

Riassunto IS Completo

Marco Briglia 2024/2025

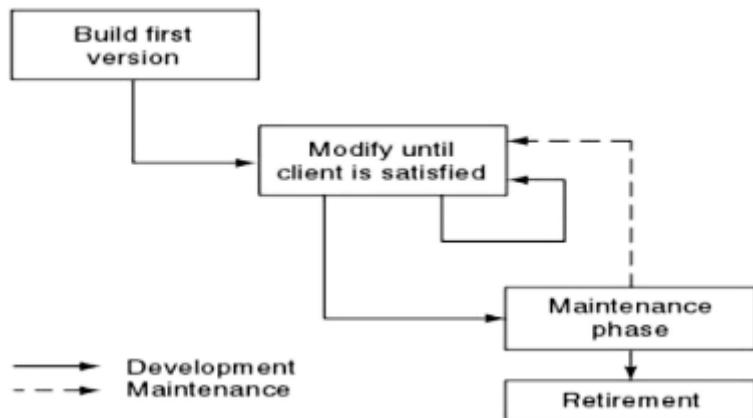
Capitolo 1

Processo Software

Il *processo software* è il percorso per sviluppare un prodotto o sistema software. Per modellare un sistema software bisogna definire un *modello di ciclo di vita*, cioè un modo di operare e suddividere il processo in attività con regole ben definite.

Build-and-Fix Model

Si scrive un programma e si sistema piano piano fino a quando non soddisfa il cliente (fa schifo)



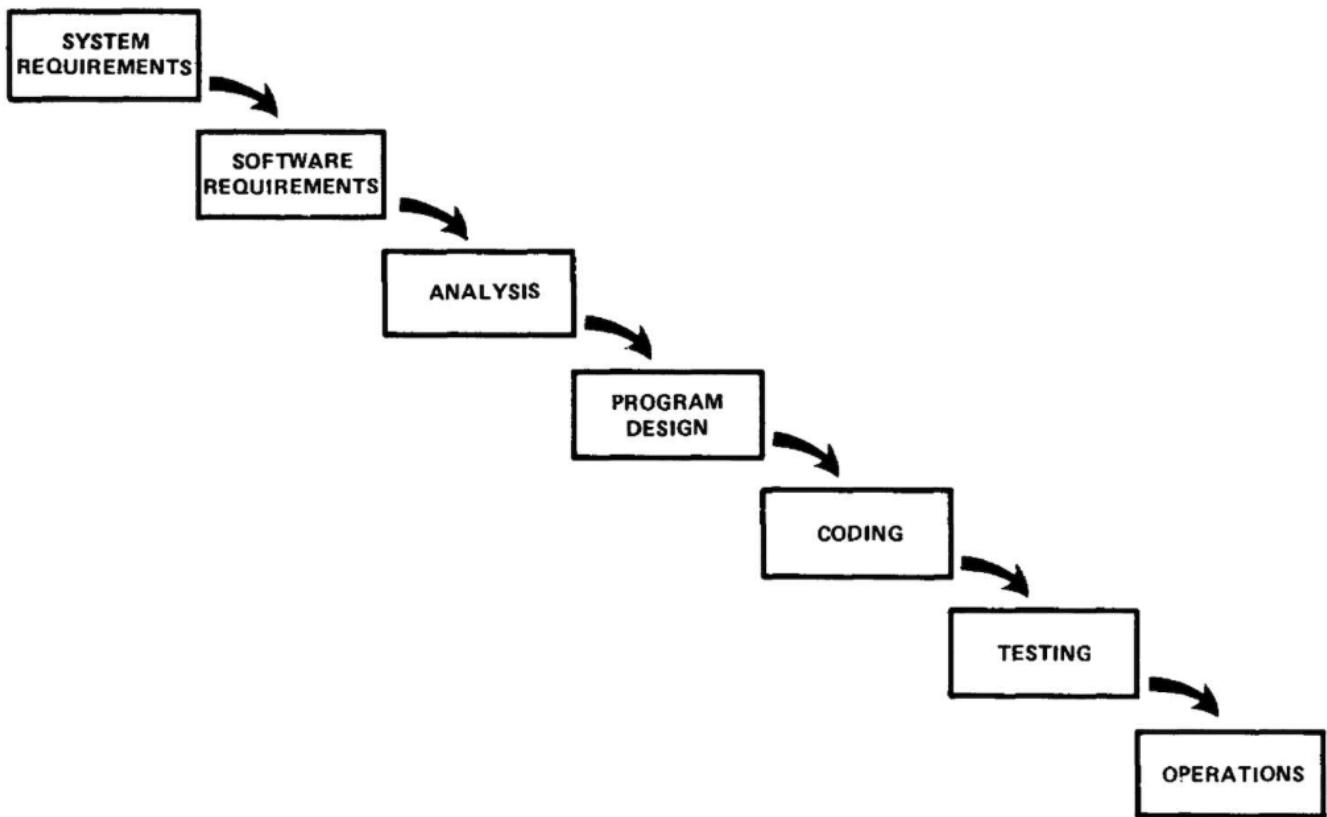
Cascata

Il valore di questo modello è stato quello di distinguere e definire le fasi di un processo software.

Il modello richiede che il passaggio a una nuova fase sia possibile solo dopo il completamento della precedente e sia revisionato un documento che ne attesta le funzionalità.

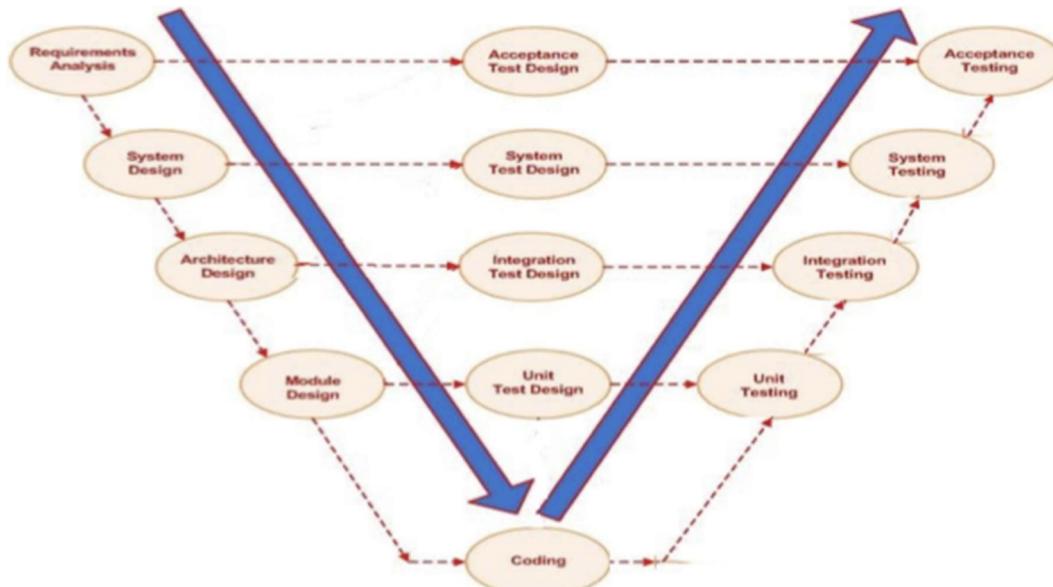
Qua nascono i suoi difetti per cui è stato aspramente criticato:

- Manca *l'interazione col cliente* che vede solo il prodotto finito, alla fine del processo.
- *Eccessiva produzione di documenti*.



Modello a V

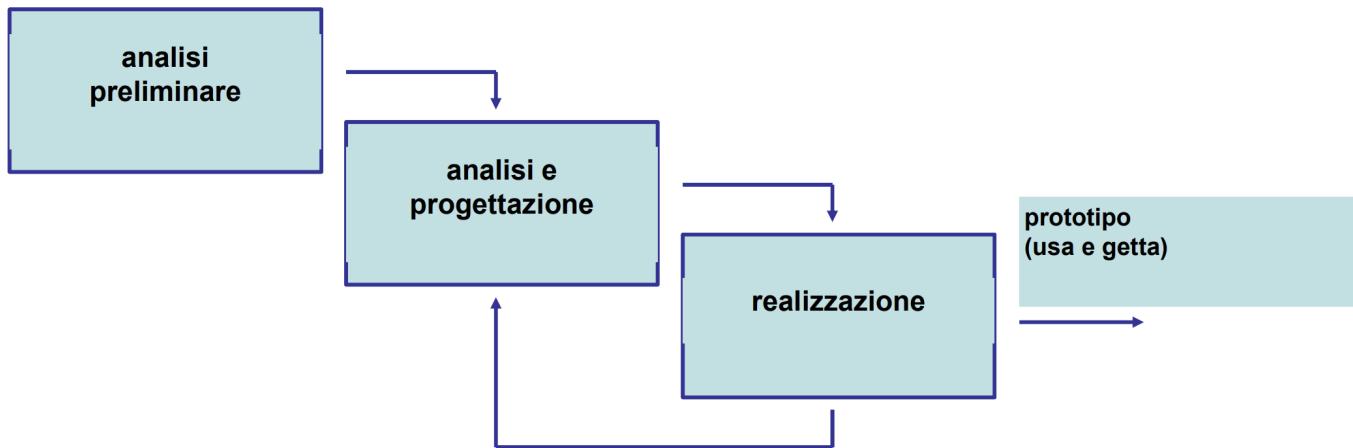
Evidenzia come sia possibile *progettare i test durante le fasi di sviluppo*. Le attività di test sono parte del processo di sviluppo del software



Rapid prototyping

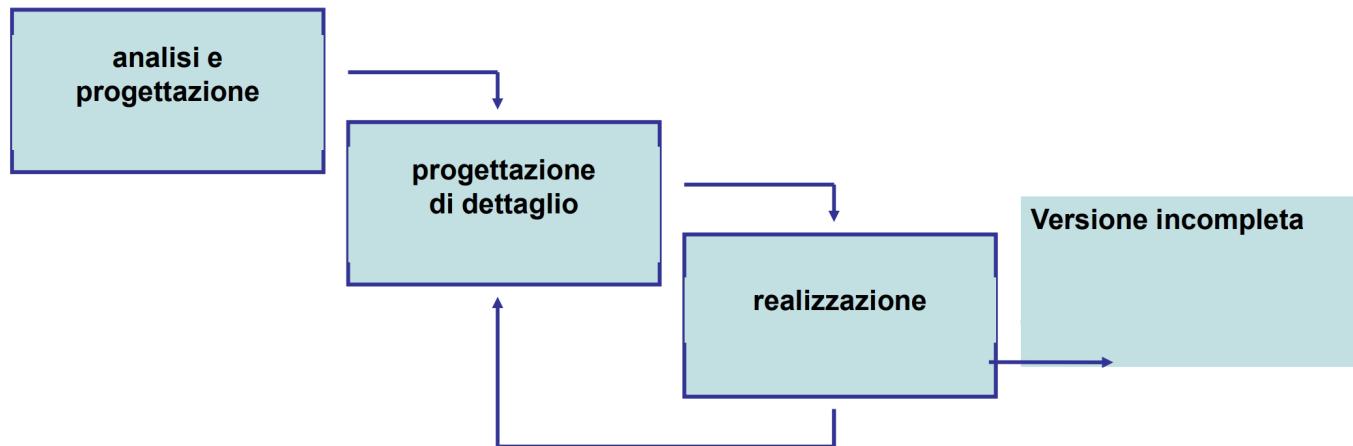
Lo scopo è quello di costruire velocemente un prototipo per permettere al committente di sperimentarlo.

Si usa questo modello quando i *requisiti* del cliente non sono chiari in modo da definirli meglio mostrandogli il processo di sviluppo.

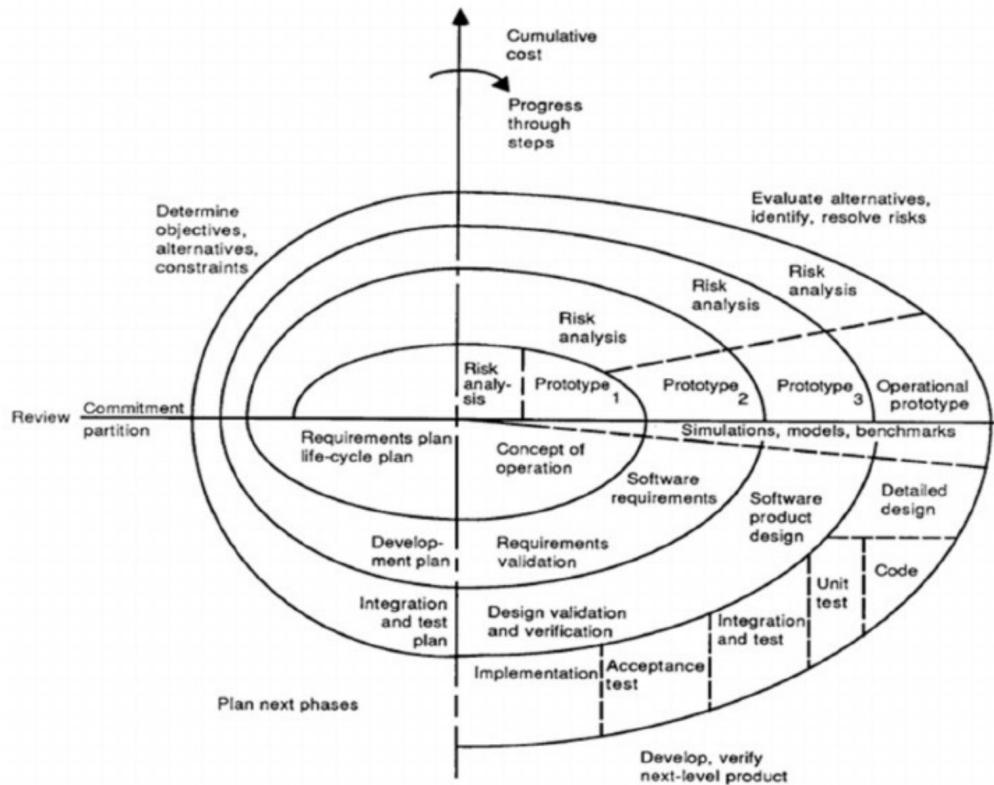


Modello incrementale

Il sistema è costruito iterativamente aggiungendo nuove funzionalità. Serve quando bisogna far "uscire velocemente" un prodotto sul mercato avendo ben chiaro i requisiti del sistema.



Modello a Spirale



Si ripetono 4 fasi ciclicamente fino al termine dello sviluppo:

1. Determinazione di obiettivi, vincoli e alternative
2. Valutazione di rischi e alternative
3. Sviluppo e verifica
4. Pianificazione della fase successiva

Agile

Per Agile, *l'interazione* con le persone è più importante dei processi, bisogna collaborare con il cliente perché la collaborazione diretta offre risultati migliori.

È più importante avere *software funzionante* che documentazione.

Bisogna mantenere il codice *semplice* e *avanzato tecnicamente*.

Bisogna rilasciare versioni del software frequentemente in modo da raccogliere *Feedback*.

Principi di Agile:

- *Continuous Integration*: rendere il più possibile automatico il processo di consegna e integrazione dei moduli.
- *Continuous Delivery*: Rilascio frequente e supportato da strumenti di nuove versioni del sistema software.

- *DevOps*: “Development” e “Operations”. Se scrivi il codice, lo revisioni.

Scrum

Scrum è un modello iterativo e incrementale per lo sviluppo e la gestione di prodotti, che offre funzionalità potenzialmente rilasciabili alla fine di ogni iterazione (*Sprint*).

Il processo si articola in tre fasi:

1. *Pre-game*: creazione della *Product Backlog List* con i requisiti conosciuti.
2. *Game*: sviluppo del sistema tramite Sprint.
3. *Post-game*: preparazione del prodotto per il rilascio.

Ruoli in Scrum:

- *Product Owner*: rappresenta il cliente e coordina i requisiti.
- *Development Team*: squadra di circa 7 persone che realizza il prodotto.
- *Scrum Master*: facilitatore che supporta il team senza esercitare autorità.

Eventi di uno Sprint:

1. *Planning*: definizione dello Sprint Backlog con il Product Owner.
2. *Implementation*: lavoro sul prodotto, con daily meeting per aggiornamenti.
3. *Review*: ispezione del software con feedback degli utenti.
4. *Retrospettiva*: analisi per migliorare gli Sprint futuri.

Il *WIP limit* è un limite al numero di task per colonna nel kanban (Lavagnetta dei post-it), utile per ridurre il task switching e velocizzare il completamento dei lavori.

Capitolo 2

Studio di fattibilità

Fase preliminare che consiste nel valutare l’opportunità o meno di realizzare il sistema software.

Esso si basa su una descrizione sommaria del sistema e di quali sono le necessità dell’utente.

Coinvolge un'*analisi di mercato* per valutare i costi di sviluppo e produzione e un'*analisi tecnica* per capire se il progetto in sé sia realizzabile.

Analisi dei Requisiti

Processo di studio e analisi delle esigenze del committente e dell'utente per giungere alla *definizione del dominio del problema e dei requisiti del sistema*, posti all'interno di una scala di priorità. Inoltre, dominio e requisiti del sistema devono essere propriamente documentati in documenti formali.

Dominio

Si intende il del sistema da realizzare.

Si stabilisce:

- Un *glossario* di termini rilevanti per il dominio considerato al fine di eliminare ambiguità.
- Un *modello statico* per descrivere la **struttura generale del dominio**. In esso compaiono tutte le entità coinvolte e le relazioni tra esse. Si usano spesso linguaggi grafici come UML.
- Un *modello dinamico* per ogni contesto, il quale descrive la **sequenza di azioni che devono essere svolte**. È spesso realizzato tramite descrizione testuale in linguaggio naturale.

Requisiti

Un requisito è una *proprietà* che il sistema deve garantire per soddisfare una necessità di un utente o del committente.

I requisiti si dividono in due categorie principali:

- *Requisiti funzionali*: Descrivono le **funzionalità del sistema**, indicando come deve comportarsi, reagire agli input e le azioni da eseguire. Esempi: (Fornire un visualizzatore per documenti memorizzati, consentire ricerche su basi di dati complete o parziali).
- *Requisiti non funzionali*: Descrivono **proprietà relative ai servizi o alle funzioni**, come: efficienza, affidabilità, sicurezza, robustezza, linguaggio di programmazione, metodi...

Documentazione in Linguaggio naturale

Il *documento dei requisiti* è un contratto tra sviluppatore e committente che elenca le funzionalità e i vincoli del sistema, specificandone anche la deadline. Un buon requisito è chiaro e conciso, evitando ridondanze e dettagli tecnici.

L'analisi dei requisiti si articola in cinque passi:

- *Acquisizione* (interviste strutturate o non)
- *Elaborazione* (bozza del documento)
- *Convalida* (correzione di difetti come omissioni o ambiguità)
- *Negoziazione* (priorità basate su esigenze e costi/tempi)
- *Gestione* (identificatori unici, attributi come stato, rischio e tracciabilità)

Il sistema di gestione **MoSCoW** prevede la suddivisione dei requisiti in quattro livelli di priorità:

- *Requisiti obbligatori*: irrinunciabili per il cliente.
- *Requisiti desiderabili*: non necessari ma utili.
- *Requisiti opzionali*: relativamente utili, da realizzare se c'è tempo.
- *Requisiti postponibili*: contrattabili per successive versioni del sistema.

User Stories

Capitolo 4

Modelli

Un modello è un'astrazione del sistema che ne specifica *struttura* e/o *comportamento*.

Un modello può essere:

- *Statico*: descrive il sistema a livello di struttura e componenti. Ne definisce entità e relazioni ma non ha in alcun modo aspetti dipendenti dal tempo.
(EX. Semaforo descritto come oggetto)
- *Dinamico*: descrive il sistema e la sua evoluzione nel tempo. Descrive i vari stati in cui il sistema può trovarsi nel tempo e le varie transizioni di stato.
(EX. Transizione da rosso a verde, verde a giallo, giallo a rosso)

Un modello può essere rappresentato in diverse forme:

- Bozza (Progetto sketch)
- Blueprint (Progetto dettagliato)
- Eseguibile (Progetto praticamente completo)

Diagramma dei casi d'uso

Describe i *requisiti funzionali* e il *modello statico* del sistema, catturando le funzionalità che deve offrire .

Un *attore* è un'entità esterna al sistema, che interagisce direttamente con esso in un determinato ruolo.

Ci sono 3 tipologie di attori:

- Utente umano in un determinato ruolo
- Altro sistema
- Tempo (attore speciale)

Un *caso d'uso* è una funzionalità o un servizio offerto dal sistema a uno o più attori.

Uno *scenario* è una sequenza di interazioni (scambi di messaggi) tra sistema e attori.

Il diagramma dei casi d'uso è composto da:

- *Attori*: hanno un nome.
- *Casi d'uso*: hanno anch'essi un nome.
- *Relazioni tra attori e casi d'uso*: rappresentano un tipo di interazione, intesa come sequenza di messaggi. Ogni caso d'uso può essere iniziato da un solo attore, il quale assumerà il ruolo di attore principale, mentre gli altri saranno attori secondari. Possono anche esservi casi d'uso dove l'unico attore associato ad essi è il tempo.
- *Confine del sistema*: rettangolo disegnato intorno ai casi d'uso per indicare il confine del sistema (oggetto del modello).

Tra gli *attori* ci possono essere *Relazioni si sottotipo*.

È possibile definire *attori astratti*, i quali possono solo essere padre di altri attori concreti figli.

Anche tra i **casi d'uso** possono esserci relazioni di sottotipo.

L'inclusione e l'estensione sono relazioni tra casi d'uso:

- **Inclusione**: Il caso d'uso 1 include il caso d'uso 2 quando il primo incorpora l'interazione descritta dal secondo. È simile a una chiamata di funzione, dove il caso 1 invoca l'esecuzione del caso incluso. Si rappresenta con una freccia tratteggiata accompagnata dallo stereotipo «include».
(EX. Per prendere in prestito un libro o estendere il prestito, è necessario seguire un'interazione comune.)
 - **Estensione**: Il caso d'uso 1 è esteso dal caso d'uso 2 quando il primo può incorporare l'interazione del secondo solo in determinate circostanze, senza dipenderne obbligatoriamente.
(EX. Creare un corso può includere, opzionalmente, la prenotazione di un'aula.)
-

Capitolo 5

Un *Oggetto* è un entità caratterizzata da **identità, stato, comportamento**.

Una *Classe* descrive un insieme di oggetti dello stesso tipo.

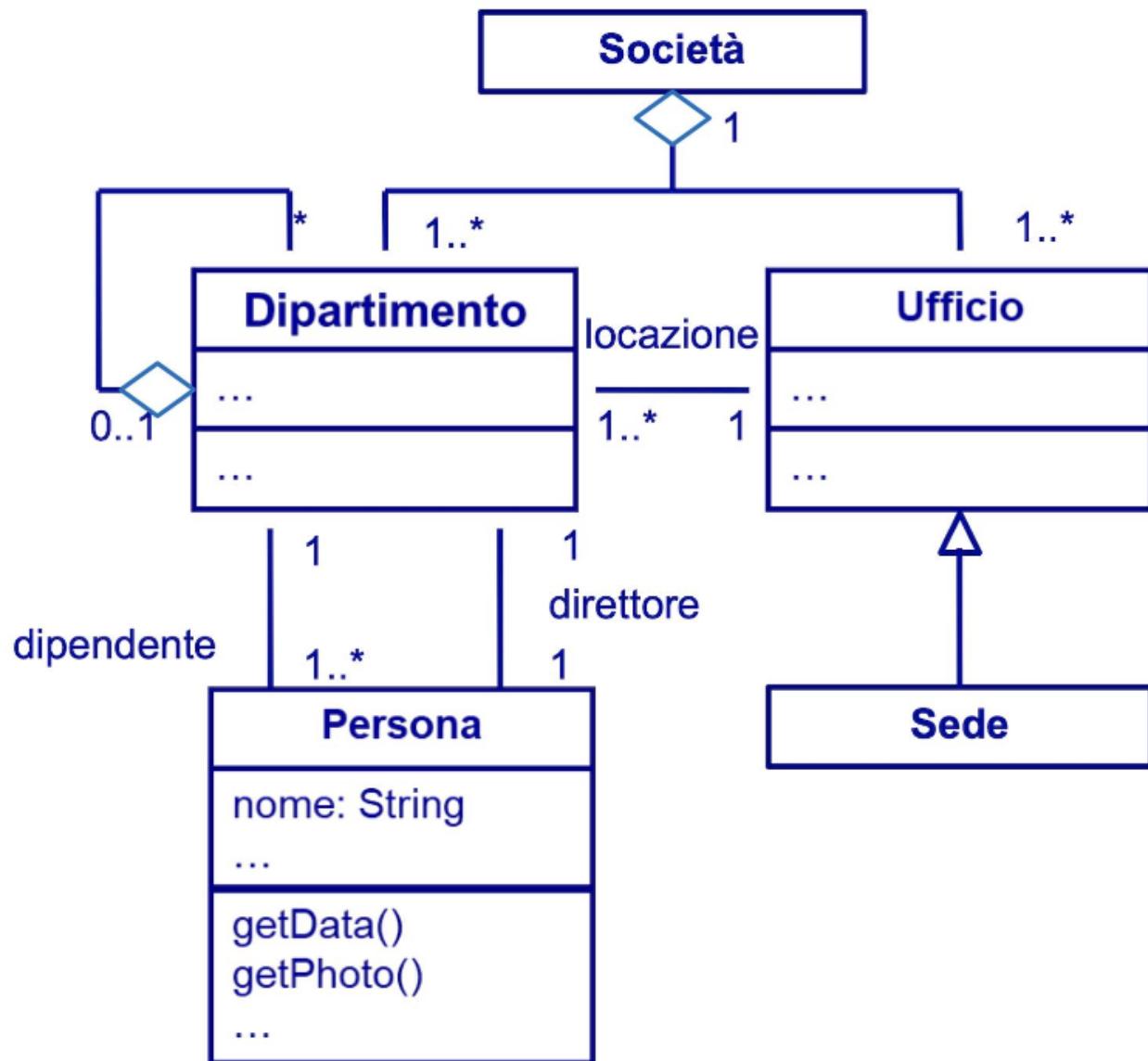
Diagramma delle Classi

Il diagramma delle classi descrive:

- *Tipo degli oggetti*.
- *Relazioni statiche* fra di essi.

