, oltre che sviluppate ed aggiornate in modo indipendente.

- **Ortogonalità:** le cose che non sono collegate concettualmente non dovrebbero essere "legate" in maniera fissa nel sistema, ma dovrebbero poter essere cambiate indipendentemente.

- E' associato alla semplicità ed alla separazione delle preoccupazioni
- Più il design è ortogonale, minore è il numero di eccezioni o casi particolari da considerare e ricordare.

- **Principio della minima sorpresa:** il codice deve essere strutturato in modo da non sorprendere o confondere chi dovrà leggerlo o manutenerlo.

 - Usare sempre le stesse convenzioni per i nomi
 - evitare che la stessa cosa venga chiamata in punti diversi con nomi diversi

Detto diversamente: "codifica sempre come se la persona che manterrà il tuo codice fosse uno psicopatico violento che sa dove vivi".

- **Evitare l'ottimizzazione precoce:** pensare all'ottimizzazione del codice solo quando il codice funziona correttamente ma è più lento di quanto vorremmo.

 - "L'ottimizzazione prematura è la radice di tutti i mali" - Donald Knuth
 - "La strategia è sicuramente: prima farlo funzionare, poi farlo funzionare bene ed, infine, renderlo veloce" - gente

- **Regola del boy-scout:** "lasciate il campo più pulito di come lo avete trovato".

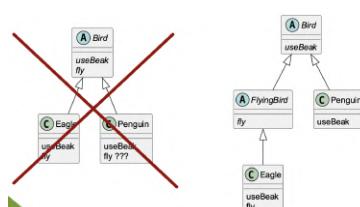
 - Quando si apportano modifiche ad un codice esistente, la qualità tende a diminuire
 - In questi casi occorre prestare molta attenzione a non compromettere la qualità
 - Ogni volta che ci si imbatte in codice non sufficientemente chiaro, si dovrebbe cogliere l'opportunità di correggerlo subito.

ROBERT 5.1.1 Principi di buona progettazione orientata agli oggetti

R. Martin nel 2000 ha descritto 5 principi come fondamentali per la progettazione orientata agli oggetti.

SOLID:

1. Single Responsibility Principle: un componente del codice deve svolgere un singolo compito ben definito
→ **APPRO ALCUNI CONCETTI / CHIUSO ALLE MODIFICHE**
2. Open-Closed Principle: nella programmazione orientata agli oggetti, meccanismi come l'ereditarietà e il polimorfismo possono essere di aiuto per raggiungere questo obiettivo.
 - evitare l'accesso ai dettagli implementativi della classe (incapsulamento)
 - inserire nella classe metodi che corrispondono alle parti che si pensa possa essere utile cambiare
 - usare l'overloading in una classe derivata per cambiare questi metodi, senza modificare la classe base
3. Liskov Substitution Principle: gli oggetti devono poter essere sostituiti con istanze dei loro sottotipi senza alterare la correttezza del programma.
→ **UN OGGETTO DELLA CLASSE DERIVATA**



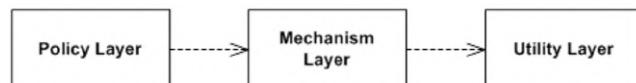
DEVE POTER ESSERE USATO IN UNA COMPOSIZIONE DENTRO CLASSE BASE → **MAT PER REFERENZE (PARAFETTI - TPG DI RITORNO)**

4. Interface Segregation Principle: un client non dovrebbe dipendere da metodi che non utilizza, pertanto è preferibile che le **interfacce** siano *molte, specifiche e piccole*, piuttosto che poche, grandi e troppo generali.

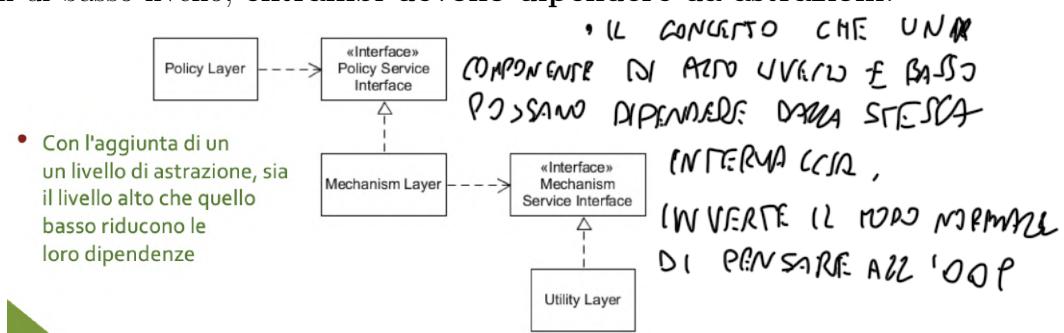
5. Dependency Inversion Principle.