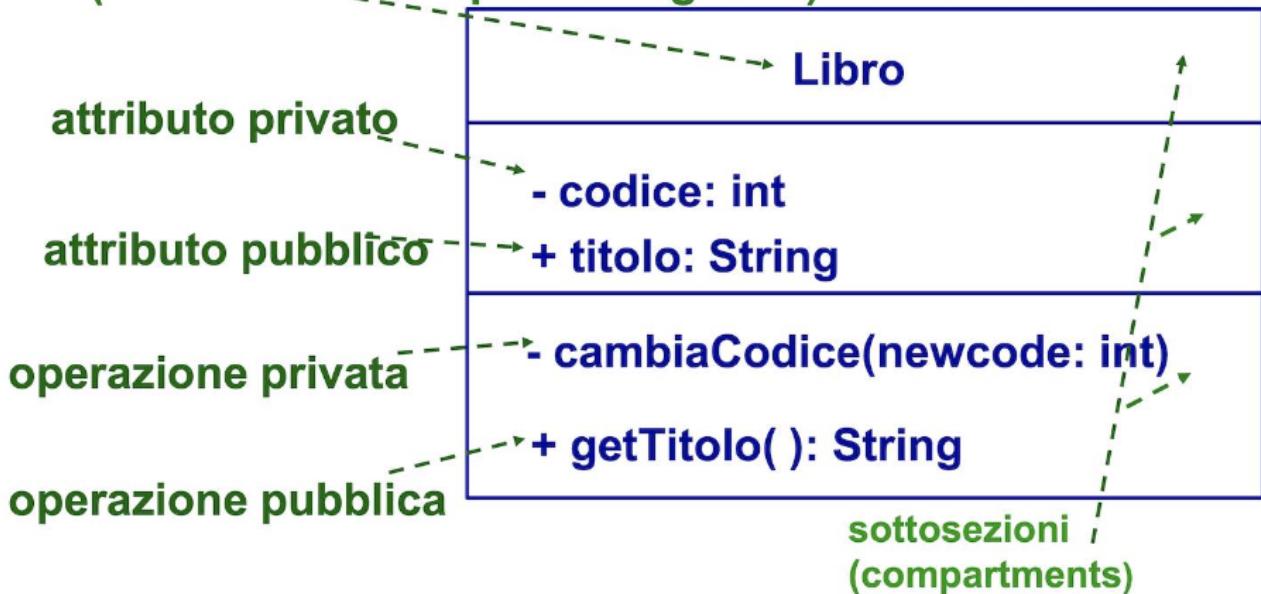
EX.

Una società è formata da dipartimenti e uffici. Un dipartimento ha un direttore e più dipendenti, ed è situato in un ufficio. Esiste una struttura gerarchica dei dipartimenti. Le sedi sono uffici.



Una classe si rappresenta così:

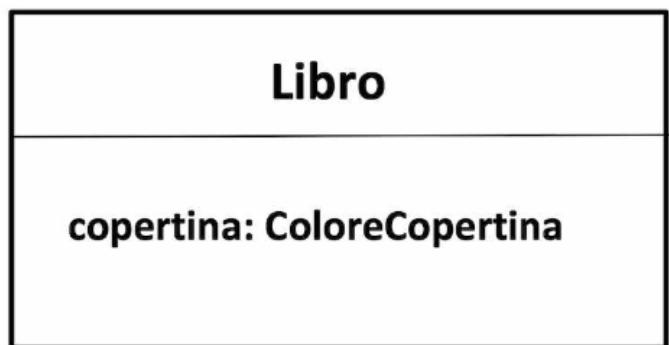
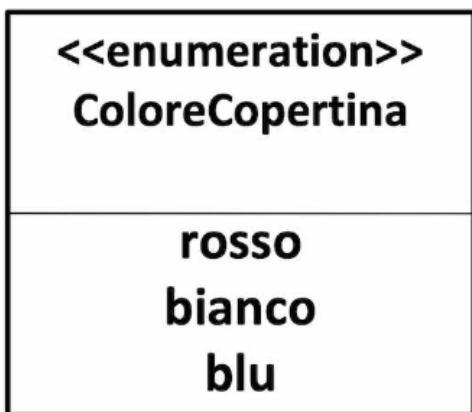
nome (maiuscolo e sempre al singolare)



I tag di visibilità possono essere:

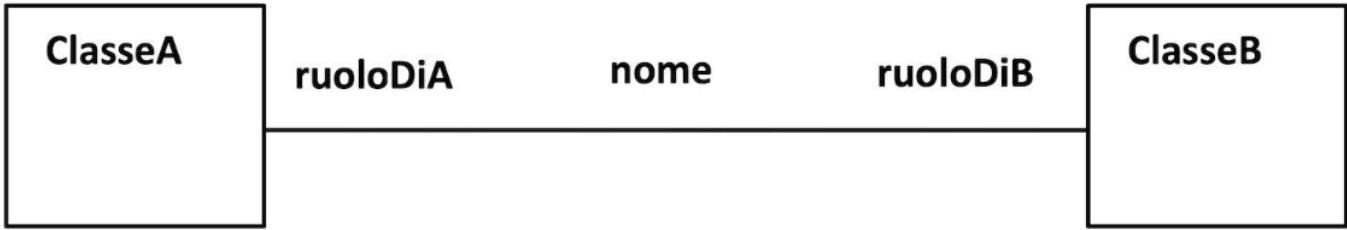
- **+ (public)**: accessibile ad ogni elemento che può vedere e usare la classe.
- **- (private)**: solo le operazioni della classe possono vedere e usare l'elemento in questione.
- **# (protected)**: accessibile ad ogni elemento discendente.
- **~ (package)**: accessibile solo agli elementi dichiarati nello stesso package.

L'*Enumerazione* è la lista completa di tutti i valori che gli attributi di un determinato tipo possono assumere.

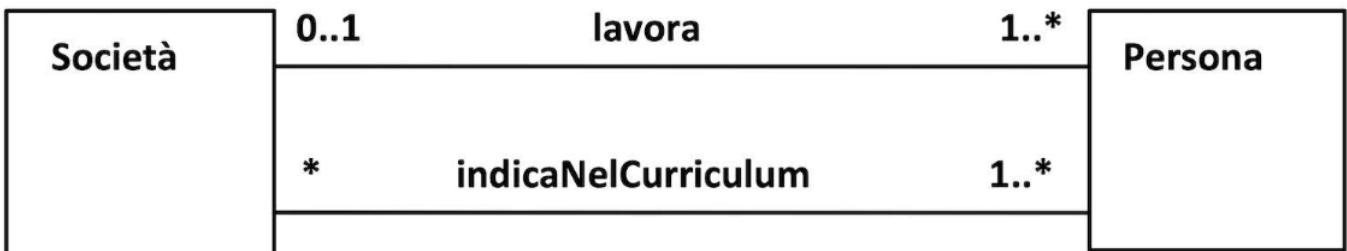


Relazione

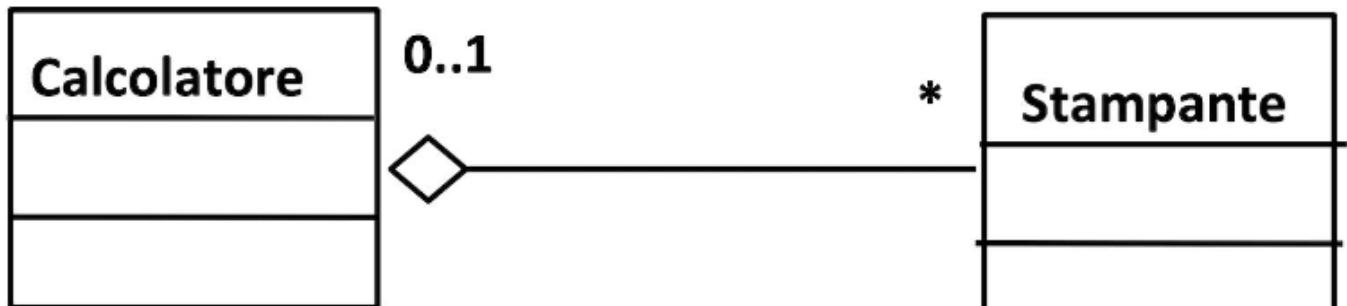
Una *Relazione* rappresenta un legame tra due o più oggetti, spesso di classi diverse. Essa è rappresentata con una linea retta, etichettata con nome (minuscolo, spesso un verbo), tra le classi degli oggetti coinvolti.



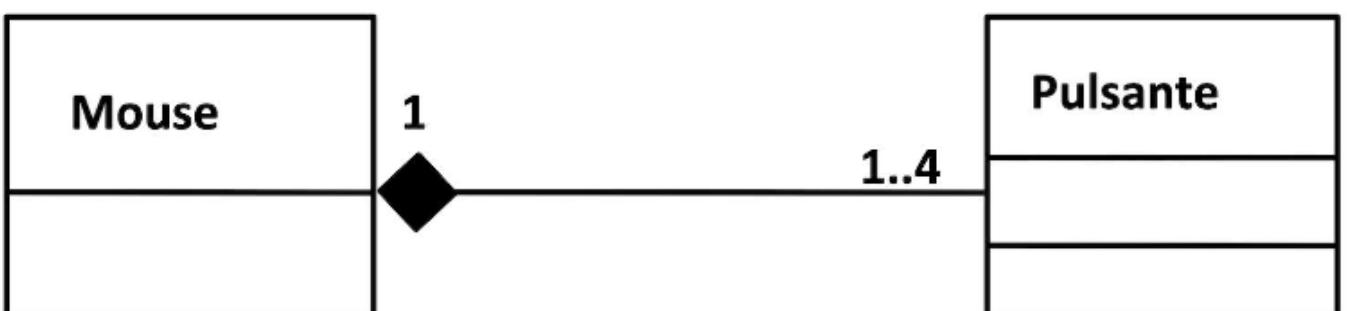
La *Molteplicità* indica il numero di oggetti coinvolti nella relazione in un dato istante.



L'*Aggregazione* è una relazione debole tra oggetti, ovvero una relazione nella quale le classi parte hanno un significato anche in assenza della classe tutto.

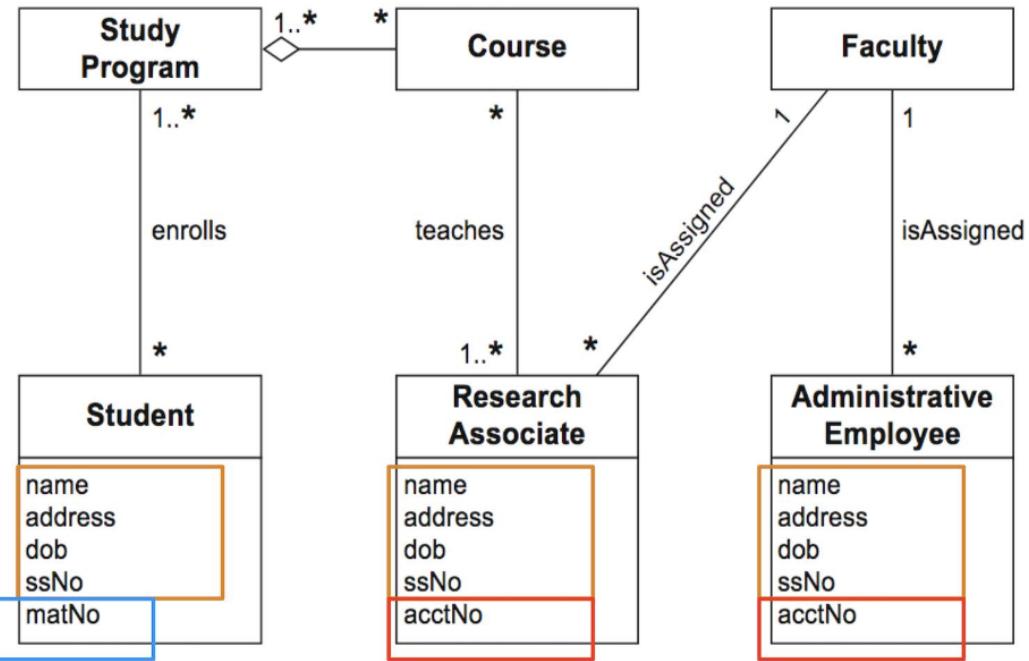


La *Composizione* è una relazione forte tra oggetti, ovvero una relazione nella quale le classi parte hanno un reale significato solo se sono legate alla classe tutto.

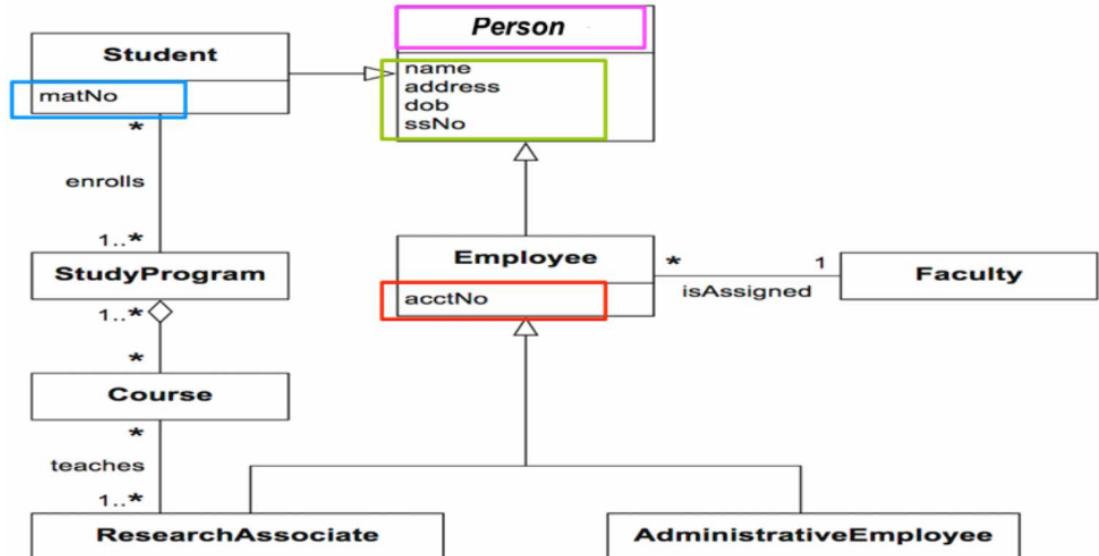


La **Generalizzazione** è una relazione di sottotipo fra due relazioni.

Le sottoclassi ereditano tutte le caratteristiche della superclasse (attributi, vincoli, relazioni, operazioni).



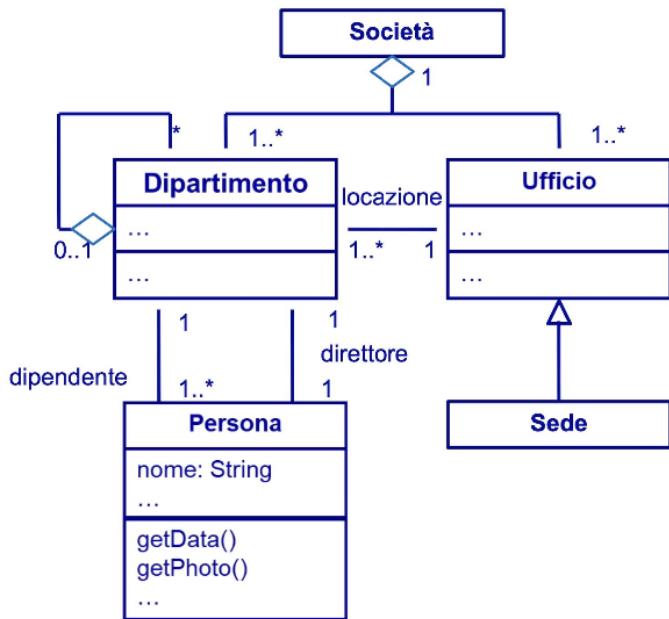
possia essere compattato, sfruttando la generalizzazione, nel seguente modo:



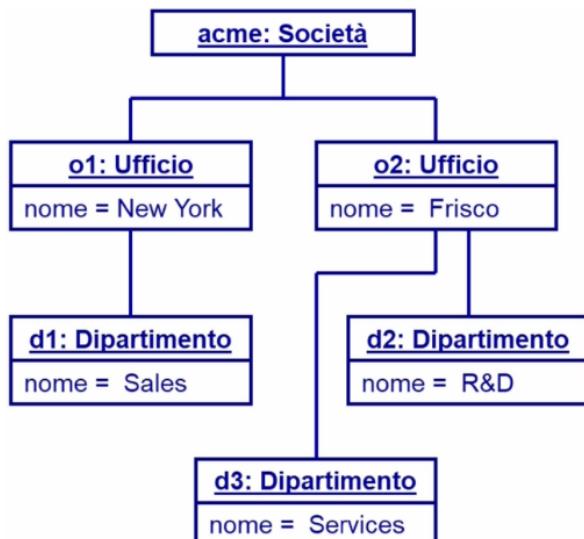
Si possono anche definire **Interfacce** che si usano in fase di progettazione per classi con solo comportamento e senza stato. Si inserisce un tag `<<Interface>>`.

Diagramma degli Oggetti

Utile quando le connessioni tra gli oggetti sono complicate e non si discernono bene dal diagramma delle classi.



: diagramma degli oggetti



Un *Oggetto* si rappresenta così:

nomeoggetto: nomeclasse

attr1: tipo = valore

attr2: tipo = valore

attr3: tipo = valore

Capitolo 7

Diagramma di Attività

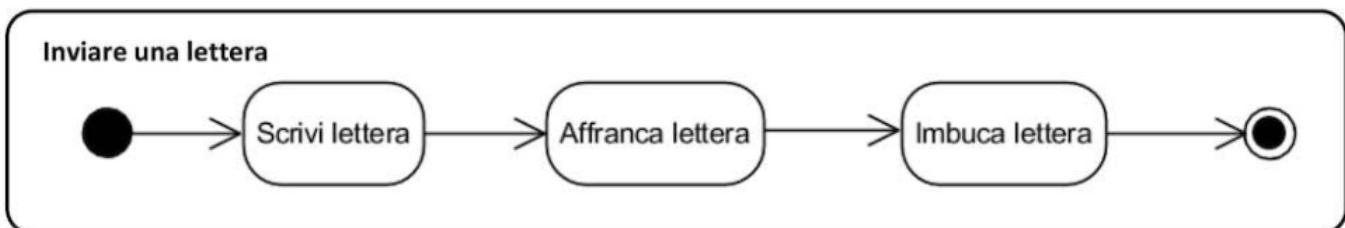
Un' **Attività** descrive la coordinazione di un *insieme di azioni*.

Un **diagramma delle attività** offre un modello di rappresentazione di un'attività.

In un diagramma delle attività, un'attività ha un nome ed è rappresentata con un grafo diretto, in cui:

- I **Nodi** rappresentano le componenti dell'attività (inizio, fine, etc.).
- Gli **Archi** rappresentano il **control flow**, cioè i possibili path seguibili per l'attività.

Nome e Grafo sono contenuti all'interno di un rettangolo dagli angoli smussati.



Le azioni sono rappresentate anche esse da rettangoli con angoli smussati.

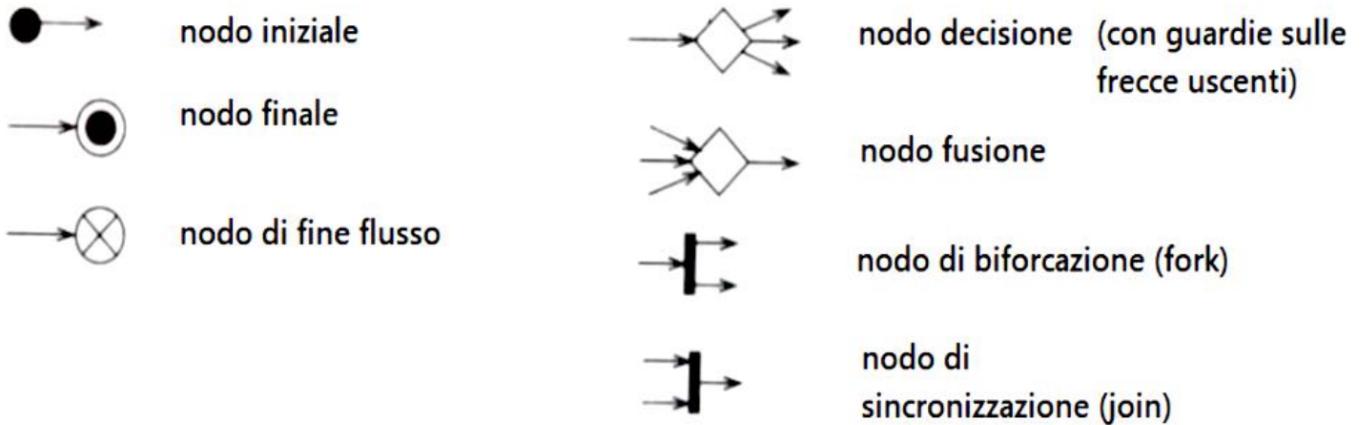
Caratteristica fondamentale delle azioni è la loro **atomicità**.

Ogni azione può avere solo una freccia entrante ed una uscente.

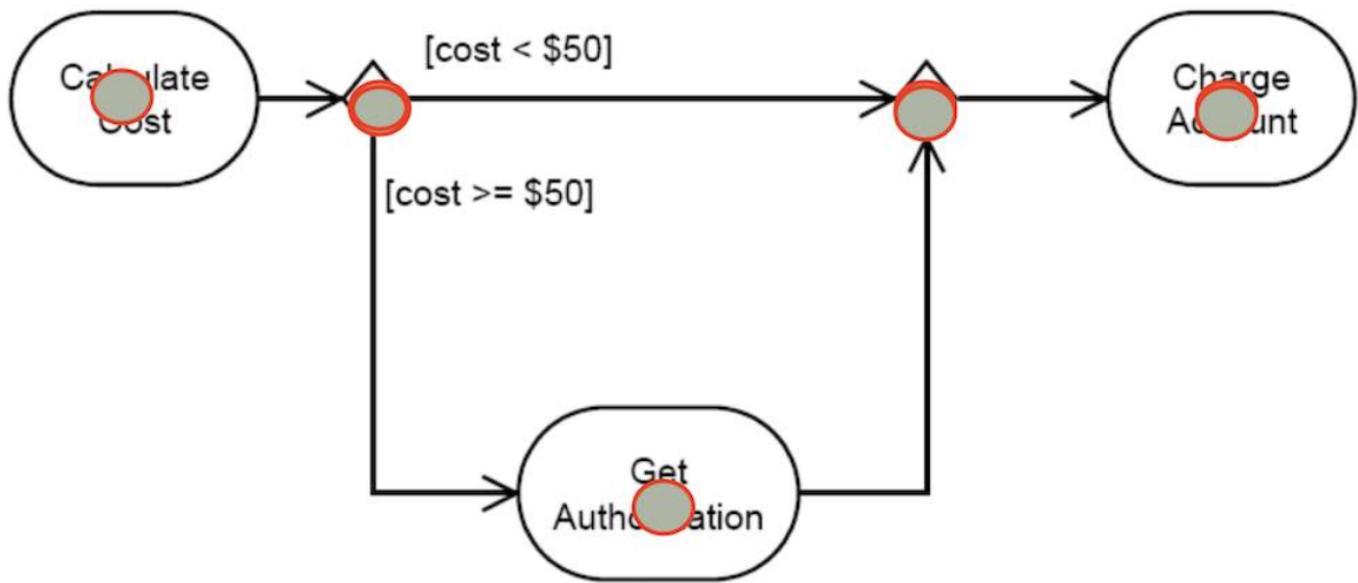
Quando un'azione ha terminato il proprio lavoro, scatta una *transizione automatica* che porta all'azione successiva.

Ogni azione viene eseguita se e solo se riceve un *token*, che, quando terminata, passa a sua volta sull'arco uscente.

I Nodi del grafo, oltre ad azioni, possono essere anche *nodi di controllo*:



Gli Archi possono essere etichettati con una *guardia* che decreta se il token può passare o meno l'arco. La guardia deve avere *copertura* di tutti i casi possibili. Deve esserci sempre una strada percorribile per il token.



Si possono creare *Cicli* se e solo se esiste un modo per uscirne data una certa condizione.

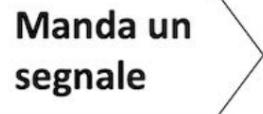
Un'attività può spesso essere *condizionata da eventi esterni* che ne modificano il flusso delle azioni.

Vi sono quindi nodi specializzati nell'*invio asincrono* e *ricezione di segnali*.

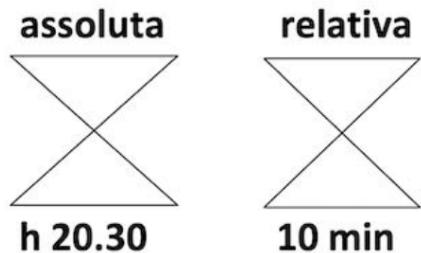
■ Accettazione di evento esterno



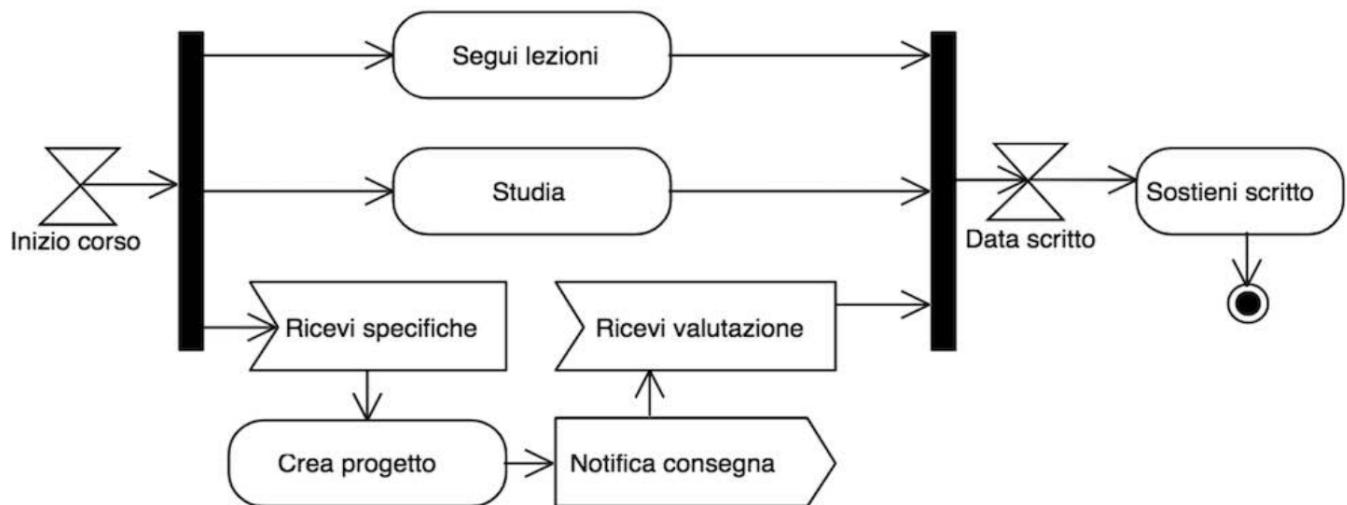
■ Invio di un segnale



■ Accettazione di evento temporale

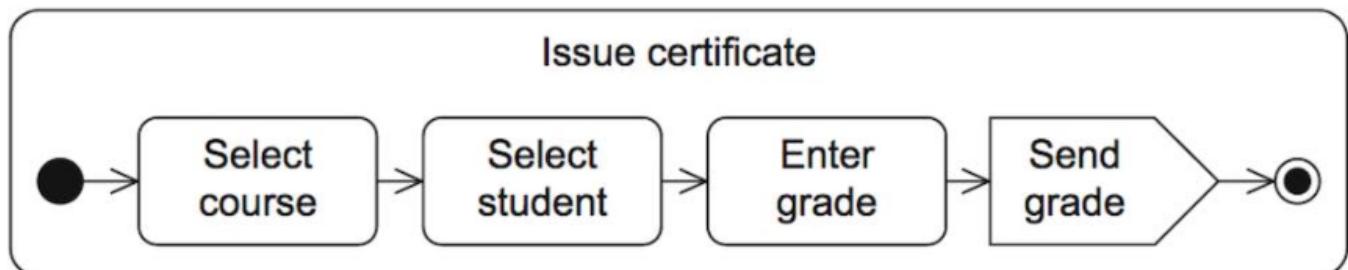
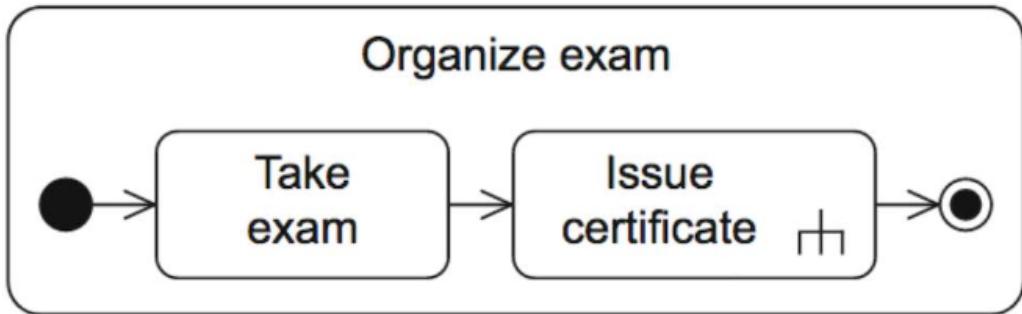


EX. Riassuntivo



Sottoattività

Un'azione può includere un'altra attività secondaria. In questo caso, la *sotto-attività* viene descritta in un diagramma a parte.



Una *partizione* divide le azioni di una attività in gruppi. Spesso, lo scopo è quello di assegnare responsabilità alle azioni.

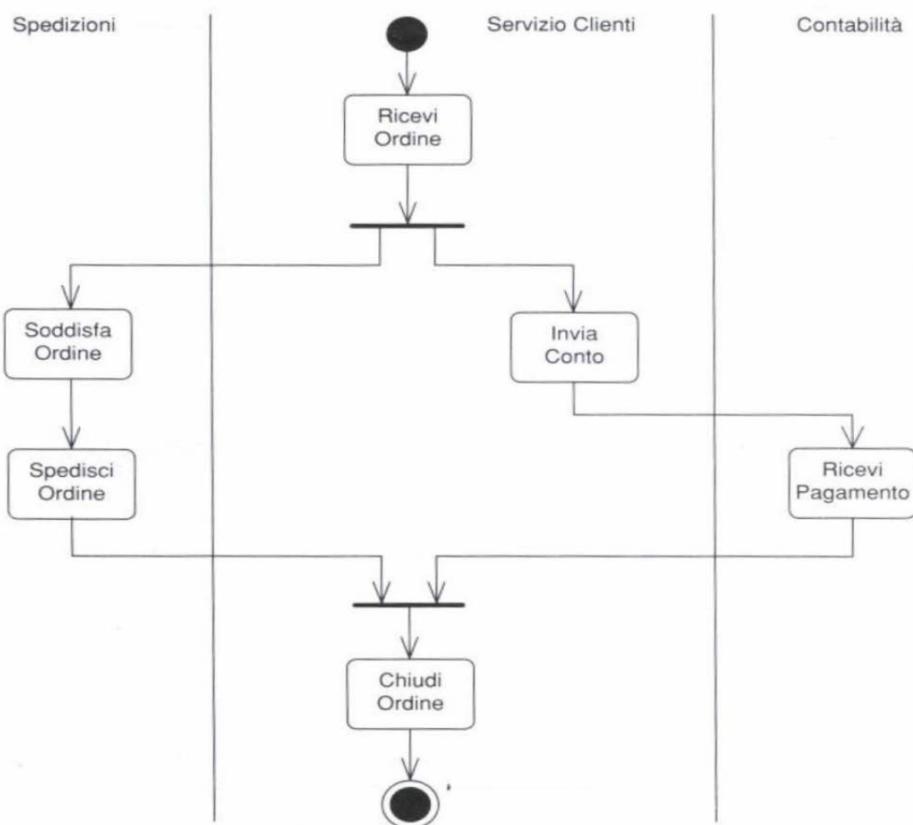


Diagramma Macchina a Stati

Una macchina a stati descrive il comportamento dinamico delle istanze di una classe, rappresentandone gli stati significativi e le transizioni tra di essi. Le transizioni avvengono in risposta a eventi, come messaggi esterni o eventi generati internamente.

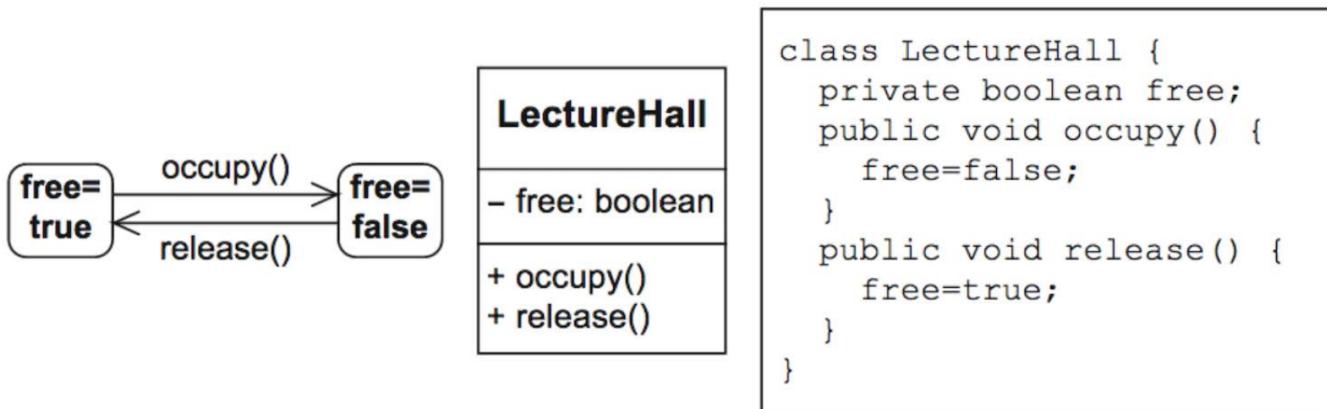
Lo **Stato** è un *insieme di valori* di un oggetto.

Stato: 

Stato iniziale: 

Stato finale: 

Una macchina a stati si rappresenta tramite un grafo dove gli stati sono identificati da nomi univoci e rappresentati da rettangoli arrotondati, e le transazioni sono gli archi che legano gli stati.



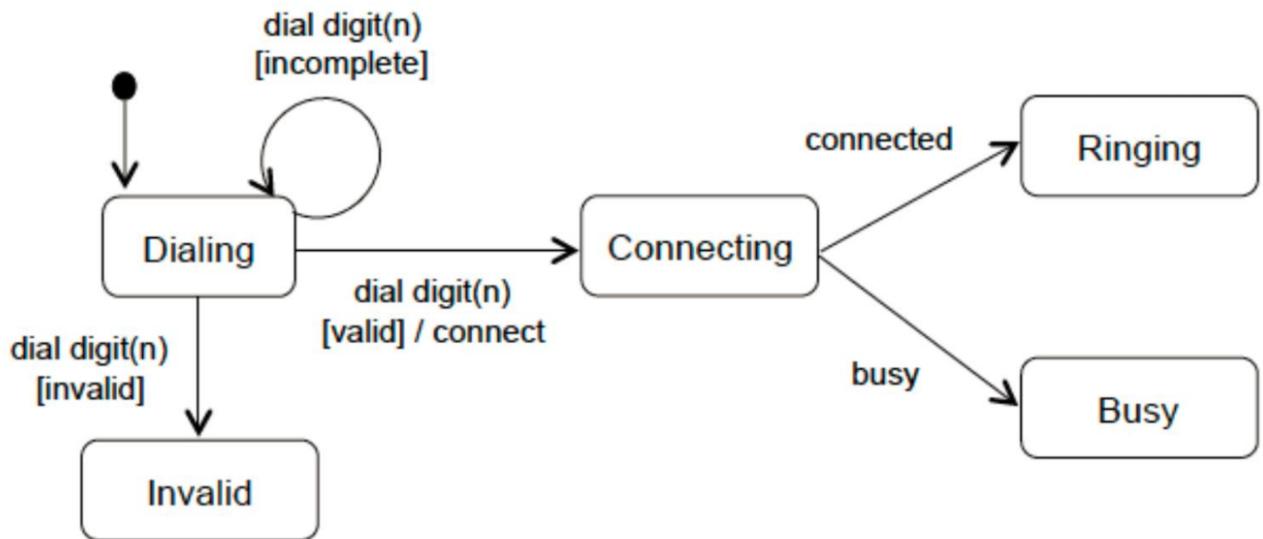
Evento

Un **Evento** occorre istantaneamente e viene considerato solo se l'oggetto è in uno stato per cui è prevista una transazione etichettata da quell'evento, sennò è ignorato.

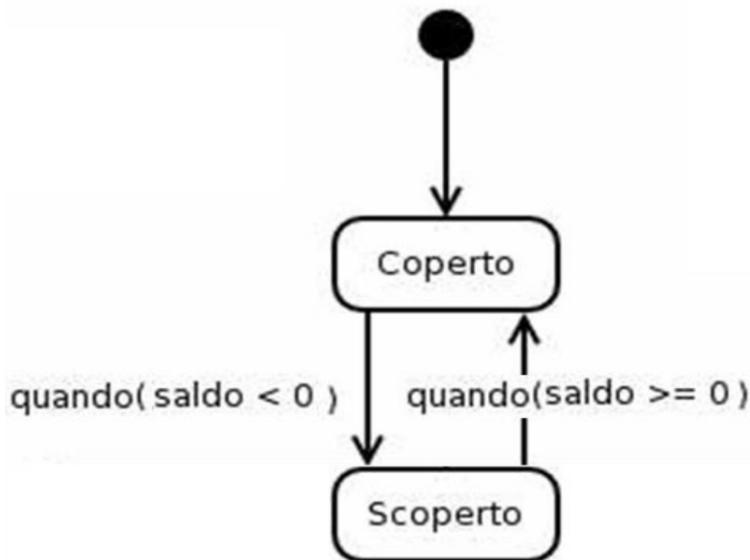
Possono essere di diversi tipi:

- **Operazionale o segnale - op(a:T)**: la transizione è abilitata quando l'oggetto riceve una chiamata di metodo, od un segnale, con parametri a e tipo T.

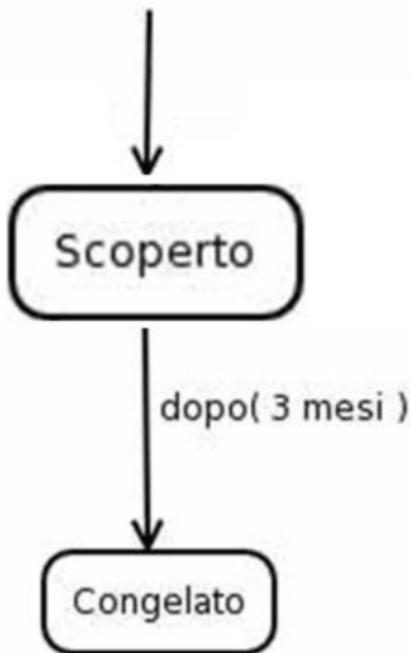
I parametri sono opzionali.



- **Evento di variazione - when(exp)**: la transizione è abilitata appena l'espressione diventa vera, la quale può indicare un tempo assoluto o una condizione su variabili.



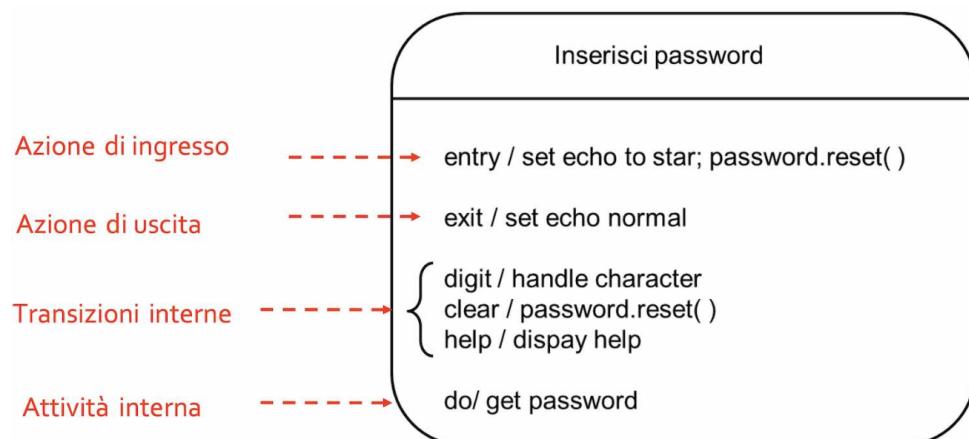
- **Evento temporale - after(time)**: la transazione è abilitata dopo che l'oggetto è stato fermo "time" in quello stato



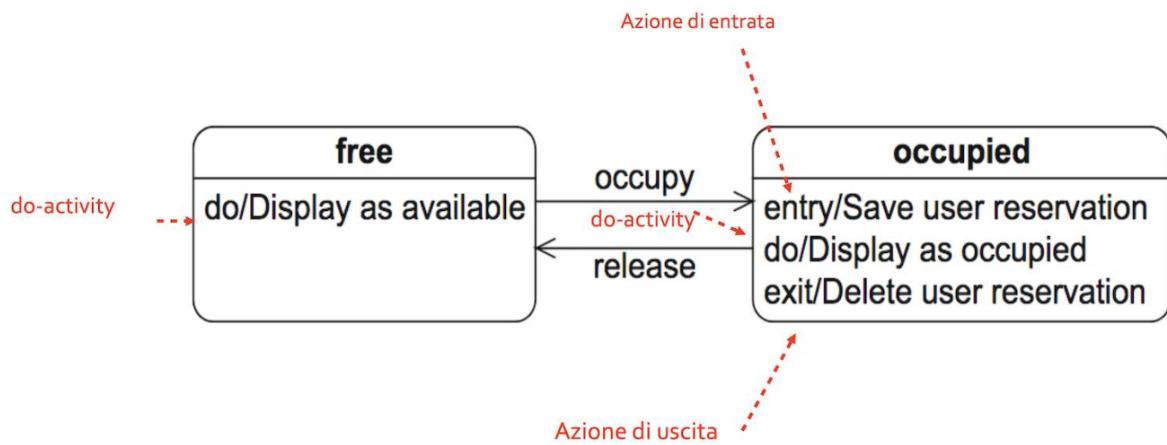
Stato Composito

Uno stato può contenere al suo interno dei *sottostati* legati da transizioni interne. In particolare, uno *stato composito* può contenere:

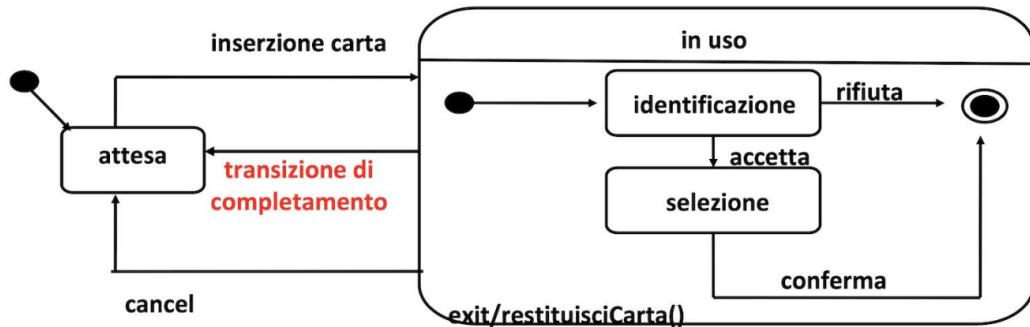
- *Azione di entrata (entry)*: eseguita all'ingresso in uno stato.
- *Azione di uscita (exit)*: eseguita all'uscita di uno stato.
- *Transizione interna*: eseguita in risposta ad un evento.
- *Attività interna (do)*: eseguita in modo continuato mentre l'oggetto si trova in quello stato, senza necessità di un evento scatenante. Le *attività interne*, al contrario delle altre azioni, *non sono atomiche* e possono quindi essere interrotte.



EX.



Avere azioni ed attività interne ad uno stato, permette di definire stati composti.



Gli stati composti possono essere:

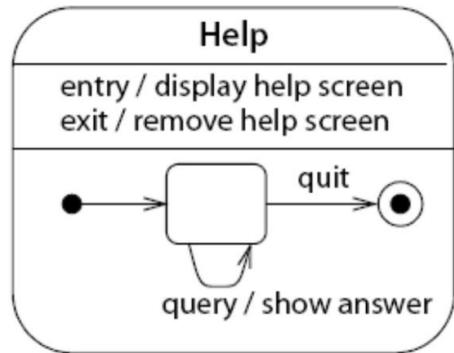
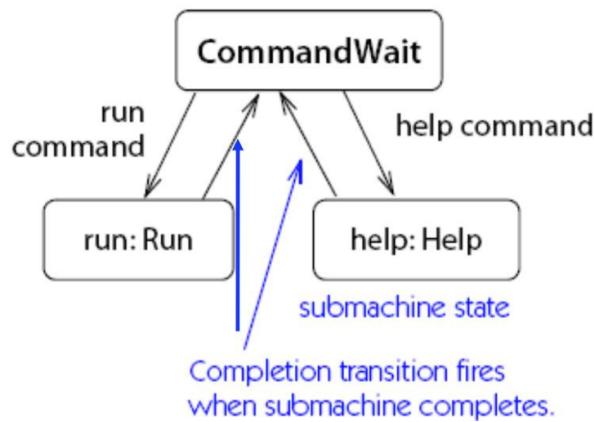
- *Sequenziali*: un solo sottostato attivo in ogni istante.
- *Paralleli*: più sottostati attivi contemporaneamente, uno per *regione*.

Le *Transizioni di completamento* sono prive di evento scatenante. Scattano nei seguenti casi:

- Al termine di un'attività composita (stato finale, exit point).
- Alla terminazione di *entry* e/o *do activity*.
- Al raggiungimento di uno pseudo-stato di transizione.

Sottomacchine

Si usano quando vogliamo descrivere un sottostato in un diagramma a parte.
Ogni sottomacchina ha un nome e un tipo (nomelstanza:Tipo).



This submachine can be used many times.