Nelle architetture convenzionali, i componenti di livello inferiore sono progettati per essere usati da componenti di livello superiore, consentendo di costruire sistemi via via più complessi.

- In questa composizione, i componenti di livello superiore dipendono direttamente da quelli di livello inferiore per svolgere un determinato compito.
- Ciò limita le opportunità di riutilizzo dei componenti di livello superiore



Dunque, per il *principio dell'inversione di dipendenza*: i moduli di alto livello non devono dipendere dai moduli di basso livello, **entrambi devono dipendere da astrazioni**.



Privilegiare l'associazione rispetto all'ereditarietà

. L'ereditarietà è una forma di accoppiamento *più forte* dell'*associazione e composizione*.

- Una classe estende un'altra si impegna a rispettare il contratto della classe base.
- Utilizzando l'ereditarietà, le sottoclassi possono facilmente fare assunzioni sbagliate ed infrangere il *principio di sostituzione di Liskov*.
- L'associazione comporta un minore accoppiamento tra le classi.

Principio di Robustezza

Il contratto di un componente software prevede che se il cliente *soddisfa le precondizioni*, il provider *soddisferà le postcondizioni*.

Questo è sufficiente per garantire la **robustezza**.

Tuttavia, nel caso in cui le *precondizioni non* sono rispettate, è preferibile progettare il componente in modo da "limitare i danni" (**robustezza**).

1. Cercare di garantire le postcondizioni anche se i dati ricevuti non soddisfano le specifiche
2. Cercare di garantire che non vi sia una perdita di dati per il client
3. Cercare di garantire che il componente non si porti in uno stato inconsistente.

Capitolo 6

Testing e debugging

6.1 Introduzione del testing

QUALITÀ DEL SOFTWARE

- **Assicurazione della qualità (QA)**: attività volte a misurare e migliorare la qualità di un prodotto e di un processo
- **Controllo qualità (QC) - (testing)**: attività volte a convalidare e verificare la qualità del prodotto attraverso l'individuazione dei difetti e al loro "correzione".

Per **qualità** si intende:

1. Conforme ai requisiti
2. Idoneo all'uso

Queste due proprietà vengono verificate attraverso la:

1. **Verifica**: il software è conforme ai requisiti specificati?
Risponde alla domanda: "*Are we building the system right?*"
2. **Validazione** - coinvolge l'utente finale: il software soddisfa le esigenze degli utenti ?
Risponde alla domanda : "*Are we building the right system?*"

Guasto vs. Difetto:

- Un **fallimento** (*guasto, failure*) è una qualsiasi deviazione del comportamento osservato di un sistema rispetto al comportamento specificato.
- Un **difetto** (*bug*) è una condizione che in alcune circostanze può causare un fallimento
- Un difetto è il risultato di un *errore* commesso da un ingegnere del software o da un programmatore nelle progettazione o realizzazione del codice.

Oltre al *testing*, possono essere effettuate **Software review**: analisi statica del sistema senza eseguirlo.

Walkthrough:

1. lo *sviluppatore presente in modo formale* l'API, il codice e la documentazione del sistema o di un componente al team di revisione
2. il *team di revisione formula commenti* circa la copertura dei requisiti e la coerenza con l'architettura.

Nota: in generale, la software review è più efficace se è svolta da una persona diversa dall'autore del codice.

Si stima che le *software review* consentono di trovare **fino all'85%** dei difetti del codice. Per alcuni tipi di difetti è l'unica tecnica di verifica *efficace* (e.g., problemi legati all'esecuzione parallela o concorrente, diffici da riprodurre).

Una software review può anche evidenziare difetti di progettazione (*debiti tecnici*).

I SPETTACOLI: SINCE AL WALKTHROUGH

MA LO SVILUPPATORI NON PRESENTA
Gli ARREFAZI, SOG SF
VENGONO RICHIESTI CHIARIMENTI

6.2 Le attività di testing

Test del software

Definizione (debole):

- Il test è un processo di verifica *sperimentale* teso a dimostrare l'**assenza di difetti** in un sistema software.
- *sperimentale*: comporta l'esecuzione del programma o di una sua parte (a differenza della software review)

Tuttavia:

- dimostrare l'assenza di difetti *non è possibile*, tranne per programmi banali
- i test possono solo mostrare la presenza di bug, non la loro assenza !