Si possono definire *Entry e Exit points* che servono a collegare le transizioni della macchina principale

Pseudo-stato

Nei diagrammi di macchina a stati vi sono *nodi* particolari, chiamati *pseudo-stati*:

Giunzione



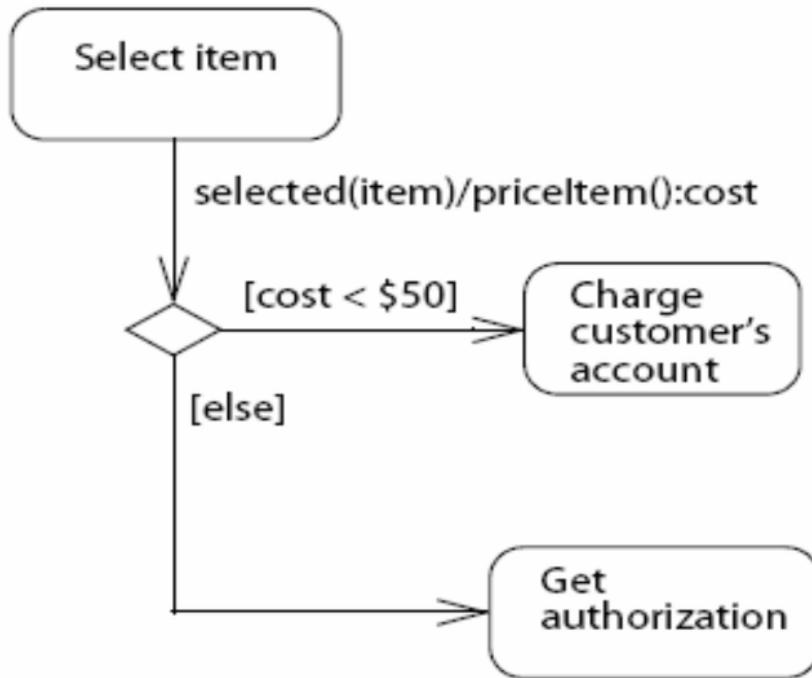
Storia



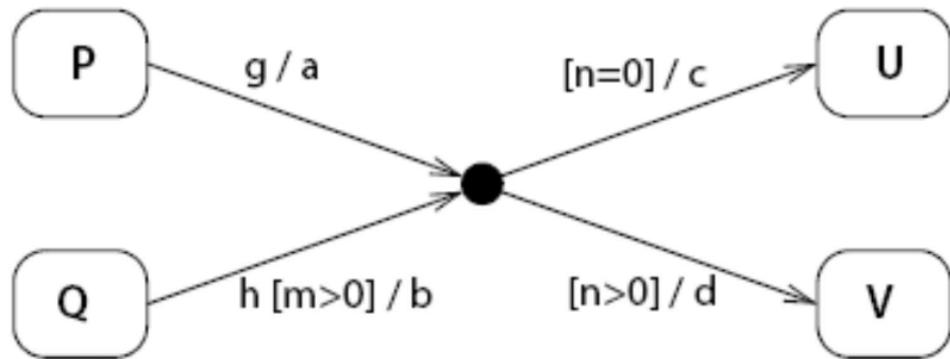
Decisione



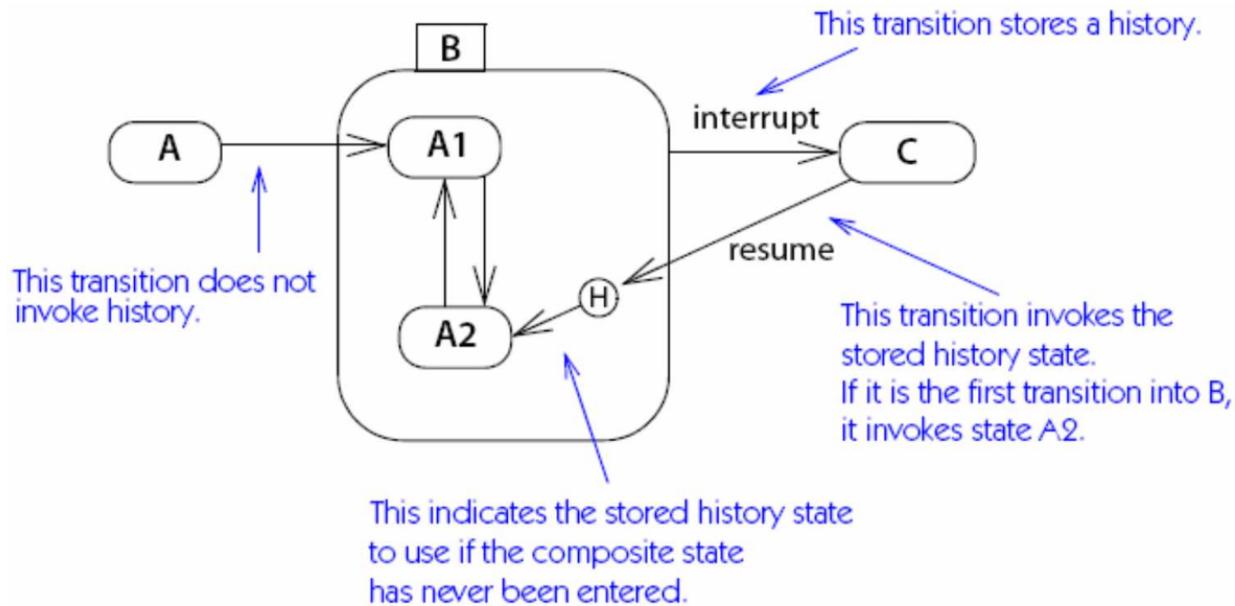
- **Decisione:** Permette di decidere quale transizione scatenare, a seconda di quale guardia viene soddisfatta. Essa valuta condizioni dinamicamente, e la disgiunzione delle guardie deve essere true, al fine di non bloccare il tutto. E anche ammesso il non-determinismo.



- **Giuⁿzione:** Entrano/Escono due o più transizioni. La disgiunzione delle guardie in uscita non deve necessariamente essere true.



- **Storia:** Permette di memorizzare lo stato corrente di una certa sotto-macchina. In questo modo, essa potrà riprendere ad operare dopo essere stata sospesa.



Confronto

Il *diagramma di macchina a stati* parla dell'evoluzione nel tempo delle istanze di un classificatore.

Se **mostrare l'evoluzione di un oggetto in risposta ad eventi**, allora è meglio il diagramma di macchina a stati.

Il *diagramma di attività* parla di un'agenda di azioni da fare.

Se **mettere in ordine un insieme di azioni da fare**, allora è meglio il diagramma delle attività;

Tipi di evento

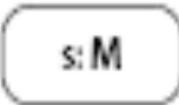
<i>Event Type</i>	<i>Description</i>	<i>Syntax</i>
call event	Receipt of an explicit synchronous call request by an object	<code>op (a:T)</code>
change event	A change in value of a Boolean expression	<code>when (exp)</code>
signal event	Receipt of an explicit, named, asynchronous communication among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

Tipi di transizione

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

Tipi di stato

<i>State Kind</i>	<i>Description</i>	<i>Notation</i>
simple state	A state with no substructure	
orthogonal state	A state that is divided into two or more regions. One direct substate from each region is concurrently active when the composite state is active.	
nonorthogonal state	A composite state that contains one or more direct substates, exactly one of which is active at one time when the composite state is active	
initial state	A pseudostate that indicates the starting state when the enclosing state is invoked	
final state	A special state whose activation indicates the enclosing state has completed activity	
terminate	A special state whose activation terminates execution of the object owning the state machine	

junction	A pseudostate that chains transition segments into a single run-to-completion transition	
choice	A pseudostate that performs a dynamic branch within a single run-to-completion transition	
history state	A pseudostate whose activation restores the previously active state within a composite state	
submachine state	A state that references a state machine definition, which conceptually replaces the submachine state	
entry point	A externally visible pseudostate within a state machine that identifies an internal state as a target	
exit point	A externally visible pseudostate within a state machine that identifies an internal state as a source	

Capitolo 9

Progettazione

La progettazione costituisce la fase di mezzo tra quelle di specifica e codifica, in quanto con essa si passa da **cosa** deve essere fatto a **come** deve essere fatto.

Il prodotto finale della fase di progettazione prende il nome di *architettura software*.

La progettazione software si svolge su due livelli principali:

- **Progettazione architettonica**: si occupa di definire l'architettura di alto livello del sistema, scomponendolo in sotto-sistemi e specificando le loro relazioni.
- **Progettazione di dettaglio**: stabilisce come realizzare le specifiche delle singole parti definite dalla progettazione architettonica.

Viste

Una vista di un'architettura software è un'astrazione che considera un aspetto specifico del sistema. Le principali viste simultanee sono:

- **Vista comportamentale**: descrive le unità del sistema, i loro comportamenti e le interazioni a tempo di esecuzione, valutando aspetti come **qualità**, **scalabilità** ed **efficienza**.
- **Vista strutturale**: mostra il sistema come un insieme di unità realizzative, utile per sviluppare **test** di unità e interrogazione.
- **Vista logistica**: analizza le relazioni del software con altre strutture nel contesto del sistema, valutando **prestazioni** e **affidabilità**.

Vista Comportamentale

La vista comportamentale (o **C&C**, *component-and-connector*) descrive un sistema software come composizione di componenti software.

Una **componente software** è un'unità concettuale che **rappresenta una parte del sistema a tempo di esecuzione** (ad esempio, processi, oggetti, database). Incapsula funzionalità e dati, fornendo accesso controllato tramite interfacce.

Un sistema software è costituito da più componenti collegate tra loro tramite connettori.

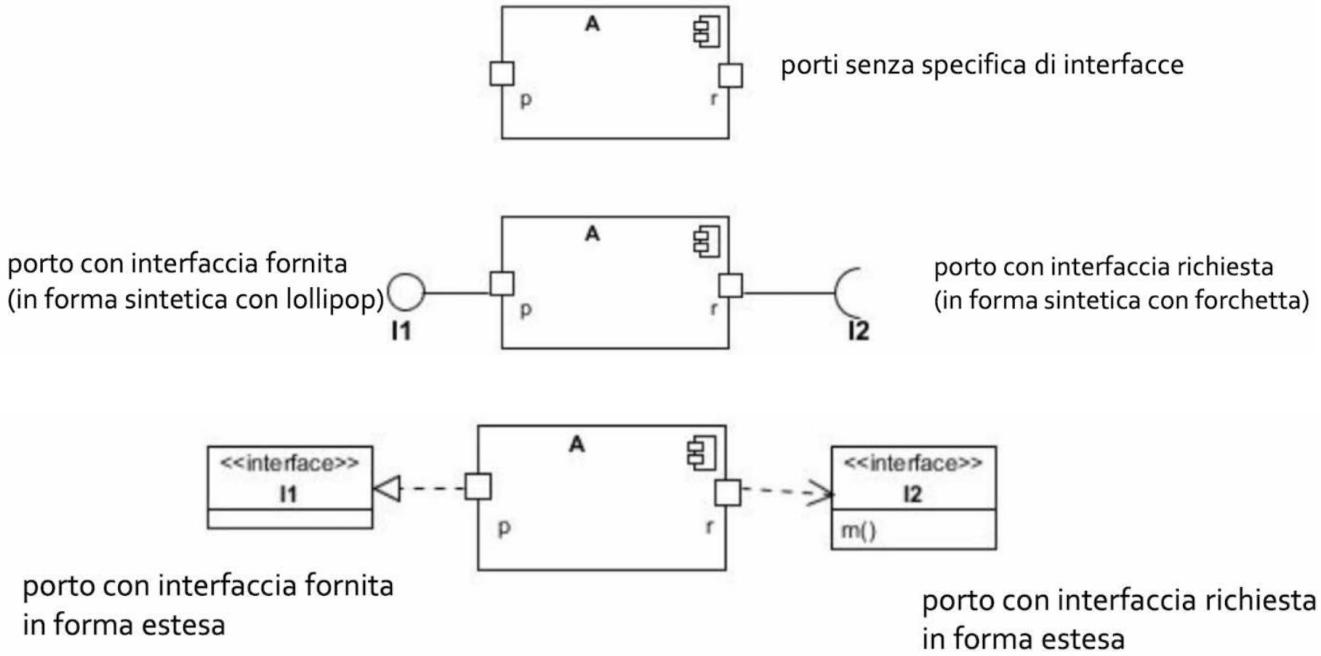
Le componenti software si possono rappresentare graficamente nei seguenti modi, tutti equivalenti:



Porti

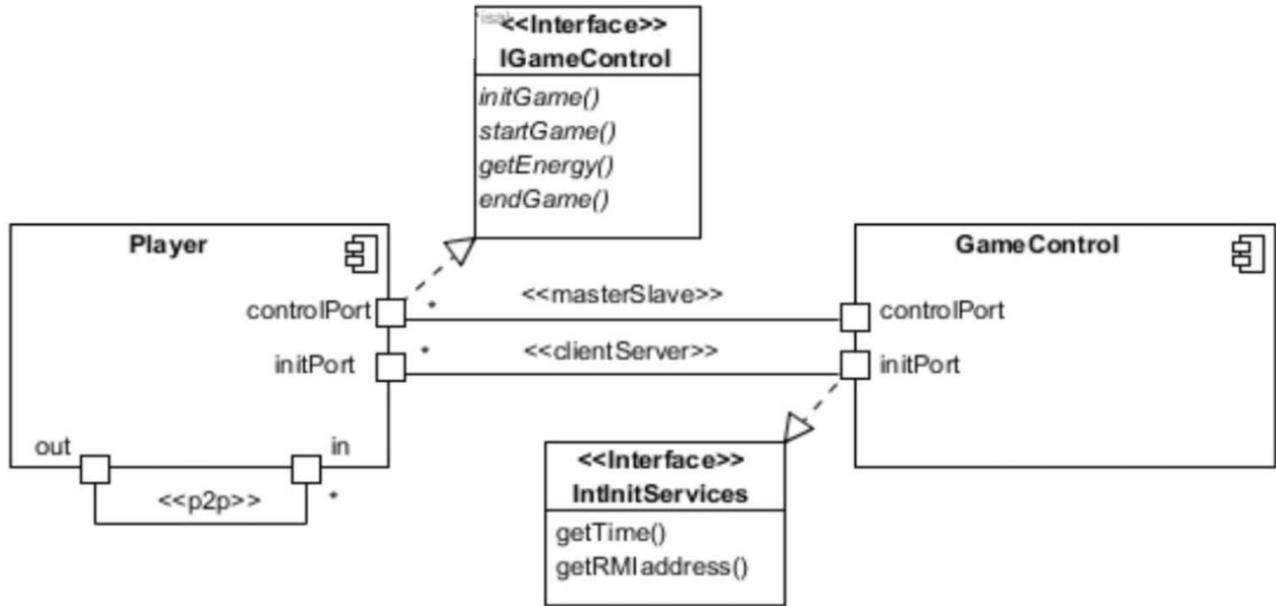
I **porti** (rappresentati mediante quadratini) identificano i **punti di interazione di un componente**, il quale può avere più porti, uno per ogni tipo di connessione con altri componenti.

Un porto **fornisce** o **richiede** una o più **interfacce** omogenee. Inoltre, ad esso è associata una molteplicità.



Connettori

I *connettori* sono canali di interazione tra componenti che collegano i porti.



Viste Strutturali

Le *viste strutturali* servono principalmente per:

- Fornire lo schema del codice e dell'albero di file e directory che verranno usati dal sistema;

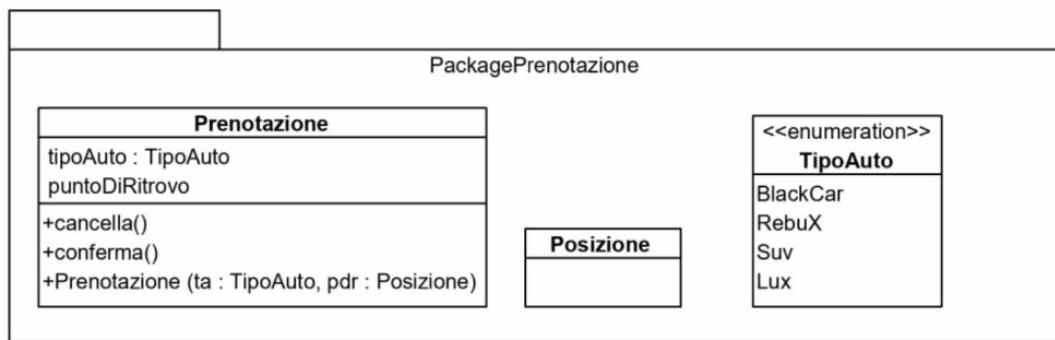
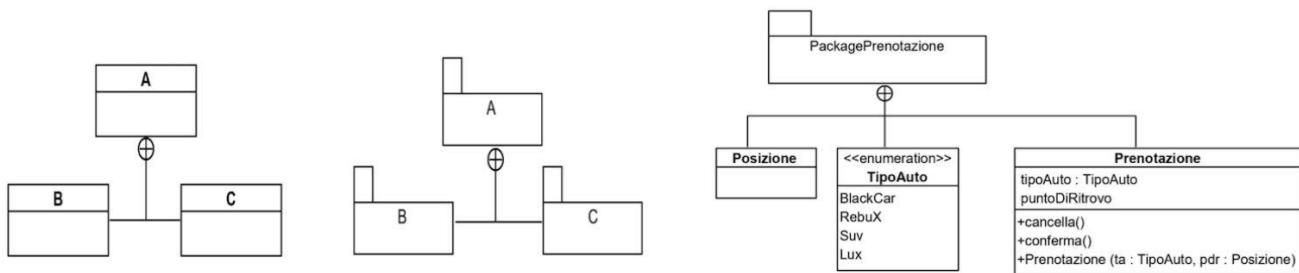
- Analizzare i requisiti e la loro possibile realizzazione;
- Progettare test di unità e di integrazione.

Vi sono 4 diversi tipi di viste strutturali, che differiscono per il tipo di relazioni utilizzate:

- *Vista strutturale di decomposizione*, usa la relazione **parte di** \oplus .

Una classe fa parte di (è contenuta in) un package, un package fa parte di uno più grande.

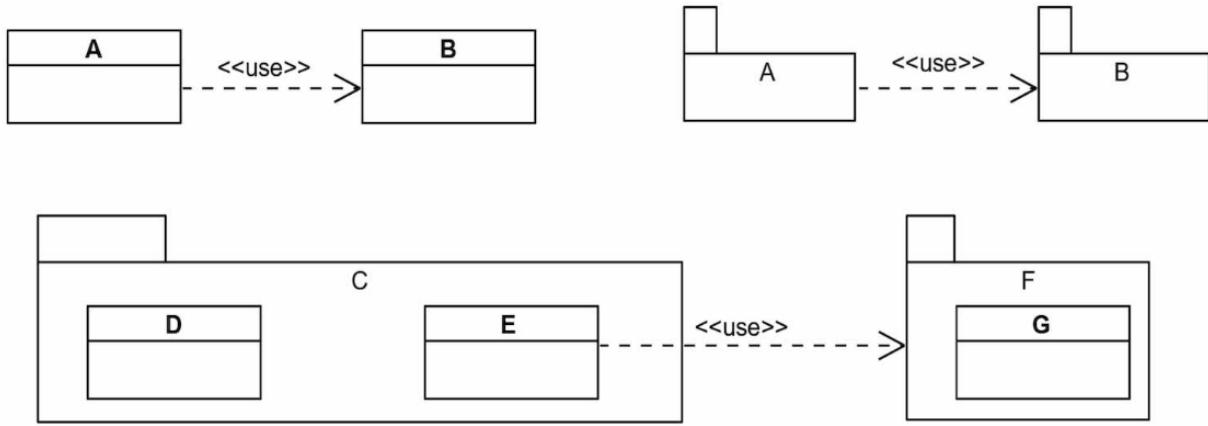
Serve per raggruppare classi e package.



- *Vista strutturale d'uso*, usa la relazione **usa**.

Il modulo A usa il modulo B se dipende dalla presenza di B per soddisfare i suoi requisiti.

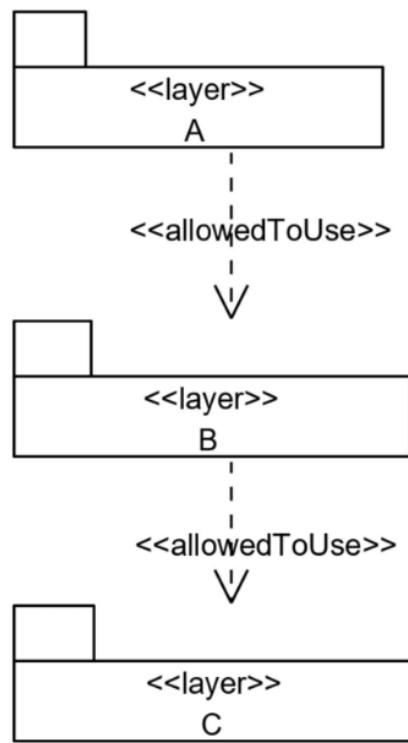
Viene usata per lo sviluppo di test e la pianificazione di sviluppo incrementale.



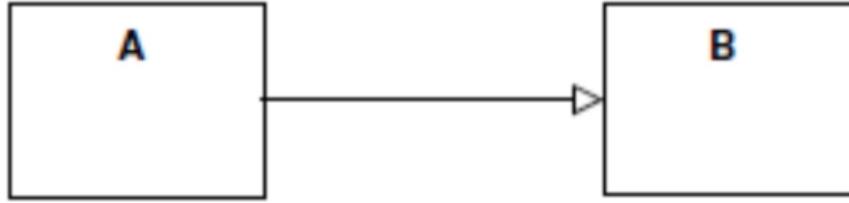
- *Vista strutturale a strati*, usa la relazione **può usare**.

Ogni elemento è uno strato, cioè un insieme coeso di moduli che offre un’interfaccia pubblica per i suoi servizi.

Utile per controllare la complessità del sistema, e quindi la sua scalabilità.



- *Vista strutturale di generalizzazione*, usa la relazione di generalizzazione.
Serve a rappresentare la relazione di sotto-tipo tra classi o packages.



Viste logistiche

Le *viste logistiche* si concentrano sull'analisi delle prestazioni di un sistema e sulla realizzazione di guide d'installazione.

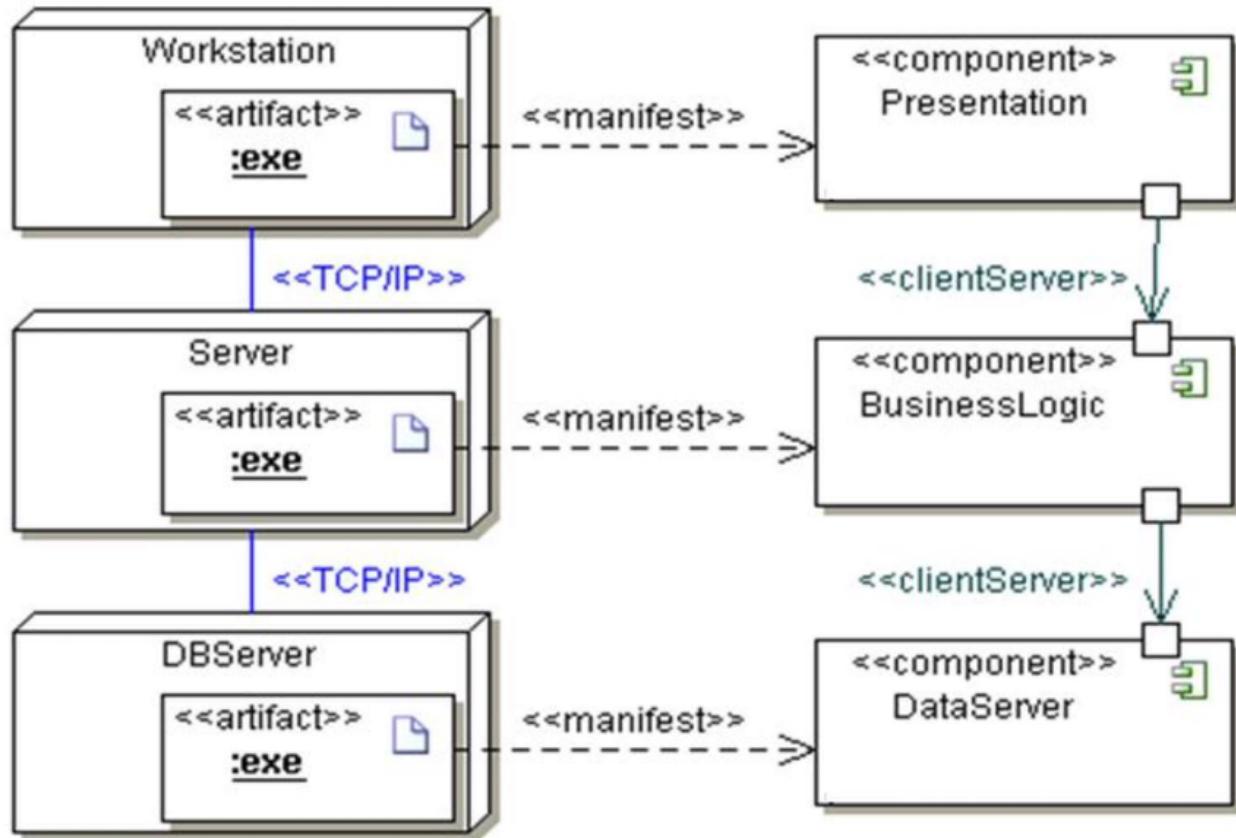
Queste viste utilizzano gli *artefatti*, che sono informazioni fisiche prodotte o utilizzate durante il processo di sviluppo software o nel funzionamento del sistema. Gli artefatti possono essere di vario tipo: codice sorgente, script, file eseguibili, tabelle di database, documenti o messaggi di posta elettronica.

Ogni artefatto viene analizzato all'interno di un ambiente d'esecuzione, che rappresenta l'entità hardware in cui il sistema opera, come un PC o un cellulare.

Un aspetto importante delle viste logistiche è che la relazione tra i componenti software e gli artefatti è unidirezionale: *prima si definiscono le componenti* (cioè, quali servizi devono essere offerti dal sistema), e *successivamente si creano gli artefatti che implementano queste componenti*.

In altre parole, gli *artefatti* rappresentano l'**implementazione concreta delle componenti software**, ed è l'artefatto a “manifestare” la componente.

Un'architettura software può includere diverse viste logistiche, mostrando come i vari artefatti vengono distribuiti tra diversi ambienti d'esecuzione



Stili architetturali

Gli *stili architetturali* rappresentano **schemi ricorrenti per organizzare** le componenti di **un sistema software** e le loro interazioni.

Gli stili architetturali rappresentano schemi ricorrenti per organizzare le componenti di un sistema software e le loro interazioni. I principali stili:

- *Condotte e Filtri*: definisce componenti come filtri che trasformano flussi di dati e connettori come condotte unidirezionali per il trasferimento dei dati (pipe).
- *Client-Server*: separa le componenti in client, che inviano richieste, e server, che forniscono servizi, spesso in contesti distribuiti.
- *Master-Slave* è una variante in cui un server (slave) serve un unico client (master).
- *P2P* (Peer-to-Peer): tutte le componenti sono pari grado, agendo sia come client che come server.
- *Publish-Subscribe*: introduce una comunicazione indiretta tramite un broker; i **publisher** inviano eventi, mentre i **subscriber** ricevono solo quelli a cui si sono

- abbonati, in modalità *push* (broker attivo che manda i messaggi ai sub) o *pull* (subscriber attivo che deve recuperare per conto suo i messaggi del broker).
- **MVC** (Model-View-Controller): separa dati (modello), presentazione (vista) e logica di controllo, migliorando modularità e manutenibilità.
 - **Coordinatore di Processi**: gestisce processi complessi attraverso un coordinatore che conosce la sequenza di attività e interagisce con server che forniscono servizi senza consapevolezza del processo globale.
-

Capitolo 10

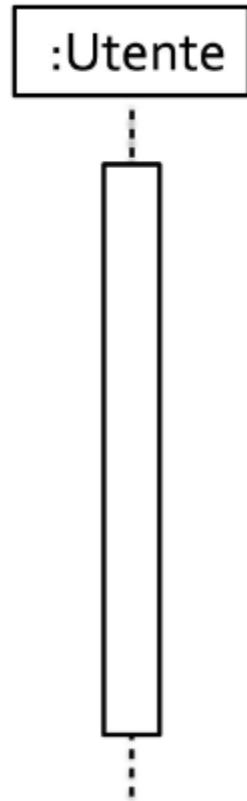
Diagrammi di sequenza

I *diagrammi di sequenza* si usano per *descrivere* le **interazioni tra oggetti**, intese come **scambi di messaggi o dati**.

Struttura

Le *interazioni* sono organizzate in una *sequenza temporale*, in cui gli *oggetti* partecipanti sono rappresentati *con linee di vita* formate da:

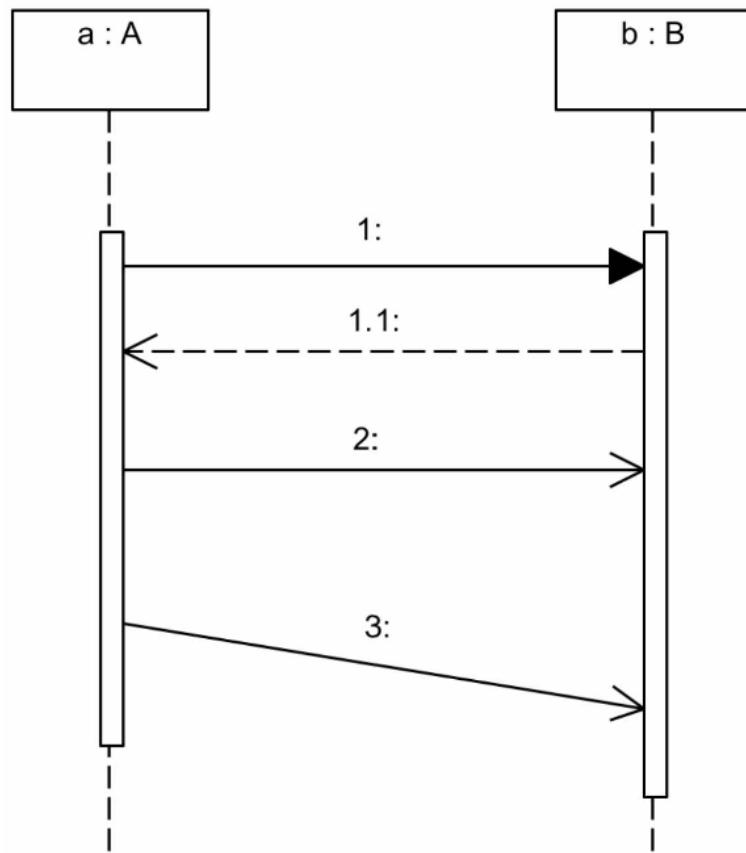
- Un rettangolo, che indica ruolo (nell'interazione) e/o tipo dell'oggetto (uno dei due obbligatorio, entrambi solo se utile);
- Una linea verticale, chiamata linea di vita dell'oggetto, tratteggiata quando l'oggetto è inattivo e continua e doppia quando attivo.



I *messaggi* scambiati tra oggetti *rappresentano* l'invocazione di **operazioni** o **segnali**

I messaggi possono essere:

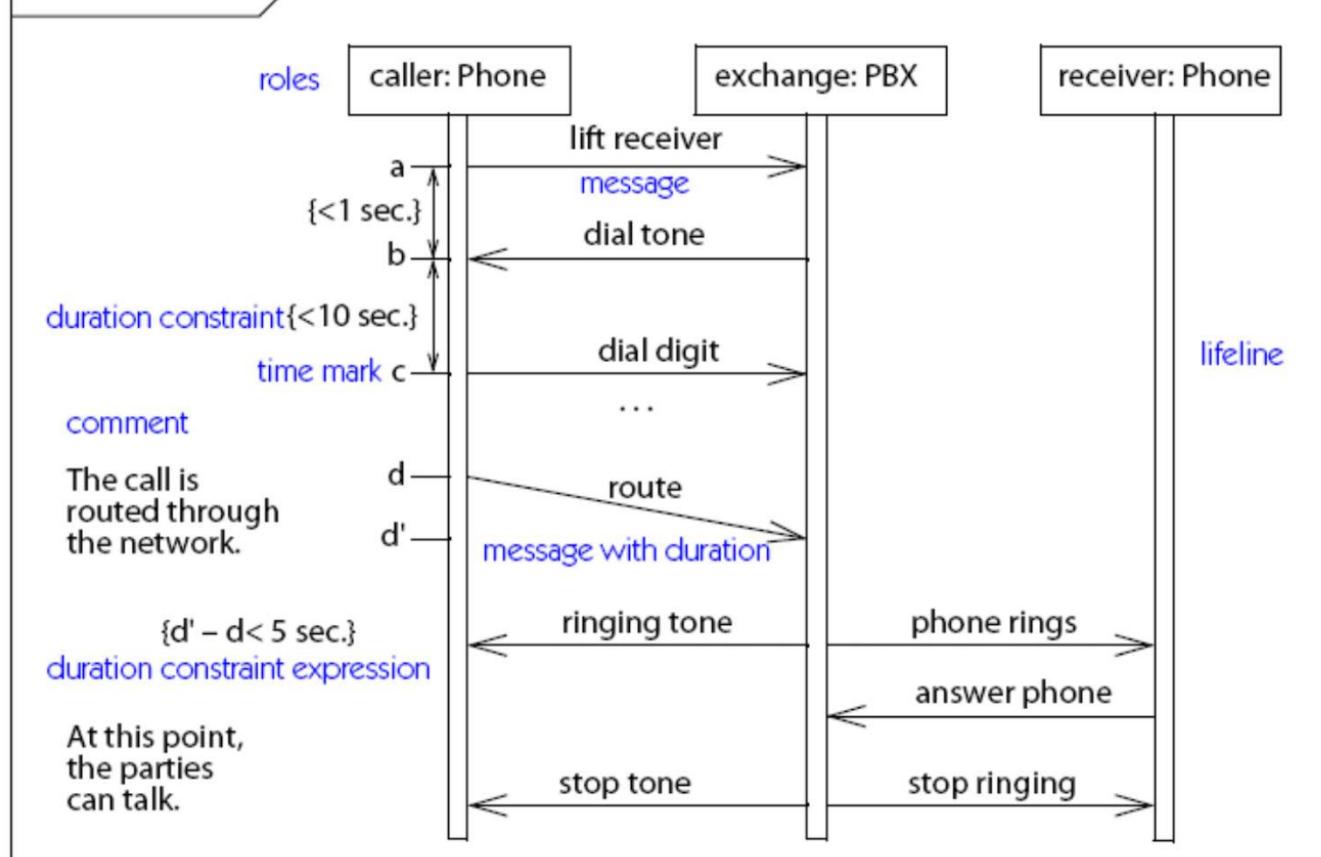
- *Sincroni* (1)
- *Return* (1.1), cioè di risposta (opzionali)
- *Asincroni* (2) come l'invio di una email
- *Asincroni con esplicito consumo di tempo* (3) come l'invio di una lettera



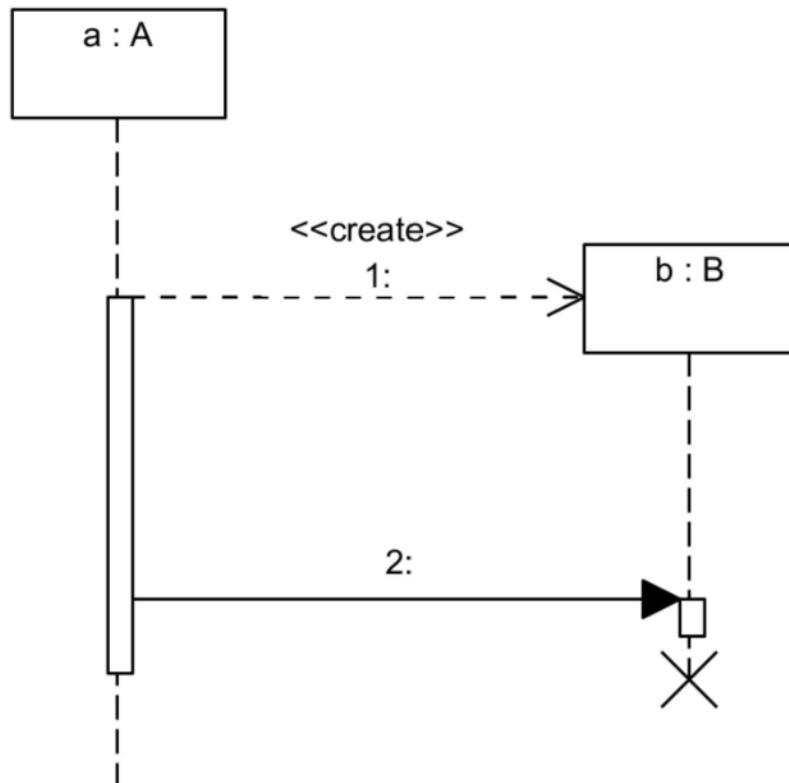
Sintassi: `attributo = nomeMessaggio(arg1, arg2, ...) : valoreDiRitorno`
 dove attributo, argomenti e valore di ritorno sono opzionali.

I *vincoli di durata* possono essere inseriti per specificare *tempi massimi o minimi di attesa tra diverse interazioni*.

sd conversation

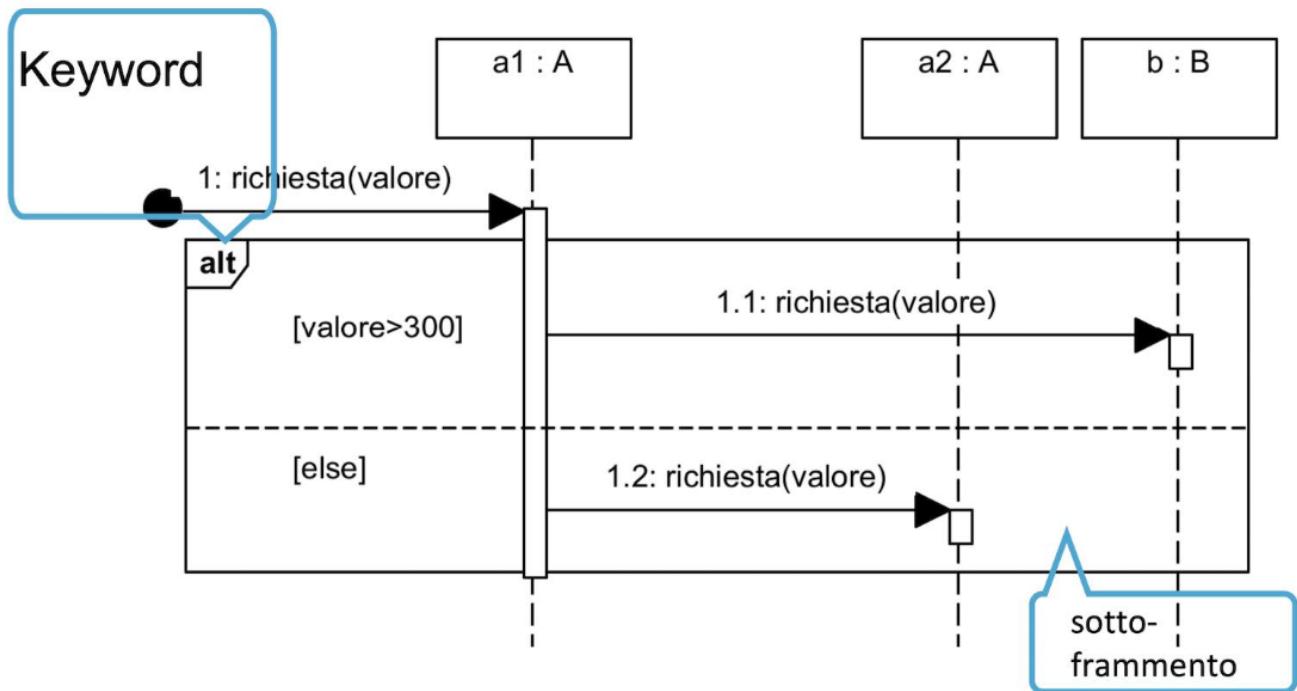


I *partecipanti* possono essere dinamicamente *aggiunti* o *cancellati* dall'interazione.

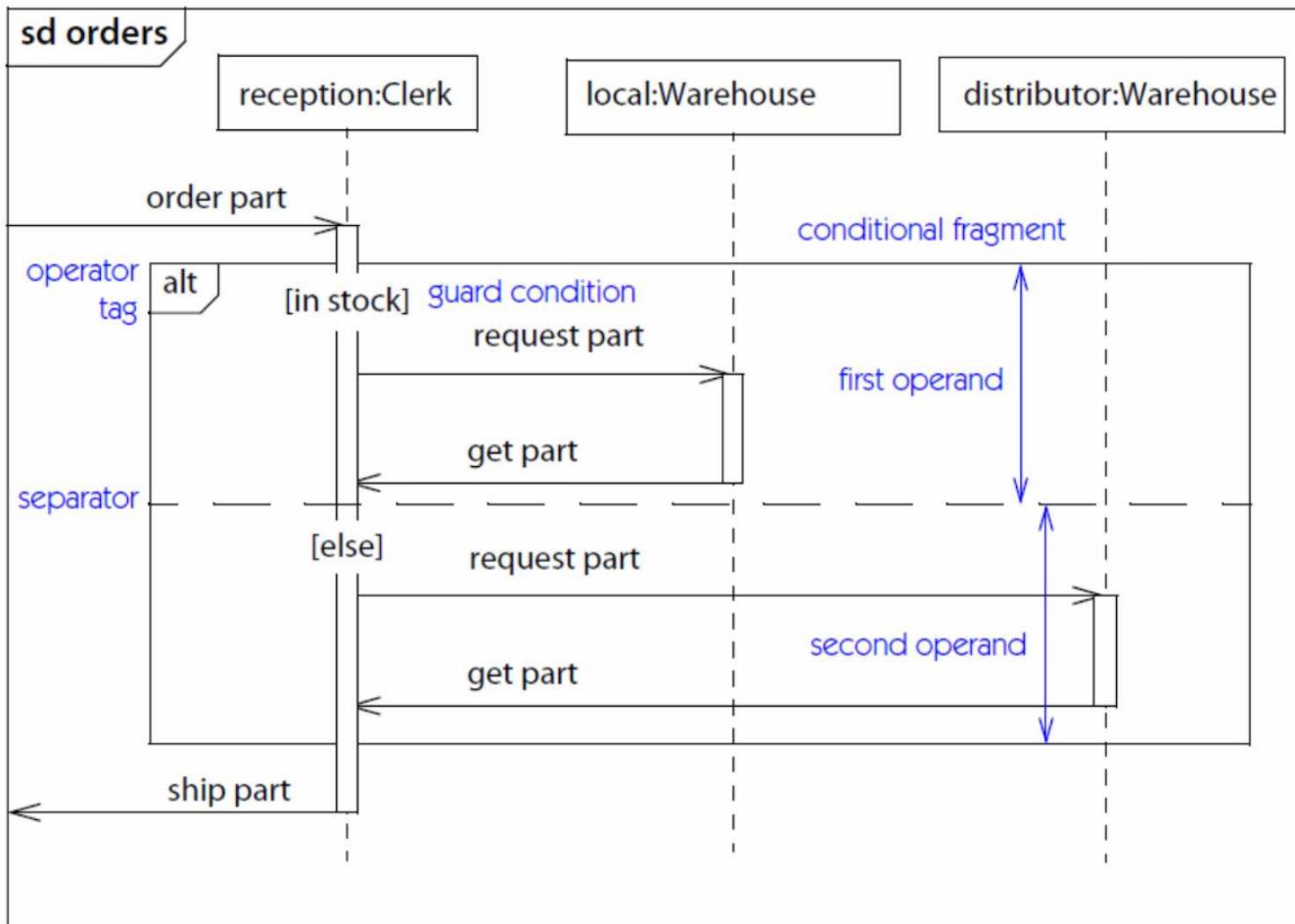


Frame

I *Frame Condizionali* rappresentati da rettangoli con la keyword `alt`, permettono di **implementare delle condizioni**. In particolare, il frame è diviso in sotto-frammenti, ognuno dei quali realizza un ramo della selezione.

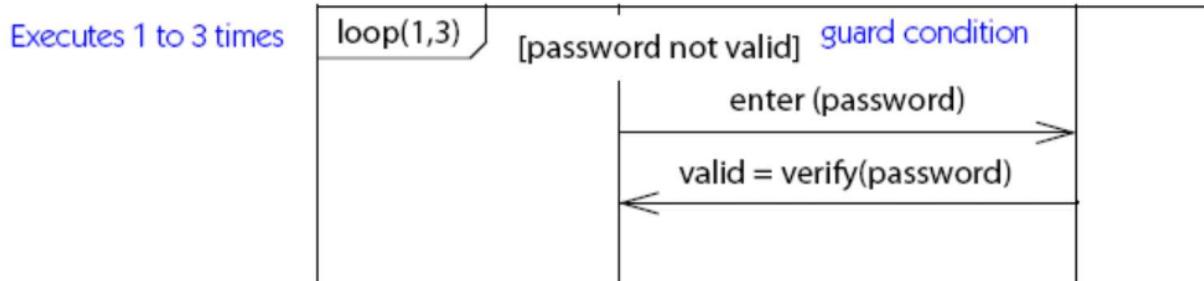


Se il frame è privo di guardia, allora è implicitamente sempre true. Invece, se ci sono più guardie vere al contempo, ne viene seguita una solo in modo non deterministico. Infine, se nessuna guardia è vera, il frame viene saltato.



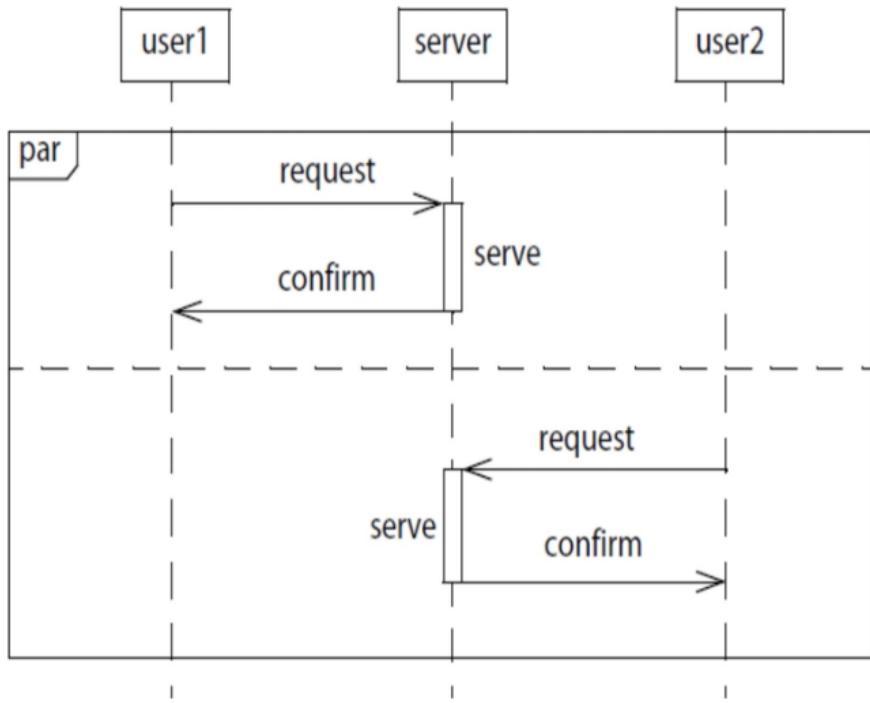
Il *frame opzionale* è nella pratica un *frame condizionale* con il **solo ramo dell'if**.
Quindi viene eseguito solamente se la condizione è vera.

I *frame iterativi* permettono di **ripetere** più volte le stesse azioni.



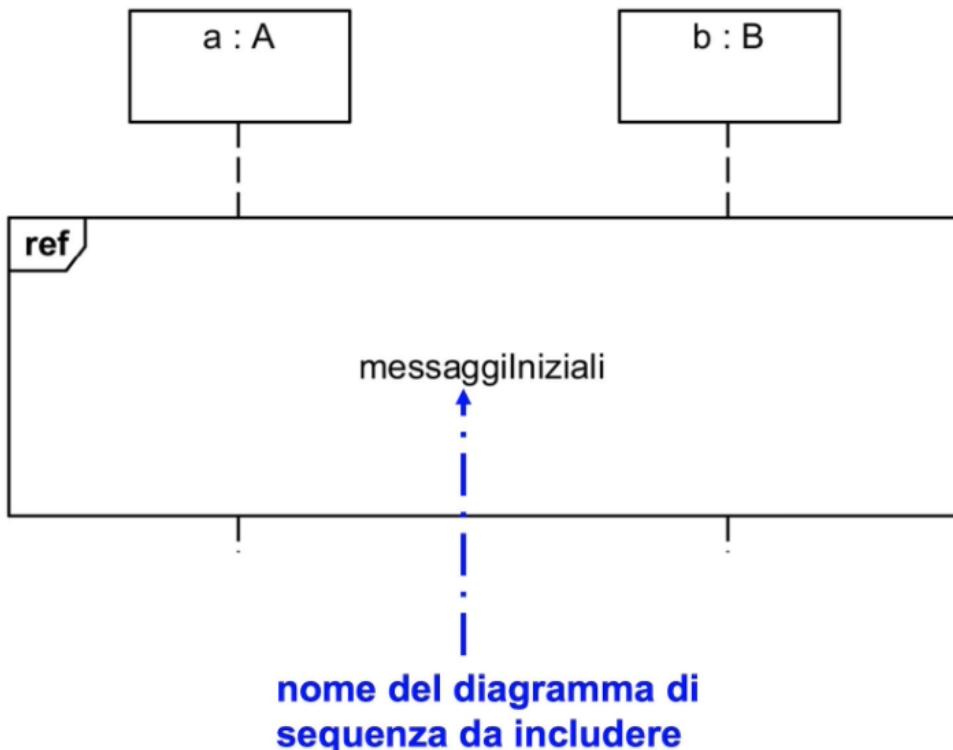
Nel *frame parallelo*, le **interazioni** nei diversi sotto-frammenti vengono **eseguite in parallelo**.

Le richieste dei due user al server possono arrivare in un ordine qualsiasi.



I *frame di inclusione ref* permettono di definire diagrammi di sequenza (identificati da un nome univoco) e **richiamarli in altri diagrammi**.

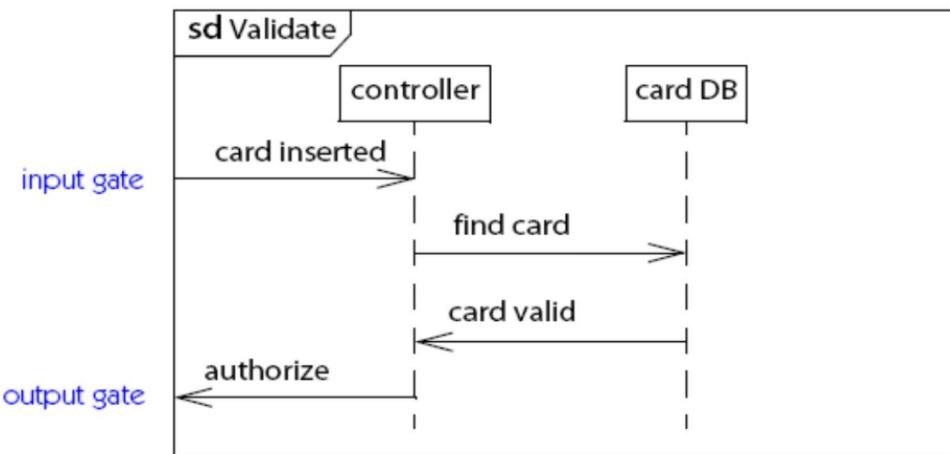
Ogni frame è una sorta di “blocco” che può essere richiamato più volte (come una funzione).