Un test che ha *successo*, ha evidenziato un bug. Altrimenti non si può dire che non ci sono errori.

Definizione (forte):

- Il test è il *tentativo sistematico* di trovare (*sperimentalmente*) *in modo pianificato* i **difetti** nel software implementato.
 - Implica che gli sviluppatori sono disposti a *smentire le cose*.
 - Ciò consente agli sviluppatori di correggere gli errori e di *aumentare l'affidabilità del sistema*.

6.2.1 Concetti di test

Componente testato

- Una *parte del sistema* che può essere isolata per il test
- Può essere una funzione, un oggetto, un gruppo di oggetti o uno o più sottosistemi

Caso di test:

- Un insieme di *input* e di *risultati attesi* che esercita un componente testato
- Lo scopo è quello di *provocare guasti* per rilevare i difetti.

Oracolo di test:

- Un'entità *al di fuori del componente testato* in grado di fornire i risultati attesi per un caso di test (oracolo **esplicito**), o almeno di verificare se i risultati ottenuti sono accettabili (oracolo **implicito**)
- Può essere un'entità software, ma anche un essere umano
- Le *postcondizioni* specificate per un'operazione nel *Design by Contract* possono essere usate per definire un oracolo (implicito).

Debugging:

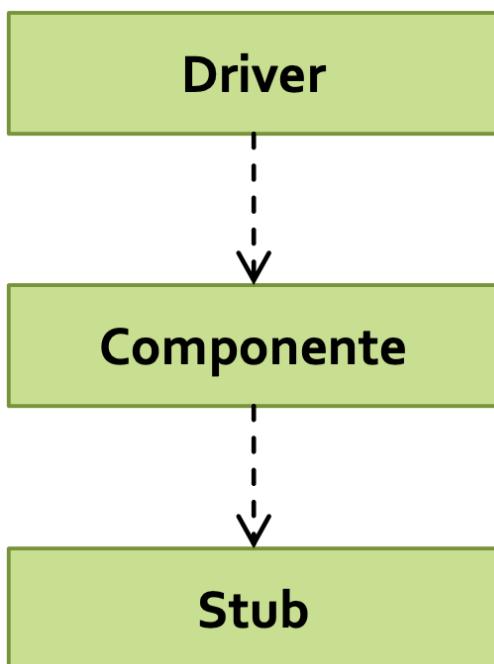
- Se il test rileva la presenza di un difetto, è necessaria una successiva attività di *debugging* per **individuare l'errore** che ha causato il difetto

Correzione (Fix):

- Una modifica ad un componente finalizzata alla riparazione di un difetto

Stub e driver:

- Gli *stub* e i *driver* di test consentono di isolare i componenti dal resto del sistema per i test:
 - **Test stub**: un'implementazione *parziale* dei componenti *da cui dipende il componente testato*
 - **Test driver**: un'implementazione parziale di un componente *che dipende dal componente testato* (e.g., una funzione che richiama la funzione testata passando gli input del caso di test)



Test Case

Un caso di test può essere descritto con una serie di **attributi**:

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

Correzioni

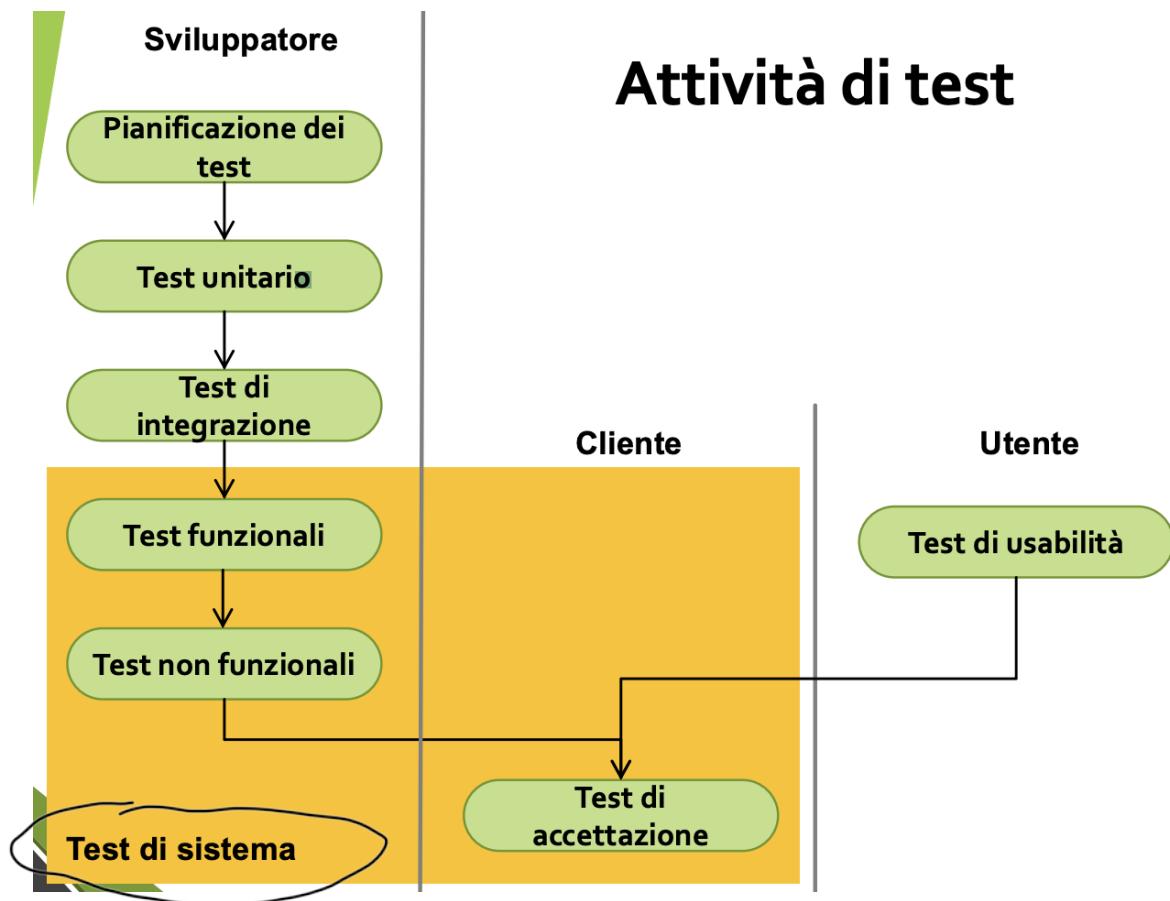
Una volta individuati i guasti, gli sviluppatori *modificano* il *componente* per eliminare i difetti che li causano:

- vanno dalla semplice modifica di un componente alla riprogettazione completa di una struttura dati o di un sottosistema
- la probabilità che lo sviluppatore introduca *nuovi difetti* nel componente revisionato è elevata.

Test di *regressione*:

- è la *riesecuzione di tutti i test precedenti* dopo una modifica
- in questo modo si garantisce che tutto ciò che funzionava prima della correzione non sia stato influenzato

6.2.2 Attività di test



- **Pianificazione dei test:**

- Si allocano le *risorse* e si *programmano* i test
- gli sviluppatori dovrebbero progettare i *casi di test* non appena i modelli che convalescono diventano stabili

- **Test di usabilità:**

- Cercano di trovare i difetti nella progettazione dell'*interfaccia utente* del sistema
- spesso i sistemi non raggiungono lo scopo prefissato semplicemente perché *gli utenti sono confusi dall'interfaccia utente*

- **Test unitari (*Unit test*):** cercano di rilevare i difetti dei componenti *presi singolarmente*. Normalmente sono svolti dagli stessi sviluppatori del componente.

- **Test di integrazione:** cercano di rilevare i difetti testando i *componenti in combinazione con altri componenti*

- **Test di sistema:** test di tutti i componenti *come unico sistema*.

Includono:

- test funzionali che verificano i requisiti funzionali; test non funzionali, verificano i requisiti non funzionali; test di accettazione, validano il sistema rispetto all'*oracolo* stipulato con il cliente del progetto.

6.3 Test unitari

Si concentrano sugli *elementi costitutivi* del sistema: **oggetti** e **sottosistemi**.

- I candidati per i test unitari sono scelti nel *diagramma delle classi*
- I *sottosistemi* devono essere testati solo dopo che ogni componente è stato testato singolarmente.

Vantaggi:

Riducono la complessità delle attività complessive di test, consentendo di concentrarsi su unità più piccole *Facilitano l'individuazione e la correzione dei difetti*, dato che i componenti coinvolti sono pochi

6.3.1 Tecniche per l'individuazione dei casi di tests

Black-box testing (detto anche *approccio funzionale* o *testing basato sulla specifica*: i casi di test sono definiti esclusivamente a partire dalla **specificità** del componente testato.