Gate

I *gate* sono punti sul bordo del diagramma (identificati da un nome) a cui è collegato un messaggio, in ingresso o in uscita.

Essi sono particolarmente utili quando un diagramma deve essere incluso in altro diagramma.



Capitolo 11

Principi di progettazione

La fase di progettazione non si limita a pianificare il lavoro per soddisfare i requisiti funzionali e di qualità, ma mira anche a garantire che il sistema sia:

- *Facilmente mantenibile*: riducendo i costi di manutenzione lungo il ciclo di vita.
- *Riusabile*: favorendo il riutilizzo di parti del sistema in futuri progetti.

La manutenzione comprende tutti i cambiamenti apportati al prodotto software, anche dopo la consegna al cliente. Si distingue in:

1. **Manutenzione correttiva** (~20%): Rimuove errori senza modificare la specifica.
2. **Manutenzione migliorativa**:
 - **Perfettiva** (~60%): Introduce nuove funzionalità o migliora quelle esistenti.
 - **Adattativa** (~20%): Modifica il software per adeguarlo a nuovi contesti (leggi, hardware, sistemi operativi).

Poiché la manutenzione è uno dei costi più elevati nel ciclo di vita del software, è fondamentale seguire principi e pattern di progettazione per ottenere un sistema ben gestibile e sostenibile.

Principi generali

Information Hiding

L'*information hiding* consiste nel nascondere i dettagli interni di un oggetto, rendendo visibili solo i metodi e gli attributi necessari tramite un'interfaccia pubblica. Questo approccio migliora la comprensibilità, la manutenibilità, la sicurezza dei dati e facilita il lavoro in team.

I *metodi getter e setter* servono per accedere e modificare gli attributi in modo controllato, nascondendo la rappresentazione interna e garantendo una gestione sicura delle modifiche.

Astrazione

Esistono due tipologie di *astrazione*, entrambe estremamente desiderabili:

- **Astrazione sul controllo:** ottenuta facendo ricorso a librerie che forniscono funzioni e procedure già scritte e debuggata.
- **Astrazione sui dati:** ottenuta facendo uso di strutture dati astratte, le quali forniscono un'interfaccia nota per gestire i dati, astraendo da come essi siano internamente memorizzati.

Coesione

La *coesione* misura quanto strettamente sono collegati gli elementi di un'unità progettuale (es. un sottosistema o una classe). Un sistema coeso garantisce che ogni sottosistema realizzi un solo concetto e che le funzionalità simili siano raggruppate insieme.

Questo è un esempio di sistema per niente *Coeso* perché *Activities* è veramente troppo generale e i suoi metodi sono tutti scollegati fra loro.

```
public class Activities
{
    public void PrintDocument(Document doc) {
        ....
    }

    public void SendEmail(string rcpnt, string sbj, string txt) {
        ....
    }

    public void CalculateDist(int x1, int y1, int x2, int y2) {
        ....
    }
}
```

Tipologie di coesione (dalla migliore alla peggiore):

1. **Coesione funzionale**: gli elementi collaborano per una specifica funzionalità. È la situazione ideale.
2. **Coesione comunicativa**: gli elementi operano sugli stessi dati di input o output. Non facilita il riuso.
3. **Coesione procedurale**: gli elementi realizzano i passi di una procedura. Sono debolmente connessi e poco riutilizzabili.
4. **Coesione temporale**: le azioni sono legate dal tempo di esecuzione. È meglio separare le azioni in unità distinte e coordinarle con eventi.
5. **Coesione logica**: gli elementi sono logicamente, ma non funzionalmente, correlati. È poco efficace per il riuso.
6. **Coesione accidentale**: gli elementi sono raggruppati senza alcuna relazione. È la peggiore forma di coesione.

Un buon progettista mira sempre alla *coesione funzionale* per ottenere unità più modulari e riutilizzabili.

Disaccoppiamento

Il *disaccoppiamento* si riferisce alla separazione tra unità progettuali (come sottosistemi o classi), in modo che siano il più indipendenti possibile. L'obiettivo è ridurre le dipendenze tra le unità per facilitare la manutenzione, il riuso e la comprensione del sistema.

Coesione vs Disaccoppiamento:

- **Coesione alta**: all'interno di ogni unità, gli elementi sono strettamente collegati per realizzare un'unica funzionalità.
- **Accoppiamento basso**: le unità interagiscono il meno possibile, mantenendo l'indipendenza.

In genere, un sistema con alta coesione ha anche un basso accoppiamento, perché le unità ben coese hanno meno bisogno di interagire tra loro.

L'ideale è bilanciare questi aspetti per creare sistemi modulari e gestibili.

SOLID

SOLID è un **insieme di principi di progettazione** molto diffusi in letteratura.

Essi sono:

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

Single Responsibility Principle

Il *Single Responsibility Principle* (SRP) stabilisce che una classe, un modulo o un metodo debba avere una sola responsabilità, ossia un unico motivo per essere modificata. Questo principio si basa sull'idea di evitare che un'unica classe gestisca più funzionalità non correlate, promuovendo così la coesione e la manutenibilità.

Motivi:

- **Isolamento delle modifiche**: Se una classe ha una sola responsabilità, le modifiche richieste per una funzionalità non influenzano le altre, riducendo il rischio di introdurre errori.

- **Facilità di comprensione e gestione:** Una classe con un'unica responsabilità è più semplice da leggere, testare e riutilizzare

EX.

Se abbiamo una classe `Rectangle` che calcola sia l'area del rettangolo, sia disegna il rettangolo su uno schermo, la classe non rispetta il *SRP*. Queste due funzionalità (calcolo e visualizzazione) sono indipendenti e dovrebbero essere separate:

- Una classe `Rectangle` per il calcolo geometrico.
- Una classe `RectangleRenderer` per gestire la visualizzazione.

Open Closed Principle

Il principio *Open/Closed* afferma che le entità software (classi, moduli, funzioni) devono essere:

- **Aperte per estensione:** È possibile aggiungere nuove funzionalità senza alterare il comportamento esistente.
- **Chiuse per modifiche:** Il codice esistente non dovrebbe essere modificato per integrare nuove funzionalità.

Come implementarlo:

1. Classi astratte e concrete:

- Si definiscono classi astratte che contengono il comportamento comune.
- Le classi concrete implementano o estendono queste funzionalità, senza modificare il codice della classe astratta.

2. Interfacce:

- Si utilizzano interfacce per definire un contratto generale.
- Nuove classi possono essere create implementando le interfacce esistenti, aggiungendo nuovi comportamenti senza cambiare il codice base.

3. Deleghe:

- Si spostano le responsabilità specifiche in classi dedicate.
- Un oggetto delega il lavoro ad altre classi senza modificare se stesso.

EX.

```
class ShapeCalculator {
    public double calculateArea(Rectangle rectangle) {
```

```
        return rectangle.getWidth() * rectangle.getHeight();
    }

    public double calculateArea(Circle circle) {
        return Math.PI * Math.pow(circle.getRadius(), 2);
    }
}
```

SBAGLIATO

```
—
class ShapeCalculator {
    public double calculateArea(Shape shape) {
        return shape.calculateArea();
    }
}
```

CORRETTO

Se vogliamo aggiungere una nuova forma dovremmo modificare la classe esistente violando il principio *Open/Closed*.

```
abstract class Shape {
    public abstract double calculateArea();
}

class Rectangle extends Shape {
    private double width, height;
    public double calculateArea() {
        return width * height;
    }
}

class Circle extends Shape {
    private double radius;
    public double calculateArea() {
        return Math.PI * Math.pow(radius, 2);
    }
}
```

Aggiungere nuove forme richiede solo la creazione di una nuova classe che estenda `Shape`, senza modificare il codice esistente.

Questo approccio rispetta il principio *Open/Closed*.

Liskov Substitution Principle

Le classi derivate devono potersi sostituire alle classi base.
Posso usare una classe figlio al posto della classe padre.

Interface Segregation Principle

Le interfacce devono essere specifiche e granulari, progettate per soddisfare i requisiti di un singolo cliente o gruppo di clienti. I clienti non devono dipendere da metodi che non utilizzano.

EX.

```
interface Animal {  
    void eat();  
    void fly();  
    void swim();  
}
```

Questo non è corretto perchè non tutti gli animali possono volare o nuotare

```
interface Eater {  
    void eat();  
}  
  
interface Flyer {  
    void fly();  
}  
  
interface Swimmer {  
    void swim();  
}
```

```
—  
  
class Dog implements Eater {  
    public void eat() {  
        System.out.println("Dog is eating");  
    }  
}
```

```

class Fish implements Eater, Swimmer {
    public void eat() {
        System.out.println("Fish is eating");
    }

    public void swim() {
        System.out.println("Fish is swimming");
    }
}

class Bird implements Eater, Flyer {
    public void eat() {
        System.out.println("Bird is eating");
    }

    public void fly() {
        System.out.println("Bird is flying");
    }
}

```

Così miglioriamo decisamente il codice.

Dependency Inversion Principle

I moduli di alto livello non devono dipendere dai moduli di basso livello, ma entrambi devono dipendere da astrazioni.

Inoltre, le astrazioni non devono dipendere dai dettagli concreti, ma i dettagli concreti devono dipendere dalle astrazioni.

EX.

```

class KeyboardReader {
    public String read() {
        return "Input dalla tastiera";
    }
}

class PrinterWriter {
    public void write(String data) {
        System.out.println("Stampa: " + data);
    }
}

```

```
class Copier {
    private KeyboardReader reader;
    private PrinterWriter writer;

    public Copier(KeyboardReader reader, PrinterWriter writer) {
        this.reader = reader;
        this.writer = writer;
    }

    public void copy() {
        String data = reader.read();
        writer.write(data);
    }
}
```

BRUTTO PERCHÈ COPIER DIPENDE DIRETTAMENTE DALLE CLASSI SOPRA E PER LEGGERE O SCRIVERE DA FONTI DIVERSE DOVREMMO MODIFICARE LA CLASSE

```
-
```

```
interface Reader {
    String read();
}
```

```
interface Writer {
    void write(String data);
}
```

```
class KeyboardReader implements Reader {
    public String read() {
        return "Input dalla tastiera";
    }
}
```

```
class PrinterWriter implements Writer {
    public void write(String data) {
        System.out.println("Stampa: " + data);
    }
}
```

```
class Copier {
    private Reader reader;
    private Writer writer;
```

```

public Copier(Reader reader, Writer writer) {
    this.reader = reader;
    this.writer = writer;
}

public void copy() {
    String data = reader.read();
    writer.write(data);
}
}

public class Main {
    public static void main(String[] args) {
        Reader reader = new KeyboardReader();
        Writer writer = new PrinterWriter();

        Copier copier = new Copier(reader, writer);
        copier.copy();
    }
}

```

MOLTO MEGLIO PERCHÈ ADESSO READER E WRITER POSSONO ESSERE QUALSIASI CLASSE CHE ESTENDE LE INTERFACCE E COPIER ESEGUE CORRETTAMENTE

GRASP

GRASP (General Responsibility Assignment Software Patterns) è un insieme di principi di progettazione che aiuta a definire le **responsabilità delle classi** e le **interazioni tra gli oggetti** nel contesto di un sistema software.

Dopo aver definito i requisiti e il modello di dominio, GRASP viene utilizzato per assegnare compiti specifici alle classi.

Le responsabilità si suddividono in due tipi principali:

1. **Fare**: include compiti come **eseguire azioni** (creare oggetti, eseguire calcoli), controllare altre azioni o modificare dati.
2. **Conoscere**: riguarda la **gestione di dati** privati, la conoscenza di oggetti correlati e il **calcolo o derivazione di informazioni**.

GRASP include una serie di pattern di progettazione che aiutano a definire come assegnare responsabilità alle classi e agli oggetti.

I principali pattern GRASP sono i seguenti:

1. **Information Expert**: Assegna la responsabilità a una *classe che possiede le informazioni essenziali per eseguire un'azione*, migliorando l'incapsulamento e riducendo l'accoppiamento.
2. **Creator**: Assegna la responsabilità di *creare istanze di una classe a un'altra classe* che contiene o aggrega informazioni relative alla classe da creare. È un'alternativa al pattern Factory.
3. **Controller**: Assegna la responsabilità di *gestire gli eventi di sistema* a una classe (Controller), che media tra l'interfaccia utente e le classi che implementano la logica del sistema.
4. **Low Coupling**: *Minimizza l'accoppiamento tra le classi*, migliorando la modularità e il riuso.
5. **High Cohesion**: Mantiene le *classi con responsabilità fortemente legate* tra loro, favorendo la coesione funzionale e la manutenibilità.
6. **Polymorphism**: Assegna la responsabilità di *gestire le variazioni comportamentali a classi o tipi che implementano tali variazioni*, sfruttando il polimorfismo.
7. **Pure Fabrication**: Crea una *classe* che *non rappresenta un concetto* del dominio ma è *progettata per* migliorare l'architettura, come ad esempio *ridurre l'accoppiamento o aumentare il riuso*.
8. **Indirection**: Introduce un *oggetto intermedio per mediare la comunicazione tra due componenti*, riducendo l'accoppiamento diretto tra di esse. Un esempio è l'uso del controller nel pattern Model-View-Controller (MVC).
9. **Protected Variations**: *Protegge gli elementi dalle modifiche* di altri sistemi o componenti mascherando le variazioni tramite interfacce e utilizzando il polimorfismo per gestire le diverse implementazioni.

Qualità del software

Il modello di qualità ISO/IEC 25010 è composto da 8 caratteristiche a loro volta suddivise in 31 sotto-caratteristiche.



Adeguatezza funzionale

Misura in cui un sistema soddisfa le esigenze dichiarate e implicite.

Le sotto-caratteristiche sono:

- *Completezza funzionale*: verifica se **tutte le funzioni richieste sono coperte**.
- *Correttezza funzionale*: misura la **precisione dei risultati ottenuti**.
- *Appropriatezza funzionale*: valuta quanto le **funzioni facilitate siano adatte** per raggiungere gli obiettivi richiesti.

Efficienza delle prestazioni

Si riferisce alle *prestazioni del sistema* in relazione alle risorse utilizzate.

Le sotto-caratteristiche includono:

- *Capacità*: misura i **limiti massimi** soddisfatti dal sistema.

- *Uso delle risorse*: verifica se le **risorse** sono **utilizzate** in modo **efficiente**.
- *Comportamento temporale*: misura la **velocità** e i tempi di **risposta**.

Compatibilità

Misura quanto un sistema può interagire con altri sistemi e componenti o funzionare in ambienti hardware o software comuni.

Le sotto-caratteristiche comprendono :

- *Interoperabilità*: verifica se i sistemi possono **scambiarsi e usare informazioni reciprocamente**.
- *Coesistenza*: misura la capacità di un sistema di **operare efficientemente in un ambiente condiviso** con altri prodotti.

Usabilità

Misura quanto un sistema è efficace, efficiente e soddisfacente per gli utenti.

Le sotto-caratteristiche includono:

- *Operabilità*: verifica la **facilità** d'uso del sistema.
- *Protezione dagli errori dell'utente*: valuta se il sistema **previene gli errori**.
- *Riconoscibilità dell'appropriatezza*: misura se gli **utenti** possono **riconoscere se il sistema soddisfa i loro bisogni**.
- *Apprendibilità*: verifica se il sistema è **facile da imparare**.
- *Estetica dell'interfaccia utente*: misura l'**interazione piacevole**.
- *Accessibilità*: valuta se il sistema può essere **utilizzato da un ampio spettro di utenti**.

Affidabilità

Misura quanto un sistema, prodotto o componente *riesca a eseguire funzioni in determinate condizioni* per un certo periodo di tempo.

Le sotto-caratteristiche sono:

- *Disponibilità*: verifica l'**operatività** del sistema quando richiesto.
- *Recuperabilità*: misura la **capacità di ripristinare** il sistema dopo un guasto.
- *Maturità*: valuta la **capacità di soddisfare i requisiti di affidabilità** durante il funzionamento normale.

- **Tolleranza ai guasti:** misura la capacità del sistema di **funzionare nonostante guasti** hardware o software.

Sicurezza:

Riguarda la protezione delle informazioni e dei dati in modo che l'accesso sia consentito solo a chi autorizzato.

Le sotto-caratteristiche includono:

- **Integrità:** previene l'accesso o la modifica **non autorizzata** ai dati.
- **Riservatezza:** garantisce che i **dati** siano **accessibili solo a chi ha il diritto**.
- **Non Ripudio:** assicura che le **azioni non possano essere negate**.
- **Responsabilità:** verifica se le **azioni possano essere tracciate**.
- **Autenticità:** assicura che l'**identità degli utenti o risorse sia verificabile**.

Manutenibilità

Misura quanto un sistema possa essere **modificato** per migliorarlo, correggerlo o adattarlo ai cambiamenti.

Le sotto-caratteristiche comprendono:

- **Modularità:** misura la **separazione dei componenti** del sistema
- **Riusabilità:** valuta se un **asset** possa essere **utilizzato in più contesti**,
- **Modificabilità:** verifica la **facilità di modifica** del sistema senza introdurre difetti,
- **Testabilità:** misura la **capacità di eseguire test** efficaci,
- **Analizzabilità:** valuta quanto è **facile comprendere l'impatto di una modifica**.

Portabilità

Riguarda quanto un sistema possa essere **trasferito** in ambienti diversi.

Le sotto-caratteristiche includono:

- **Installabilità:** misura la **facilità** con cui un sistema può essere **installato**,
- **Sostituibilità:** valuta se un sistema possa essere **sostituito da un altro** con le stesse funzionalità,
- **Adattabilità:** verifica la **capacità** di un sistema **di adattarsi a nuovi ambienti operativi**.

Scalabilità

Misura la capacità di un'applicazione di aumentare le sue prestazioni con l'aumento delle risorse hardware.

Esistono due tipologie di scalabilità:

- *Scalabilità verticale (scale up)*: riguarda l'**aggiunta di risorse a un singolo nodo**
 - *Scalabilità orizzontale (scale out)*: implica l'**aggiunta di più nodi hardware** per migliorare la capacità di gestione.
-

Capitolo 13

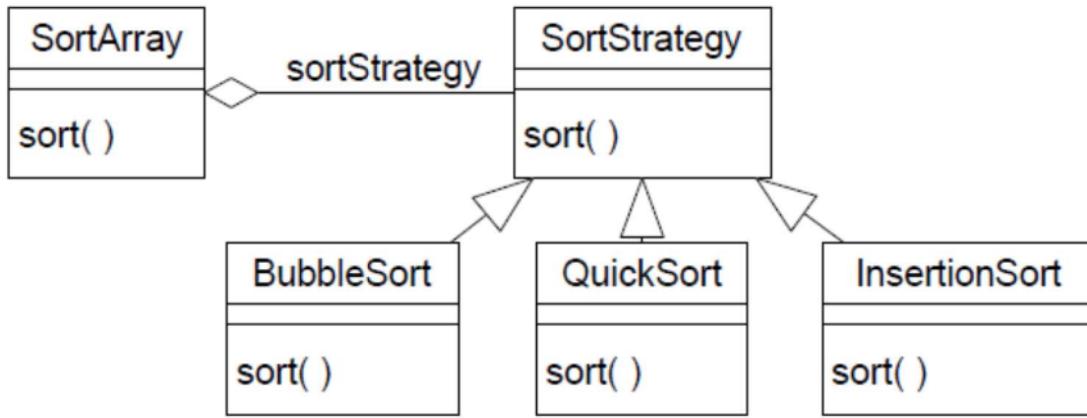
Strategy

Il *Design Pattern Strategy* è utilizzato quando abbiamo una **serie di classi che implementano lo stesso metodo**, ma con **implementazioni diverse**.

Invece di implementare questi metodi direttamente nelle classi, si crea una separazione, delegando l'implementazione a classi separate che possono essere cambiate dinamicamente.

Spiegazione del pattern Strategy:

1. *Interfaccia Strategy*: Si crea un'**interfaccia che definisce la segnatura del metodo** che si vuole eseguire. Questo metodo **non ha un'implementazione diretta**, ma è solo una “promessa” che le classi concrete dovranno implementare.
2. *Classi Concrete*: Si creano n classi (chiamate **ConcreteStrategy**) che implementano l'interfaccia Strategy, fornendo un'implementazione specifica del metodo definito nell'interfaccia. Ogni classe concrete avrà un comportamento diverso per lo stesso metodo.
3. *Classe Context*: ognuna delle nostre classi (**Context**) utilizzerà per delega la versione del metodo che fa al caso suo, istanziando la classe che lo realizza.



```

public class MyArray {
    private int[] array;
    private int size;
    private ArrayDisplayFormat format;

    public MyArray(int size) {
        array = new int[size];
    }

    public void setValue(...) {...}
    public int getValue(...) {...}
    public int getLength(...) {...}

    public void setDisplayFormat(ArrayDisplayFormat adf) {
        format = adf;
    }

    public void display() {
        format.printData(array);
    }
}

public interface ArrayDisplayFormat {
    void printData(int[] arr);
}

public class StandardFormat implements ArrayDisplayFormat {
    public void printData(...) {...}
}

```

```

public class MathFormat implements ArrayDisplayFormat {
    public void printData(...) {...}
}

public class StrategyExample {
    public static void main(String[] args) {
        MyArray m = new MyArray(10);
        m.setValue(...);
        m.setValue(...);
        System.out.println("Array in standard format:");
        m.setDisplayFormat(new StandardFormat());
        m.display();

        System.out.println("Array in math format:");
        m.setDisplayFormat(new MathFormat());
        m.display();
    }
}

```

Questo codice mostra l'implementazione del *Pattern Strategy* per visualizzare un array in diversi formati (standard e matematico).

La classe `MyArray` delega la visualizzazione dell'array alla strategia selezionata tramite il metodo `setDisplayFormat()`. Le classi `StandardFormat` e `MathFormat` implementano l'interfaccia `ArrayDisplayFormat` per definire come stampare l'array in ciascun formato.

Capitolo 14

Factories

Una *Factory* è una classe il cui solo scopo è quello di creare istanze di altre classi. Essa realizza quindi un **pattern creazionale**.

Vantaggi delle Factory:

1. Nascondono i dettagli di come un oggetto viene creato, semplificando l'utilizzo e migliorando la modularità.
2. Facilitano la costruzione di oggetti complessi, delegando alla factory l'inizializzazione degli oggetti in modo coerente.

3. Favoriscono il principio *code to an interface*, ovvero programmare su un'interfaccia anziché su un'implementazione concreta. Usare direttamente il costruttore di una classe, ad esempio, implica conoscere l'implementazione, cosa che una factory evita.

Tipi di pattern per le Factory:

1. **Simple Factory (Concrete Factory):** Una classe con un metodo statico che crea e restituisce oggetti di un tipo specifico.
2. **Abstract Factory:** Un pattern che fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.
3. **Factory Method:** Un pattern che delega alle sottoclassi la responsabilità di decidere quale classe concreta istanziare, seguendo il principio di inversione delle dipendenze.

Simple Factory

La *Simple Factory* è una semplificazione molto diffusa di Abstract Factory che si occupa banalmente di creare oggetti implementandone la logica di creazione.

EX.

```
// Interfaccia comune per tutte le pizze
public interface Pizza {
    void prepare();
    void bake();
    void cut();
    void box();
}

// Implementazione concreta di una Pizza Margherita
public class MargheritaPizza implements Pizza {
    @Override
    public void prepare() {
        System.out.println("Preparing Margherita Pizza...");
    }

    @Override
    public void bake() {
        System.out.println("Baking Margherita Pizza...");
    }
}
```

```
@Override
public void cut() {
    System.out.println("Cutting Margherita Pizza...");
}

@Override
public void box() {
    System.out.println("Boxing Margherita Pizza...");
}
}

// Implementazione concreta di una Pizza al Salame
public class SalamiPizza implements Pizza {
    @Override
    public void prepare() {
        System.out.println("Preparing Salami Pizza...");
    }

    @Override
    public void bake() {
        System.out.println("Baking Salami Pizza...");
    }

    @Override
    public void cut() {
        System.out.println("Cutting Salami Pizza...");
    }

    @Override
    public void box() {
        System.out.println("Boxing Salami Pizza...");
    }
}

// Simple Factory per creare pizze
public class PizzaFactory {
    public Pizza createPizza(String type) {
        if (type.equalsIgnoreCase("Margherita")) {
            return new MargheritaPizza();
        } else if (type.equalsIgnoreCase("Salami")) {
            return new SalamiPizza();
        } else {

```

```

        throw new IllegalArgumentException("Unknown pizza type: " +
type);
    }
}
}

// Classe client
public class Pizzeria {
    private final PizzaFactory pizzaFactory;

    public Pizzeria(PizzaFactory pizzaFactory) {
        this.pizzaFactory = pizzaFactory;
    }

    public void orderPizza(String type) {
        Pizza pizza = pizzaFactory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
    }
}

// Metodo main per testare il sistema
public class Main {
    public static void main(String[] args) {
        PizzaFactory factory = new PizzaFactory();
        Pizzeria pizzeria = new Pizzeria(factory);

        System.out.println("Ordering a Margherita Pizza:");
        pizzeria.orderPizza("Margherita");

        System.out.println("\nOrdering a Salami Pizza:");
        pizzeria.orderPizza("Salami");
    }
}

```

Con questa struttura, è possibile aggiungere nuovi tipi di pizze modificando solo la **factory**, senza toccare il codice della pizzeria o del client.

Factory Method

Factory method è un altro design pattern di factories, il quale si basa sull'uso dell'**ereditarietà** per creare oggetti.

Riprendiamo l'esempio di prima.

```
// Classe astratta PizzaStore che utilizza il Factory Method
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type); // Factory Method
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

    // Factory Method: deve essere implementato dalle sottoclassi
    protected abstract Pizza createPizza(String type);
}

// Implementazione concreta di un New York Pizza Store
public class NewYorkPizzaStore extends PizzaStore {
    @Override
    protected Pizza createPizza(String type) {
        if (type.equalsIgnoreCase("Margherita")) {
            return new MargheritaPizza();
        } else if (type.equalsIgnoreCase("Salami")) {
            return new SalamiPizza();
        } else {
            throw new IllegalArgumentException("Unknown pizza type: " +
type);
        }
    }
}

// Implementazione concreta di un Chicago Pizza Store
public class ChicagoPizzaStore extends PizzaStore {
    @Override
    protected Pizza createPizza(String type) {
        if (type.equalsIgnoreCase("Margherita")) {
            return new MargheritaPizza(); // Può essere una variante di
Margherita Pizza
        } else if (type.equalsIgnoreCase("Salami")) {
```

```

        return new SalamiPizza(); // Può essere una variante di
    Salami Pizza
    } else {
        throw new IllegalArgumentException("Unknown pizza type: " +
type);
    }
}

// Classe Main per testare il sistema
public class Main {
    public static void main(String[] args) {
        PizzaStore nyStore = new NewYorkPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        System.out.println("Ordering a Margherita Pizza from New York
Store:");
        nyStore.orderPizza("Margherita");

        System.out.println("\nOrdering a Salami Pizza from Chicago
Store:");
        chicagoStore.orderPizza("Salami");
    }
}

```

Il *factory method pattern* prevede di implementare la logica di creazione della pizza non all'interno di una factory, ma all'interno di ogni sottoclasse di `PizzaStore`, lasciando invariati gli altri metodi (`prepare()`, `bake()`, etc.) che rimangono gli stessi per tutte le sottoclassi e sono quindi definiti nella superclasse `PizzaStore`.

Abstract Factory

Le *Abstract Factory* sono utili quando vogliamo creare famiglie di oggetti correlati senza specificare le loro classi concrete.

Questo pattern è un'estensione del concetto di Factory, ma è più potente e generalizzato.

Idea di base:

1. Abbiamo un'interfaccia (*Abstract Factory*) che definisce i metodi per creare famiglie di oggetti.

2. Ogni implementazione concreta di questa interfaccia (le *Concrete Factory*) fornisce versioni specifiche degli oggetti.
3. Il client usa l'interfaccia senza preoccuparsi delle classi concrete che vengono effettivamente create.

```
// Interfaccia per le factory degli ingredienti
public interface PizzaIngredientFactory {
    Dough createDough();
    Sauce createSauce();
    Cheese createCheese();
    Veggies[] createVeggies();
    Pepperoni createPepperoni();
    Clams createClams();
}

// Implementazione concreta della factory per Chicago
public class ChicagoPizzaIngredientFactory implements
PizzaIngredientFactory {
    @Override
    public Dough createDough() {
        return new ThickCrustDough();
    }

    @Override
    public Sauce createSauce() {
        return new PlumTomatoesSauce();
    }

    @Override
    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    @Override
    public Veggies[] createVeggies() {
        Veggies[] veggies = {new BlackOlives(), new Spinach(), new
Eggplant()};
        return veggies;
    }

    @Override
    public Pepperoni createPepperoni() {
```

```
        return new SlicedPepperoni();
    }

    @Override
    public Clams createClams() {
        return new FrozenClams();
    }
}

// Classe astratta per le pizze
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies[] veggies;
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clams;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }
}

// Pizza concreta: CheesePizza
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    @Override
    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        veggies = ingredientFactory.createVeggies();
        pepperoni = ingredientFactory.createPepperoni();
        clams = ingredientFactory.createClams();
    }
}
```

```
        cheese = ingredientFactory.createCheese();
    }

}

// Classe astratta per il PizzaStore
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        System.out.println("---- Making a " + pizza.getName() + " ----");
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        return pizza;
    }

    // Factory Method: delega la creazione alle sottoclassi
    protected abstract Pizza createPizza(String type);
}

// Implementazione concreta del PizzaStore per Chicago
public class ChicagoPizzaStore extends PizzaStore {
    @Override
    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory = new
ChicagoPizzaIngredientFactory();

        if (item.equals("cheese")) {
            pizza = new CheesePizza(ingredientFactory);
        }
        // Aggiungere altre varianti se necessario
        return pizza;
    }
}

// Classe Main per testare il sistema
public class Main {
    public static void main(String[] args) {
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        System.out.println("Ordering a Cheese Pizza from Chicago
Store:");
        Pizza pizza = chicagoStore.orderPizza("cheese");
```

```

        System.out.println("Ordered a " + pizza.getName() + "\n");
    }
}

```

Nell'*abstract factory* una *classe delega la responsabilità della creazione di un oggetto ad una factory*.

ALTRO EX.

```

// ABSTRACT FACTORY
public interface FurnitureFactory {
    Chair createChair();
    Table createTable();
    Sofa createSofa();
}

// Factory concreta per mobili moderni
public class ModernFurnitureFactory implements FurnitureFactory {
    public Chair createChair() {
        return new ModernChair();
    }
    public Table createTable() {
        return new ModernTable();
    }
    public Sofa createSofa() {
        return new ModernSofa();
    }
}

// Factory concreta per mobili classici
public class ClassicFurnitureFactory implements FurnitureFactory {
    public Chair createChair() {
        return new ClassicChair();
    }
    public Table createTable() {
        return new ClassicTable();
    }
    public Sofa createSofa() {
        return new ClassicSofa();
    }
}

//CLASSI DEI PRODOTTI

```

```
public interface Chair {
    void sitOn();
}

public class ModernChair implements Chair {
    public void sitOn() {
        System.out.println("Seduto su una sedia moderna!");
    }
}

public class ClassicChair implements Chair {
    public void sitOn() {
        System.out.println("Seduto su una sedia classica!");
    }
}

// Allo stesso modo, definiamo Table e Sofa per stile moderno e
// classico.
...
...

public class FurnitureStore {
    private FurnitureFactory factory;

    public FurnitureStore(FurnitureFactory factory) {
        this.factory = factory;
    }

    public void furnish() {
        Chair chair = factory.createChair();
        Table table = factory.createTable();
        Sofa sofa = factory.createSofa();

        chair.sitOn();
        System.out.println("Tavolo e divano forniti.");
    }
}

public class Main {
    public static void main(String[] args) {
        FurnitureFactory modernFactory = new ModernFurnitureFactory();
        FurnitureStore modernStore = new FurnitureStore(modernFactory);
        modernStore.furnish();
    }
}
```

```
        FurnitureFactory classicFactory = new ClassicFurnitureFactory();
        FurnitureStore classicStore = new
FurnitureStore(classicFactory);
        classicStore.furnish();
    }
}
```

Riassumendo

Simple Factory

La Simple Factory è una classe responsabile della creazione di oggetti specifici. La logica di creazione è centralizzata in questa classe, che decide quale oggetto istanziare in base a un parametro o a una richiesta del client. Questo approccio semplifica il processo di creazione e nasconde al client i dettagli specifici di come gli oggetti vengono costruiti. Tuttavia, non è considerata un vero e proprio design pattern formale, poiché non aderisce sempre al principio *Open/Closed*, rendendo necessario modificare la factory stessa per aggiungere nuovi tipi di oggetti.

Factory Method

Il Factory Method delega la creazione degli oggetti alle sottoclassi, utilizzando un metodo astratto definito nella superclasse. Questo metodo, implementato nelle sottoclassi, consente di creare oggetti specifici a seconda del contesto. La superclasse definisce la struttura generale del processo (es. *orderPizza*), mentre le sottoclassi gestiscono i dettagli specifici di creazione (es. *createPizza*). Questo approccio offre maggiore flessibilità ed estensibilità rispetto alla Simple Factory, permettendo di aggiungere nuove logiche di creazione senza modificare il codice esistente. Tuttavia, introduce maggiore complessità, aumentando il numero di sottoclassi.

Abstract Factory

L'Abstract Factory fornisce un'interfaccia per creare famiglie di oggetti correlati senza specificare le loro classi concrete. La factory astratta definisce i metodi per creare vari tipi di oggetti, mentre le factory concrete implementano questi metodi per produrre specifiche versioni degli oggetti. Questo pattern è particolarmente utile

quando è necessario creare oggetti interdipendenti, ad esempio componenti di una GUI con stili diversi. Il client lavora con l'interfaccia astratta, mantenendo il codice indipendente dalle implementazioni concrete. L'Abstract Factory promuove l'aderenza ai principi *Open/Closed* e *Single Responsibility*, ma può diventare complesso da gestire quando si devono supportare molte famiglie di oggetti.

Capitolo 15

Singleton e State

Singleton

Il *singleton* pattern assicura che di **una classe** vi sia solo **un'istanza**, fornendo un **punto d'accesso globale ad essa**. Ciò è particolarmente *utile quando avere più di un'istanza porterebbe a possibili problemi di sincronizzazione*.

Per fare in modo che una classe non possa essere istanziata più di una volta, si può procedere nel seguente modo:

1. Crea un singolo costruttore privato.
2. Rendi l'unica istanza disponibile attraverso un metodo pubblico che chiama indirettamente il costruttore.

Eager Initialization

```
public class ChocolateBoiler {  
  
    private ChocolateBoiler() {} //Costruttore vuoto  
    private static ChocolateBoiler chocoboiler = new  
    ChocolateBoiler();  
  
    public static ChocolateBoiler GetChocolateBoiler(){  
        return chocoboiler;  
    }  
}
```

Semplice e sicuro in ambienti single-threaded.

Non richiede logiche aggiuntive per garantire il controllo della creazione

dell'istanza.

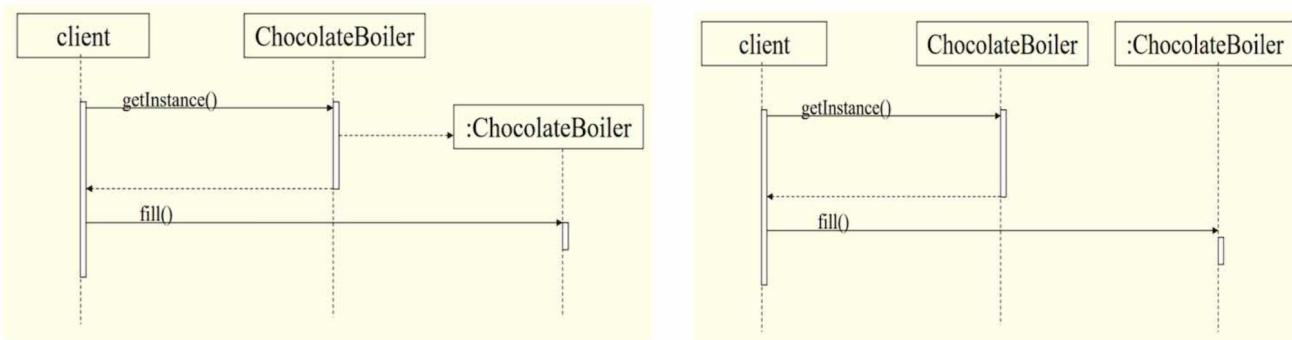
L'istanza è creata anche se non viene mai utilizzata, sprecando memoria se non necessaria.

Lazy Initialization

```
public class ChocolateBoiler {  
  
    private ChocolateBoiler() {} //Costruttore vuoto  
    private static ChocolateBoiler chocoboiler = new  
ChocolateBoiler();  
  
    public static ChocolateBoiler GetChocolateBoiler(){  
        if(chocoboiler == null){  
            chocoboiler = new ChocolateBoiler();  
        }  
        return chocoboiler;  
    }  
}
```

Qui l'istanza viene creata solo quando è effettivamente necessaria. Questo approccio può migliorare l'efficienza quando il Singleton non è sempre richiesto.

Non è thread-safe: in un ambiente multi-thread, due thread potrebbero creare due istanze separate contemporaneamente.



Per evitare problemi di *race condition* con la *lazy initialization* del Singleton in ambienti multithread, esistono tre soluzioni principali:

1. **Eager Initialization**: crea l'istanza subito, evitando problemi di concorrenza.
Potrebbe sprecare risorse se l'istanza non viene utilizzata.

2. **Metodo sincronizzato**: sincronizza l'accesso a `GetChocolateBoiler()`, ma introduce un overhead significativo, soprattutto con molte richieste.
3. **Double-Checked Locking**: utilizza un controllo doppio e sincronizzazione solo quando necessario, migliorando l'efficienza ma con maggiore complessità.

Double-Checked Locking

```
public class ChocolateBoiler {  
    private static volatile ChocolateBoiler chocolateBoiler;  
  
    private ChocolateBoiler() {}  
  
    public static ChocolateBoiler GetChocolateBoiler() {  
        if (chocolateBoiler == null) {  
            synchronized (ChocolateBoiler.class) {  
                if (chocolateBoiler == null) {  
                    chocolateBoiler = new ChocolateBoiler();  
                }  
            }  
        }  
        return chocolateBoiler;  
    }  
}
```

Per garantire che ci sia un solo oggetto anche quando si desiderano più *sottoclassi* di un `Singleton`, si può utilizzare una struttura come quella mostrata nell'esempio sottostante:

1. Si definisce una classe astratta `Singleton` con un'istanza statica condivisa (`uniqueInstance`) e un metodo astratto `instance()`.
2. Le sottoclassi di `Singleton` implementano il metodo `instance()`, che crea l'istanza solo se `uniqueInstance` è `null`:

- Se sì, viene creata un'istanza della sottoclasse.
- Se no, viene restituita l'istanza già esistente.

In questo modo, si assicura che esista sempre e solo un'istanza. Tuttavia, questa struttura implica che *solo una sottoclasse possa essere attiva contemporaneamente*, poiché l'istanza è condivisa tra tutte le sottoclassi.

```
public abstract class Singleton {  
    protected static Singleton uniqueInstance = null;
```

```

        public abstract static Singleton instance();
    }

public class SpecSingleton extends Singleton {
    private SpecSingleton() { /* Costruttore privato */ }

    public static Singleton instance() {
        if (uniqueInstance == null) {
            uniqueInstance = new SpecSingleton();
        }
        return uniqueInstance;
    }
}

public class OtherSpecSingleton extends Singleton {
    private OtherSpecSingleton() { /* Costruttore privato */ }

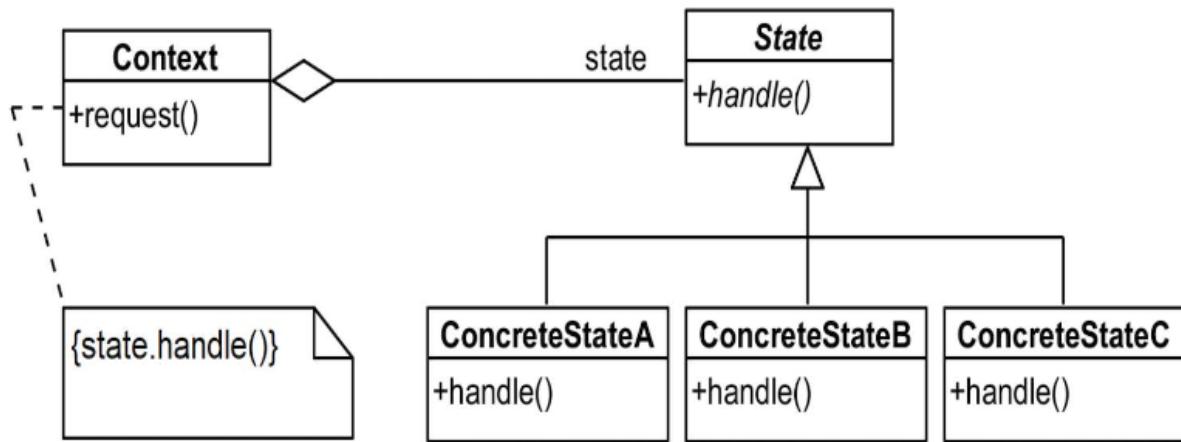
    public static Singleton instance() {
        if (uniqueInstance == null) {
            uniqueInstance = new OtherSpecSingleton();
        }
        return uniqueInstance;
    }
}

```

State

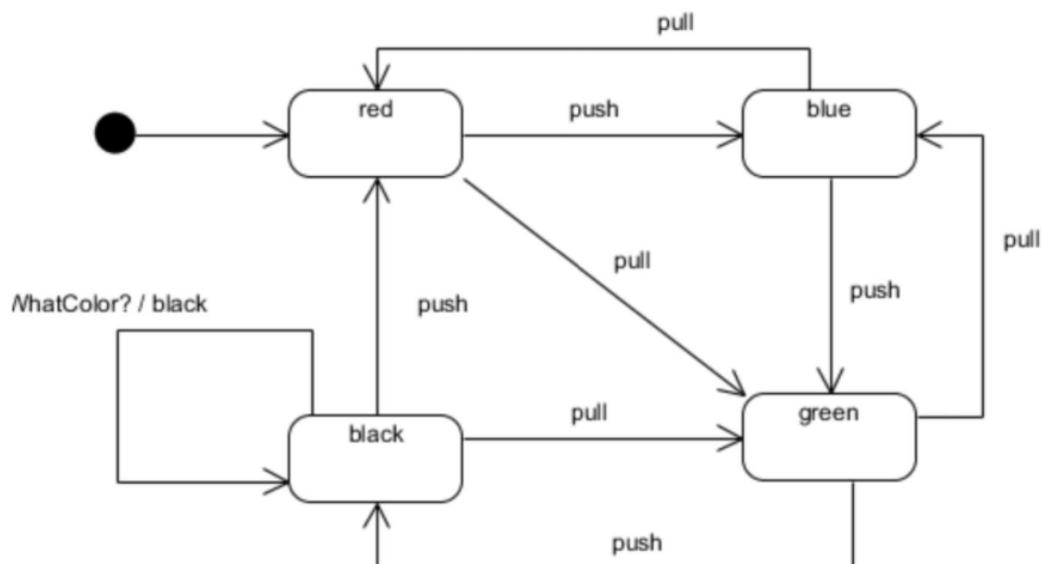
Il *State Pattern* permette a un oggetto di **alterare il suo comportamento** quando il suo **stato interno cambia**. In pratica, l'oggetto si comporta in modo diverso a seconda del suo stato.

Struttura:



1. **Context**: È l'oggetto principale che ha un comportamento che dipende dallo stato. Il comportamento è delegato all'oggetto che implementa l'interfaccia State.
2. **State (interfaccia)**: Contiene i metodi che definiscono il comportamento.
3. **ConcreteState**: Implementa l'interfaccia State e fornisce una specifica implementazione dei metodi per un determinato stato.

EX.



```
public abstract class State {  
    public abstract void handlePush(Context c);  
    public abstract void handlePull(Context c);  
    public abstract Color getColor();
```

```
}

public class BlackState extends State {

    public void handlePush(Context c) {
        c.setState(new RedState());
    }

    public void handlePull(Context c) {
        c.setState(new GreenState());
    }

    public Color getColor() {
        return Color.BLACK;
    }

}

// Implementati anche per gli altri stati con i rispettivi
`handlePush()` e `handlePull()` (RedState, BlueState e GreenState)

public class Context {
    private State state; // Variabile che contiene lo stato attuale e
    che verrà modificata quando handlePush o handlePull verranno chiamati
    sul contesto.

    // Costruttore che permette di impostare lo stato iniziale
    public Context(State state) {
        this.state = state;
    }

    // Costruttore di default che imposta lo stato iniziale su RedState
    public Context() {
        this(new RedState()); // Stato di default è RedState
    }

    public State getState() {
        return state;
    }

    public void setState(State state) {
        this.state = state;
    }
}
```

```
// Metodo per eseguire l'azione "push" in base allo stato attuale
public void push() {
    state.handlePush(this); // Invoca il metodo del comportamento
    specifico dello stato prendendo l'intero `Context c` come parametro
}

// Metodo per eseguire l'azione "pull" in base allo stato attuale
public void pull() {
    state.handlePull(this); // Invoca il metodo del comportamento
    specifico dello stato prendendo l'intero `Context c` come parametro
}

}
```

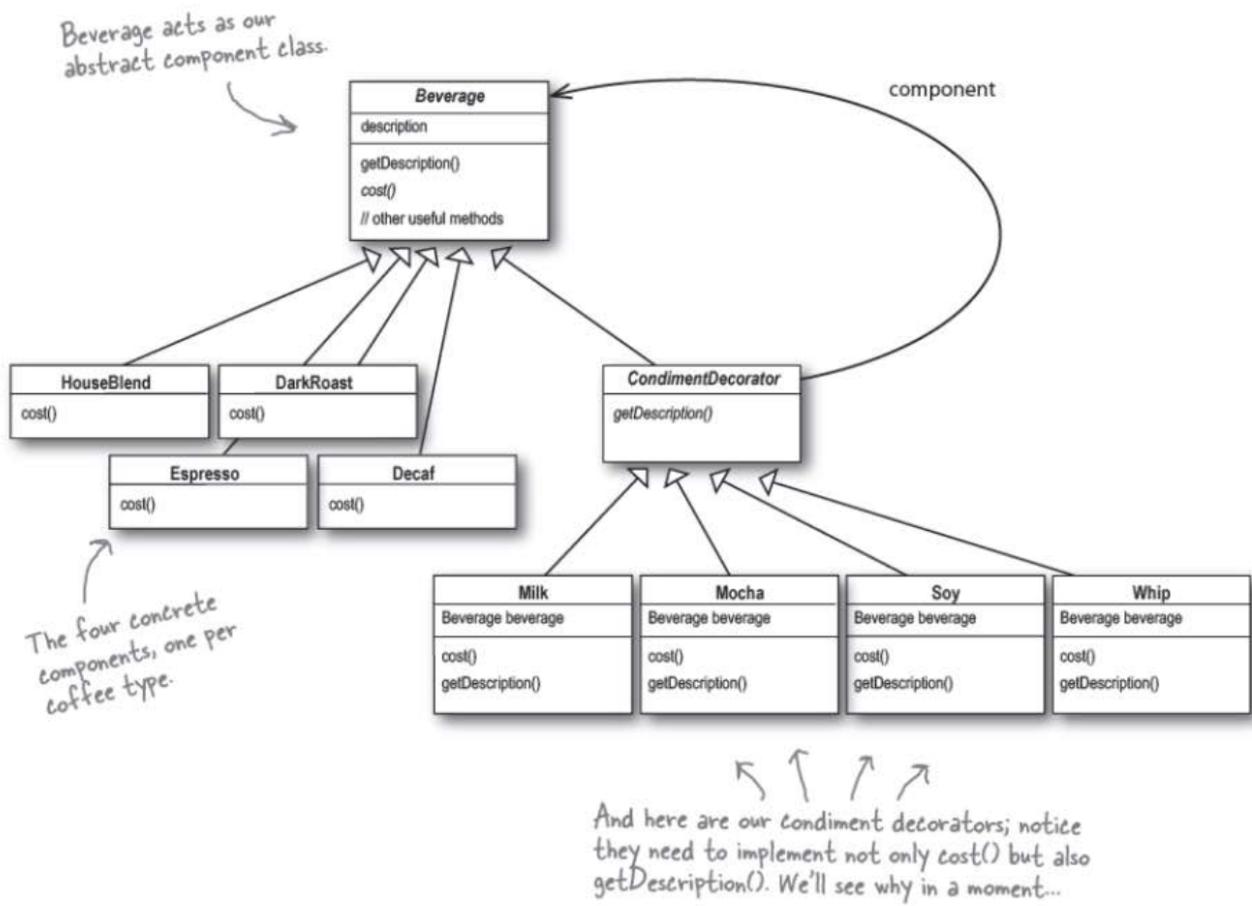
Lo stato viene cambiato nel pattern State tramite il metodo `setState()` della classe `Context`. Ogni stato concreto (come `BlackState` o `RedState`) implementa la logica per cambiare lo stato quando viene invocato un metodo come `handlePush()` o `handlePull()`. Quando uno di questi metodi viene chiamato, lo stato corrente del contesto viene sostituito con un nuovo stato, modificando il comportamento dell'oggetto.