Techne di black-box testing:

- *Partitionamento in classi di equivalenza* e test di equivalenza.

Test di equivalenza:

- minimizza il numero di casi di test suddividendo i possibili input in **classi di equivalenza**
- Per i test *vengono utilizzati* solo *alcuni membri di ciascuna classe*
- l'ipotesi è che i *sistemi di comportino in modo simile per tutti i membri di una classe*

Il test di equivalenza si compone in **due passi**:

1. Identificazione delle *classi di equivalenza*.

Criteri per la determinazione delle *classi di equivalenza*:

- **Copertura**: ogni possibile input appartiene a una delle classi
- **Disgiunzione**: nessun input appartiene a più di una classe
- **Rappresentatività**: se l'esecuzione provoca un fallimento quando viene utilizzato un membro di una classe, lo stesso fallimento dovrebbe essere rilevato utilizzando qualsiasi altro membro

Per ogni classe, vengono selezionati almeno **due input**:

- (a) Un *input tipico*, che esercita il caso comune
- (b) Un *input non valido*, che esercita la gestione delle eccezioni

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

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

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

- Anni **bisestili** (2020, 2024, 2028, ecc.)
- Anni **non bisestili**

Valori non validi: numeri interi negativi

2. Selezione degli *input di test*: I casi di test si ottengono selezionando **un valore valido** per ogni classe di equivalenza, e combinando gli input.
- *Analisi dei valori limite e boundary testing*: il *test dei casi limite* è un caso speciale di *test di equivalenza*:
 - gli elementi sono selezionati dai confini tra le classi di equivalenza
 - gli sviluppatori spesso trascurano i casi speciali

White-box testing (detto anche *approccio strutturale*):

- i casi di test sono definiti usando la **specifiche** insieme alla conoscenza del **codice** del componente testato
- Ha l'obiettivo di garantire che tutte le parti del codice siano adeguatamente sollecitate dai casi di test (**copertura**)

Tecniche di *white-box* testing:

1. **Copertura delle istruzioni** (*statement coverage*): si scelgono i casi di test in modo da garantire che **tutti i nodi** del diagramma di flusso siano attraversati in almeno un caso di test
2. **Copertura delle diramazioni** (*branch coverage*): si scelgono i casi di test in modo da garantire che **tutti gli archi** del diagramma di flusso siano attraversati in almeno un caso di test (e.g., ogni If-Else sia coperto).
3. **Copertura dei percorsi** (*path coverage*): garantire che **tutti i percorsi** possibili del diagramma di flusso siano attraversati in almeno un caso di test È SÌ SOLO L.I.

Il punto di partenza è il **diagramma di flusso** (*control flow diagram*):

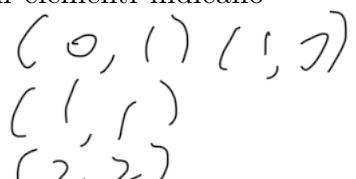
- Un diagramma di flusso è costituito da **nodi** che rappresentano blocchi eseguibili e da **archi** che rappresentano il flusso di controllo
- Come i *diagrammi delle attività*, costruiti **a partire dal codice**

Il criterio di *copertura* dei percorsi viene semplificato richiedendo di coprire tutti i **percorsi linearmente indipendenti**.

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

Un insieme di percorsi definisce linearmente indipendente se:

1. i vettori associati ai percorsi sono linearmente indipendenti
2. ovvero, nessuno dei vettori può essere ottenuto come combinazione lineare degli altri



Il *numero massimo di percorsi indipendenti* attraverso il diagramma di flusso si può ottenere mediante la **complessità ciclomatica**, definita come : $CC = E - N + 2$; dove E è il numero degli archi e N è il numero di nodi nel diagramma di flusso.

Back vs White box

Black Box:

1.

White Box:

1. aiuta a trovare errori legati al percorso di esecuzione all'interno del programma
→ *Miglior funzionalità*
2. non è in grado di rilevare *omissioni* nel componente testato

In generale, nessuna metodologia di test può garantire **la scoperta di tutti i difetti**.

6.4 Sviluppo guidato dai test (Test Driven Development)

In molti processi tradizionali, l'attività di testing viene svolta dopo l'implementazione del codice,

Il **Test Driven Development** (Sviluppo guidato dai test) è un metodologia di sviluppo del software in cui i test unitari vengono sviluppati **prima del codice**.

Il **TDD** si basa su tre prescrizioni:

- Test Early
- Test Often
- Test Automatically

1. Test Early

1. Quando si sviluppa una nuova funzione, per prima cosa si scrive la sua *interfaccia* con un'implementazione vuota
2. Si scrive il codice per testare la funzione (unit test), l'implementazione corrente (vuota) non dovrebbe superare il test
3. si aggiunge il codice per implementare la funzione, facendo in modo che superi i suoi test
4. si controlla che tutte le altre funzioni già esistenti superino i rispettivi test
5. solo a questo punto, si può considerare completo il lavoro per aggiungere la nuova funzione.

Vantaggi:

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

2. Test Often

Gli unit test non sono eseguiti una sola volta:

- ogni volta che il sistema viene modificato, **tutti** gli unit test sono **rieseguiti**
- il nuovo codice è integrato nel sistema solo se tutti gli unit test sono superati

Vantaggi:

1. Si scopre subito se la modifica effettuata ha compromesso il funzionamento di un'altra parte del codice
2. gli sviluppatori possono essere più tranquilli quando modificano il codice

3. Test *Automatically*

I test devono poter essere eseguiti **automaticamente**:

- un singolo comando deve poter lanciare tutti i test del progetto
- Gli sviluppatori devono ricevere un feedback sintetico con la possibilità di avere informazioni più dettagliate quando serve

Capitolo 7

Strumenti per il build ed il controllo delle revisioni

7.1 Debugging

Il *debugging* è il processo di identificazione e correzione di difetti di un programma software che ne impedisce il corretto funzionamento. **Tecniche:**

1. *Logging*: si usano le istruzioni di stampa per tracciare il flusso di esecuzione e monitorare i valori delle variabili in diversi punti del codice sorgente
2. *Debugger*: permette di esaminare l'esecuzione del programma e di capire come si comporta il codice:
 - **debugger da terminale**
 - **debugger integrato nell'IDE**: con funzionalità interattive come i punti di interruzione (*breakpoint*) e l'ispezione delle variabili

7.2 Version Control

I **sistemi di controllo versione** sono software per la *gestione di insiemi di file e directory nel tempo*:

1. tracciano ogni modifica apportata al testo, quando e da chi è stata effettuata
2. consentono di tornare indietro ad una versione precedente o confrontare diverse versioni
3. consentono di unire le modifiche apportate da diversi sviluppatori allo stesso codice sorgente

Storia:

- 1972: *SCCS*, disponibile su IBM Mainframe e successivamente su Unix. Richiedeva la codivisione del file system
- 1986: *CVS*, sistema client-server con controllo della concorrenza. Problema: blocco dei file durante la modifica
- 2000: *Subversion*, commit atomico. Supporto per i file binari, supporto delle occorrenze senza bloccare i file

- oggi: *Arch*, *Monotone*, *BitKeeper*, *Git*, *Bazaar*, basati su repository distribuite anziché centralizzati

7.3 Git

Creato da Linus Torvalds nel 2005 per lavorare sul kernel Linux. Il nome di Git deriva dal termine inglese "git", che significa *sciocco o spregevole*.

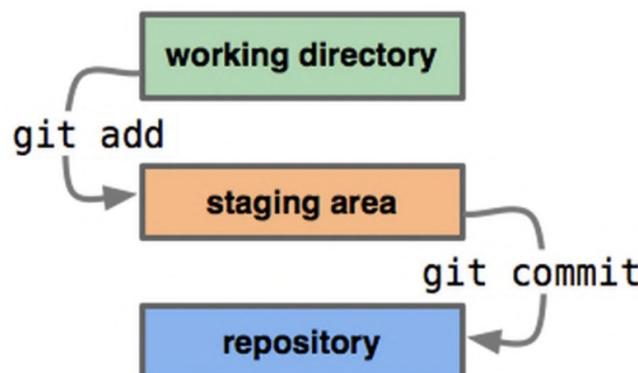
Git è stato creato con i seguenti obiettivi:

- Velocità
- Design semplice
- Supporto per lo sviluppo non lineare
- **Essere completamente distribuito:**
 - tutti i nodi conoscono la storia completa delle versioni
 - tutte le operazioni vengono fatte online
 - non esiste alcuna autorità centrale
 - le modifiche possono essere condivise anche senza server remoto
- Capacità di gestire in modo efficiente progetti di grandi dimensioni

7.3.1 Git: concetti fondamentali

Arearie di Git:

- *Directory di lavoro*: la directory effettiva del filesystem in cui si creano e si modificano i file
- *Area di Staging (indice)*: un'area intermedia in cui si preparano le modifiche prima di applicarle (**commit**)
- *Repository (head)*: l'area in cui Git memorizza in modo permanente la storia di tutte le modifiche apportate



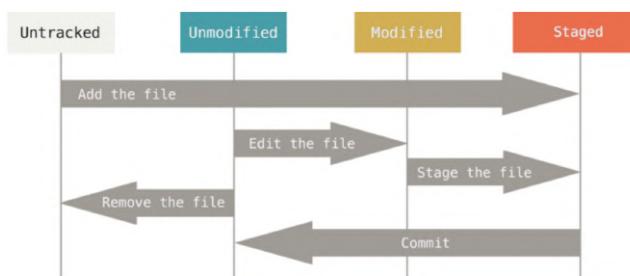
Stato dei file in Git

In Git un file può essere in uno degli stati seguenti:

- **Committed**: il file è memorizzato nel *repository*
- **Modified**: il file è stato modificato dall'ultimo **commit**. Probabilmente perché stiamo attualmente lavorando su questo file (nella *directory di lavoro*)
- **Staged**: abbiamo finito tutte le modifiche del file e lo abbiamo inserito nell'*area di staging*. Il file è ora pronto per essere aggiunto al repository con il prossimo **commit**

Altri possibili stati dei file sono:

- **Untracked**: un file nella *directory di lavoro* le cui modifiche non sono tracciate da Git. Se si inizia a tracciare le modifiche per quel file, esso passa automaticamente allo stato *Staged*
- **Unmodified (committed)**: un file della *directory di lavoro* che non è stato modificato dall'ultimo **commit**.



Storia del progetto

Nel repository, la storia del nostro progetto software è rappresentata da un **grafo**:

- I **nodi** sono creati dalle operazioni di *commit*.

Nota:

- è necessario effettuare il commit dei soli "file sorgenti" cioè i file non ottenibili automaticamente da altri file (.java).
- NON effettuare il commit dei file oggetto o eseguibili (.class, .exe)

- Ogni nodo contiene solo i file che sono stati modificati rispetto ai suoi antenati.

Si consideri un nodo come un **checkpoint** a cui è possibile facilmente tornare se qualcosa va storto

7.3.2 Git: Comandi principali

- **git init**: consente a git di prendere il controllo dei file nella directory corrente. Dopo aver effettuato *git init*, si genererà una **repository vuota**.

Si possono iniziare ad aggiungere file (*effettuando commit*) e utilizzare gli altri comandi di Git per gestire il progetto

- **git status**: mostra lo stato di tutti i file nella directory corrente

- **git add**: sposta un file controllato da git nell'*area di staging*.
 - **git add <file(s)>**: sposta file specifici
 - **git add .**: sposta i files che si trovano nella directory di lavoro
 - **git add <directory>**: sposta una directory specifica
- **git commit**: salva l'area di staging nel repository, **creando un nuovo nodo**. **git commit -m "Commit message"**: esegue il commit delle modifiche contenute nell'area di staging aggiungendo un messaggio descrittivo
- **git diff**: mostra le differenze tra la propria directory di lavoro e l'ultimo commit
- **git log**: visualizza la cronologia dei commit, mostrando id, autore, data e messaggi di commit
- **git reset <file(s)>**: rimuove un file dall'area di staging. L'opposto di **git add**
- **git revert <commit id>**: crea un nuovo *commit* che annulla le modifiche introdotte da un altro commit. Opposto a **git commit**

Esempio di utilizzo:

```
mkdir CoolProject
cd CoolProjectn
git init
notepad README.txt
git add .
git commit -m 'my first commit'
```

7.3.3 Branch e Merge

In git, i rami (*branch*) consentono di lavorare contemporaneamente su **diverse versioni** del progetto:

- Ogni ramo rappresenta una *linea di sviluppo indipendente*
- Utile per esplorare una nuova idea o risolvere un bug
- è possibile **creare** nuovi rami, **passare** da uno all'altro e **unirli**

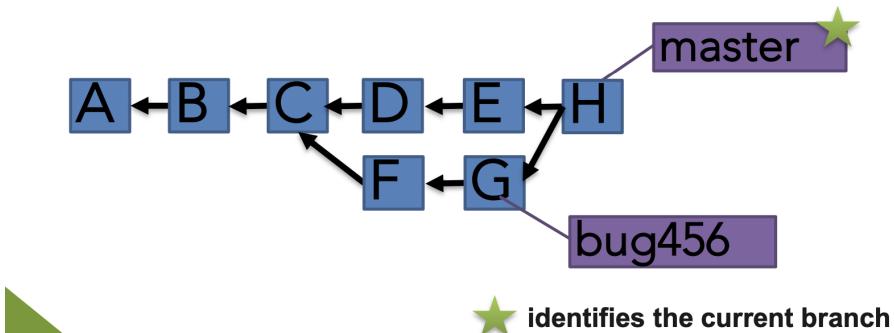
Il **ramo corrente** è il ramo su cui stiamo lavorando: tutti i commit sono collegati al ramo corrente.

Comandi:

- **git checkout -b bug123**, cambia il ramo corrente. L'opzione **-b** crea un nuovo ramo
- **git checkout master**, torna al ramo master.
- **git merge bug123**, unisce il ramo corrente con un altro, applicando le modifiche appropriate all'altro ramo.
- **git branch -d bug123**, l'opzione **-d** consente la cancellazione di un ramo.

> **git merge bug456**

Aggiunge un nuovo nodo per unire i due rami, ma solo se **non ci sono conflitti** tra i rami.



Gestione dei conflitti

L'unione potrebbe **non essere completata** se ci sono conflitti tra i rami:

- file di testo che differiscono per parti sovrapposte
- file binari diversi

Se l'unione non riesce e viene segnalato un conflitto, lo **sviluppatore deve**:

1. modificare i file contenenti i conflitti decidenti quali modifiche mantenere o cambiare
2. seguire il **commit** dei file con i conflitti risolti

7.4 Git e NetBeans

Cagate

7.5 Git Remote

Un **remote** è un repository *ospitato su un server*:

- è identificato da un URL
- può essere ospitato da piattaforme cloud come GitHub, GitLab o Bitbucket
- consente lo sviluppo collaborativo e la condivisione del codice

Git dispone di comandi per **sincronizzare** i *rami* del repository locale con il repository remoto.

Per aggiungere un remote ad un repository locale esistente: **git remote add origin <https://github.com/coolproject>**

- **origin** è il nome dato al remote (è obbligatorio)
- un repository locale può anche essere collegato a più remote

Per creare una copia di un repository da un remote: **git clone <https://github.com/coolproject>**

- da usare se non si dispone di una copia locale del repository

Comandi principali:

1. `git fetch origin`: recupera le modifiche da un remote remoto senza unirle
2. `git pull origin <nome del ramo>`: recupera le modifiche da un repository remoto e le unisce ad un ramo
 - pull = fetch + merge
3. `git push origin <nome del ramo>`: carica i commit locali su un remote aggiornando un ramo remoto
 - NOTA: git rifiuterà il push se esistono modifiche più recenti sul server remoto
 - BUONA PRATICA: prima effettuare pull, poi push

Suggerimenti per collaborare

- Non effettuare mai commit di codice con errori (che non viene compilato o che non supera tutti i test)
- Evitate di aggiungere file non contenenti codice sorgente a meno che non siate sicuri che non verranno modificati.
- Assicurarsi che tutti i file necessari alla compilazione siano nel commit
- Non includere nessun file non necessario!
- Effettuare spesso pull e push (più volte lo si fa, meglio è)

Capitolo 8

Build Automation

Insieme delle attività necessarie a produrre tutti i file necessari all'esecuzione del programma in un determinato ambiente di esecuzione.

Il build di un progetto complesso potrebbe includere:

- individuazione dei **file sorgenti** da compilare
- **Compilazione** dei file sorgente necessari
- **Link** del file oggetto e delle librerie
- Compilazione dei **test automatizzati**
- **Esecuzione** dei test automatizzati
- **Packaging** dei file eseguibili e degli altri file necessari

Build automation

Uso di strumenti che consentono di eseguire tutte le azioni necessarie per il build *senza intervento umano*. Vantaggi:

- **Completezza**: il tool assicura di aver inserito tutti gli elementi necessari
- *Ripetibilità*: eseguendo più volte il build si ottengono gli stessi risultati
- **Schedulabilità**: è possibile eseguire il build automaticamente in momenti prestabiliti
- **Portabilità**: il tool consente l'esecuzione del build allo stesso modo su piattaforme diverse

8.1 Build automation in C con make