Capitolo 16

Decorator, Adapter, Proxy

Decorator

Il *Decorator* è un design pattern che permette di aggiungere dinamicamente nuove responsabilità a un oggetto, estendendone le funzionalità senza modificare la sua struttura interna. Questo pattern è utile per evitare l'uso massiccio delle sottoclassi quando si vogliono aggiungere nuove caratteristiche a oggetti.



Il pattern si basa su una classe astratta che rappresenta il **componente**, e una serie di **decorator** che **aggiungono nuove funzionalità** estendendo il componente originale.

```

// Classe base Pizza
public abstract class Pizza {
    public abstract String getDescription();
    public abstract double cost();
}

// Pizza semplice
public class Margherita extends Pizza {
    public String getDescription() {
        return "Margherita";
    }

    public double cost() {
        return 5.00; // Prezzo base della pizza Margherita
    }
}

```

```
// Decorator astratto
public abstract class PizzaDecorator extends Pizza {
    protected Pizza pizza;

    public PizzaDecorator(Pizza pizza) {
        this.pizza = pizza;
    }

    public abstract String getDescription();
}

// Decorator per aggiungere formaggio extra
public class CheeseDecorator extends PizzaDecorator {
    public CheeseDecorator(Pizza pizza) {
        super(pizza);
    }

    public String getDescription() {
        return pizza.getDescription() + " + Cheese";
    }

    public double cost() {
        return pizza.cost() + 2.00; // Aggiungiamo il costo del
formaggio
    }
}

// Decorator per aggiungere olive
public class OliveDecorator extends PizzaDecorator {
    public OliveDecorator(Pizza pizza) {
        super(pizza);
    }

    public String getDescription() {
        return pizza.getDescription() + " + Olives";
    }

    public double cost() {
        return pizza.cost() + 1.50; // Aggiungiamo il costo delle olive
    }
}

// Classe principale per testare il funzionamento
```

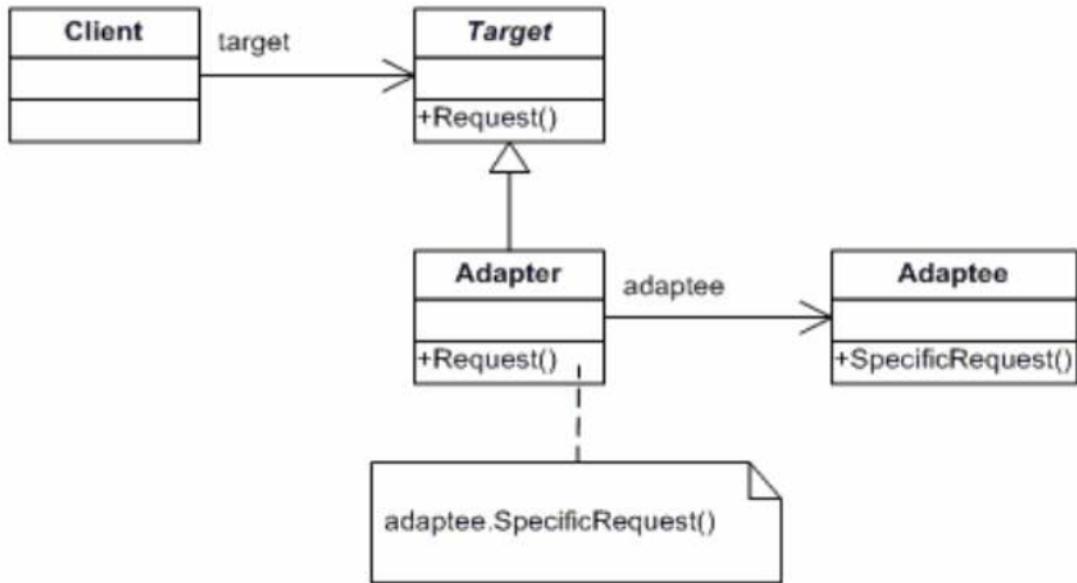
```
public class PizzaShop {  
    public static void main(String[] args) {  
        // Crea una pizza Margherita di base  
        Pizza pizza = new Margherita();  
        System.out.println(pizza.getDescription() + " - $" +  
pizza.cost());  
  
        // Aggiungi formaggio extra  
        pizza = new CheeseDecorator(pizza);  
        System.out.println(pizza.getDescription() + " - $" +  
pizza.cost());  
  
        // Aggiungi olive  
        pizza = new OliveDecorator(pizza);  
        System.out.println(pizza.getDescription() + " - $" +  
pizza.cost());  
    }  
}
```

Il *Decorator* può essere confuso con lo *Strategy* pattern, ma la differenza principale è che nel decorator, gli oggetti decorati mantengono il controllo sul comportamento senza necessitare di modifiche al componente originale, mentre nello strategy il comportamento viene separato e incapsulato in oggetti esterni.

Il decorator è più flessibile quando si desidera aggiungere funzionalità in modo dinamico, mentre lo strategy è utile per configurare comportamenti variabili senza alterare l'oggetto principale.

Adapter

L'*Adapter* pattern permette di trasformare un'interfaccia in un'altra equivalente, la quale è conforme ad un certo standard che si vuole garantire.



```

// Interfaccia che il client si aspetta
interface Printer {
    void print(String text);
}

// Classe che fornisce un metodo di stampa diverso
class OldPrinter {
    public void printDocument(String text) {
        System.out.println("Stampa documento: " + text);
    }
}

// Adapter che rende OldPrinter compatibile con Printer
class PrinterAdapter implements Printer {
    private OldPrinter oldPrinter;

    public PrinterAdapter(OldPrinter oldPrinter) {
        this.oldPrinter = oldPrinter;
    }

    @Override
    public void print(String text) {
        // Adatta la chiamata al vecchio metodo
        oldPrinter.printDocument(text);
    }
}
  
```

```

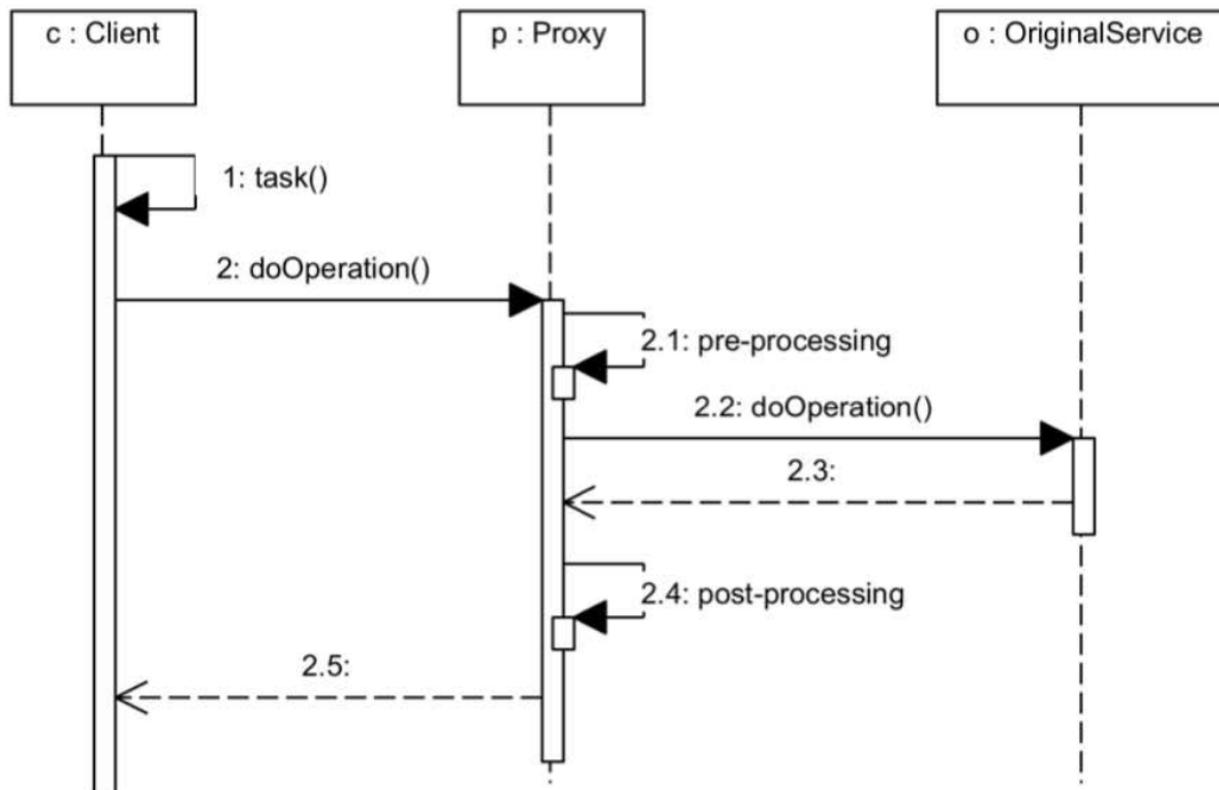
// Client che usa l'interfaccia Printer
public class AdapterExample {
    public static void main(String[] args) {
        OldPrinter oldPrinter = new OldPrinter();
        Printer printer = new PrinterAdapter(oldPrinter); // Creiamo un
adattatore per la vecchia stampante

        printer.print("Hello, Adapter Pattern!"); // Ora possiamo usare
il nuovo sistema di stampa
    }
}

```

Proxy

Il *Proxy Pattern* è un design pattern strutturale che fornisce un surrogato (proxy) di un oggetto per controllarne l'accesso. In altre parole, un proxy agisce come intermediario tra un client e l'oggetto reale, permettendo di eseguire operazioni aggiuntive, come il controllo dell'accesso, la gestione della memoria, il caricamento lazy (quando l'oggetto è creato solo quando necessario) o altre operazioni di gestione. Il proxy può quindi “proteggere” o “gestire” l'oggetto reale, aggiungendo un livello di controllo.



Esistono diverse tipologie di proxy, ognuna con uno scopo specifico:

1. **Remote Proxy**: Rappresenta un oggetto che si trova su una macchina remota, gestendo la comunicazione tra il client e l'oggetto remoto.
 2. **Cache Proxy**: Memorizza nella cache le risposte a richieste precedenti, evitando di ripetere la stessa richiesta e migliorando le prestazioni.
 3. **Synchronization Proxy**: Gestisce gli accessi concorrenti a un oggetto, prevenendo conflitti e garantendo la sincronizzazione tra più thread o processi.
 4. **Virtual Proxy**: Rappresenta un oggetto costoso da creare, ritardando la sua creazione fino al momento in cui è realmente necessario, risparmiando risorse.
 5. **Firewall**: Proxy di protezione che regola l'accesso a determinati servizi e controlla il traffico in entrata/uscita.
-

Capitolo 18

Verifica e validazione

Durante lo sviluppo software, bisogna anche essere sicuri che ciò che stiamo costruendo funzioni come previsto

Parallelamente al processo di sviluppo software, bisogna anche eseguire verifica e convalida di quello che si fa

La **verifica** si concentra sul “costruire il sistema come previsto”.

La **validazione** risponde alla domanda “stiamo costruendo il sistema giusto per l’utente?”.

In merito a ciò, ci facciamo 5 domande:

1. **Quando iniziare verifica e convalida?**: V&V **non sono attività finali**, ma continuano per tutta la vita del software, iniziando sin dalle prime fasi di sviluppo.
2. **Tecniche di testing**: **Non esiste una tecnica unica di testing**; è fondamentale combinare diverse tecniche per affrontare vari tipi di difetti, applicandole in fasi diverse del processo di sviluppo.
3. **Come valutare se un prodotto è pronto?**: Utilizzando misure come **disponibilità**, **MTBF** (tempo medio tra i guasti) e **affidabilità**, che indicano la qualità del sistema.

Vengono anche eseguiti **Alfa test** (eseguiti in ambiente controllato) e **Beta test** (sul campo da utenti reali).

4. *Controllo della qualità nelle release future*: Si eseguono **test** su nuovo codice, ri-esecuzioni dei **test** di sistema e **test** di regressione per garantire la qualità continua.
5. *Miglioramento del processo di sviluppo*: Identificando e **rimuovendo i punti deboli** nei processi di sviluppo e di test, che possono portare a difetti ricorrenti.

Terminologia IEEE usate in ambito di testing:

- *Malfunzionamento*: **comportamento errato del sistema** durante l'esecuzione, che non corrisponde alle specifiche.
- *Difetto*: **errore nel codice** che può causare un malfunzionamento.
- *Errore*: **causa di un difetto**, tipicamente dovuta a incomprensioni o errori umani.

Il *testing completo* è generalmente **impossibile**, poiché non è praticabile testare ogni possibile input.

Il test di un programma può rilevare la presenza di difetti, ma non dimostrarne l'assenza.

Tecniche di verifica statica

La *verifica statica* **non prevede l'esecuzione** del programma.

Essa può essere effettuata mediante:

- *Metodi manuali*: basati sulla **lettura del codice** (desk-check). Di fatto più comunemente usati, anche se meno rigorosi e documentabili.
- *Metodi formali*: supportati da **strumenti** di model checking, esecuzione simbolica, interpretazione astratta e theorem proving.

Metodi Manuali

Vi sono due metodi di lettura del codice:

- *Inspection*: Metodo di lettura mirata del codice, condotto seguendo una lista di controllo. Lo scopo principale è identificare difetti nel codice. La strategia utilizzata è quella dell'**error guessing**, che si concentra su errori comuni e ben definiti (ex. off-by-one errors).

Chi effettua l'inspection è di solito una persona esterna al team di sviluppo. Il processo si svolge in quattro fasi:

- 1. Pianificazione**
- 2. Definizione della lista di controllo**
- 3. Lettura del codice**
- 4. Correzione dei difetti**

La lista di controllo contiene elementi che non possono essere verificati automaticamente, ed è aggiornata a ogni ciclo di inspection.

Java Checklist: Level 1 inspection (single-pass read-through, context independent)			
FEATURES (where to look and how to check):			
Item (what to check)			
FILE HEADER: Are the following items included and consistent?	yes	no	comments
Author and current maintainer identity			
Cross-reference to design entity			
Overview of package structure, if the class is the principal entry point of a package			
FILE FOOTER: Does it include the following items?	yes	no	comments
Revision log to minimum of 1 year or at least to most recent point release, whichever is longer			
IMPORT SECTION: Are the following requirements satisfied?	yes	no	comments
Brief comment on each import with the exception of standard set: java.io.* , java.util.*			
Each imported package corresponds to a dependence in the design documentation			
CLASS DECLARATION: Are the following requirements satisfied?	yes	no	comments
The visibility marker matches the design document			
The constructor is explicit (if the class is not static)			
The visibility of the class is consistent with the design document			
CLASS DECLARATION JAVADOC: Does the Javadoc header include:	yes	no	comments
One sentence summary of class functionality			
Guaranteed invariants (for data structure classes)			
Usage instructions			
CLASS: Are names compliant with the following rules?	yes	no	comments
Class or interface: CapitalizedWithEachInternalWordCapitalized			
Special case: If class and interface have same base name, distinguish as ClassNameIfc and ClassNameImpl			
Exception: ClassNameEndsWithException			
Constants (final): ALL_CAPS_WITH_UNDERSCORES			
Field name: capsAfterFirstWord.			
name must be meaningful outside of context			
IDIOMATIC METHODS: Are names compliant with the following rules?	yes	no	comments
Method name: capsAfterFirstWord			
Local variables: capsAfterFirstWord.			
Name may be short (e.g., i for an integer) if scope of declaration and use is less than 30 lines.			
Factory method for X: newX			
Converter to X: toX			
Getter for attribute x: getX();			
Setter for attribute x: void setX			

Java Checklist: Level 2 inspection (comprehensive review in context)			
FEATURES (where to look and how to check):			
Item (what to check)			
DATA STRUCTURE CLASSES: Are the following requirements satisfied?	yes	no	comments
The class keeps a design secret			
The substitution principle is respected: Instance of class can be used in any context allowing instance of superclass or interface			
Methods are correctly classified as constructors, modifiers, and observers			
There is an abstract model for understanding behavior			
The structural invariants are documented			
FUNCTIONAL (STATELESS) CLASSES: Are the following requirements satisfied?	yes	no	comments
The substitution principle is respected: Instance of class can be used in any context allowing instance of superclass or interface			
METHODS: Are the following requirements satisfied?	yes	no	comments
The method semantics are consistent with similarly named methods. For example, a "put" method should be semantically consistent with "put" methods in standard data structure libraries			
Usage examples are provided for nontrivial methods			
FIELDS: Are the following requirements satisfied?	yes	no	comments
The field is necessary (cannot be a method-local variable)			
Visibility is protected or private, or there is an adequate and documented rationale for public access			
Comment describes the purpose and interpretation of the field			
Any constraints or invariants are documented in either field or class comment header			
DESIGN DECISIONS: Are the following requirements satisfied?	yes	no	comments
Each design decision is hidden in one class or a minimum number of closely related and co-located classes			
Classes encapsulating a design decision do not unnecessarily depend on other design decisions			
Adequate usage examples are provided, particularly of idiomatic sequences of method calls			
Design patterns are used and referenced where appropriate			
If a pattern is referenced: The code corresponds to the documented pattern			

- **Walkthrough:** Pur avendo lo stesso obiettivo di rilevare difetti, prevede una lettura critica del codice, in cui il codice viene “simulato” per verificarne il funzionamento, piuttosto che analizzare singole sezioni specifiche. Il processo si svolge in tre fasi:
 - 1. Pianificazione**
 - 2. Lettura del codice**
 - 3. Correzione dei difetti**

Metodi Formali

I metodi formali di verifica del codice usano tecniche basate su:

1. dimostrazione formale di correttezza di un modello finito;
2. istanziazione del modello.

EX.

Il protocollo *two-phase locking* si dimostra corretto e, se istanziato correttamente, garantisce assenza di malfunzionamenti dovuti alla race condition.

Capitolo 19

Progettazione della fase di test

La *verifica dinamica*, o testing, consiste nell'**esecuzione del programma per controllare il suo corretto funzionamento**.

Questo processo si articola in quattro fasi principali:

1. Progettazione dei test;
2. Esecuzione del codice;
3. Analisi dei risultati;
4. Debugging per risolvere eventuali problemi rilevati.

Per una buona progettazione dei test, è consigliabile:

- Realizzare *test a diversi livelli* dello sviluppo e da prospettive differenti, includendo il punto di vista dell'utente.
- Garantire la *ripetibilità dei test*, in modo che possano essere rieseguiti nelle stesse condizioni iniziali.

Progettazione dei casi di test

Un caso di test è una tripla di:

$$(input, output atteso, ambiente)$$

Dove l' **output** è quello atteso avendo fornito l' **input** nell' **ambiente** prestabilito.

Un insieme di casi di test costituisce una *test suite*, che viene utilizzata in una procedura di prova composta da diverse fasi:

- Progettare i test.
- Creare l'ambiente necessario.
- Eseguire i casi di test.
- Analizzarne i risultati.
- Valutarne l'efficacia.

Non è possibile dimostrare la totale correttezza di un sistema, ma si può valutare quanto la test suite sia in grado di individuare difetti.

I *criteri di adeguatezza* aiutano a **misurare l'efficacia dei test**. Ad esempio, si valuta se tutte le istruzioni del codice vengono eseguite almeno una volta e se i risultati corrispondono a quelli attesi.

Una test suite è considerata adeguata se tutti i test sono completati con successo e ogni obbligo di test (*test obligation*) è rispettato.

Le *test obligations* sono definite considerando diversi aspetti:

- **Funzionalità richieste:** Verificare che un metodo calcoli correttamente il valore assoluto di un numero.
- **Struttura del codice:** Assicurarsi che ogni ciclo venga eseguito almeno una volta.
- **Modelli usati per progettare il sistema:** Controllare tutte le transizioni in un modello di protocollo.
- **Difetti ipotetici comuni:** Testare la gestione di input molto grandi per verificare eventuali buffer overflow.

Questi criteri consentono di progettare test più completi e mirati, sia ignorando la struttura interna del sistema (*black-box testing*), sia basandosi sulla struttura del codice (*white-box testing*).

Metodi black-box

I *metodi black-box* generano valori di input ai test case basandosi sulle **funzionalità del sistema, senza considerare la struttura interna del codice**. La strategia generale prevede di separare le funzionalità da testare (ad esempio, usando i casi d'uso) e derivare test specifici per ciascuna funzionalità.

Metodo statistico

I test case vengono selezionati considerando la distribuzione di probabilità dei dati di ingresso. Questo approccio mira a esercitare il programma sui valori di input più probabili per il suo utilizzo reale. Sebbene la generazione automatica sia semplice con una distribuzione nota, questo metodo può risultare costoso nel calcolare l'output atteso ([problema dell'oracolo](#)).

Il [problema dell'oracolo](#) è una difficoltà che si incontra durante il testing del software e riguarda l'identificazione dell'output corretto (atteso) di un programma per un determinato input. In altre parole, un [oracolo](#) è un'entità che determina se l'output prodotto dal sistema sottoposto a test è corretto o meno.

EX.

Esempio: Selezionare i valori per un input che rappresenta «età il giorno della laurea» (tipo int)

- **Test obligation** generate con il metodo statistico
 - tutti i valori compresi tra 20 e 27
 - Il 40% dei valori tra 27 e 35, scelti in modo casuale
 - Il 5% dei valori tra 36 e 100, scelti in modo casuale
- **Casi di test** che soddisfano le test obligation
 - $\langle 20, \cdot \rangle, \langle 21, \cdot \rangle, \dots, \langle 27, \cdot \rangle, \langle 29, \cdot \rangle, \dots, \langle 51, \cdot \rangle, \dots$
 - Non specificano ancora output atteso e ambiente

Classi di equivalenza

Gli input vengono raggruppati in [classi di equivalenza](#), ovvero **insiemi di valori che dovrebbero produrre lo stesso comportamento del programma** (non stesso output necessariamente).

Questo criterio è economico solo per programmi con un numero limitato di comportamenti distinti rispetto alle configurazioni di input. Tuttavia, la deduzione che il corretto funzionamento per un rappresentante valga per l'intera classe dipende dalla realizzazione del programma e non è garantita a priori.

EX.

Supponiamo di avere il metodo [intcalcolaTasse\(intreddito\)](#) che segue il

seguente schema

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

È ragionevole qui definire la *test obligation*:

- Un test case per ogni aliquota.

Metodo dei valori limite

I *test case si concentrano sui valori limite*, considerando gli **estremi delle classi di equivalenza o del dominio degli input**. Questo metodo è efficace perché spesso i difetti si manifestano negli intorni dei valori limite.

EX.

Prendendo l'esempio di prima delle tasse:

Si può qui definire la *test obligation*:

- Provare tutti gli intorni degli estremi degli intervalli.

Metodo dei casi non validi

Per ogni input si definiscono *valori che devono generare errori* (es. input al di fuori dei limiti validi). Questo metodo **completa, ma non sostituisce, quelli precedenti**.

EX.

- Età inferiori a 20 o superiori a 120 per la laurea.
- Reddito negativo per il calcolo delle aliquote.
- Spesa negativa per il calcolo dello sconto.

Metodo random

Genera automaticamente un numero arbitrario di valori di input. Sebbene rapido e automatizzabile, presenta limiti come la **mancanza di ripetibilità, difficoltà nel determinare gli output attesi** e una **scarsa attenzione ai valori limite**.

Metodo del catalogo

Basato sull'esperienza, utilizza un **catalogo di test case già noto** per definire quelli più significativi per un determinato sistema. Questo approccio è meno formale e dipende dalla conoscenza pregressa di sistemi simili.

Testing combinatorio

Quando un sistema software presenta più dati di input, il numero di combinazioni di test possibili può diventare rapidamente ingestibile, specialmente considerando il prodotto cartesiano tra i vari casi. Per affrontare questo problema, è necessario adottare strategie che consentano di selezionare combinazioni significative di input, riducendo al contempo il numero totale di test case.

Due tecniche comuni per il testing combinatorio sono i *vincoli* e il *pairwise testing*.

Vincoli

L'idea alla base dei *vincoli* è quella di **ridurre il numero di test case concentrandosi su combinazioni specifiche**, evitando di testare tutte le possibili varianti.

EX.

Immaginiamo di avere cinque parametri di input (x_1, x_2, x_3, x_4, x_5) con domini così definiti:

- x_1 e x_2 : 8 classi di valori, una delle quali rappresenta input non validi.
- x_3 e x_5 : 4 classi di valori, una delle quali rappresenta input non validi.
- x_4 : 7 classi di valori, una delle quali rappresenta input non validi.

Se prendessimo una combinazione per ciascuna classe, otterremmo

$$8 \times 8 \times 4 \times 7 \times 4 = 7168 \text{ test case}$$

Per ridurre questa complessità, si possono applicare tre tipi di vincoli:

1. *Vincoli di errore*

Si seleziona un solo caso con input non valido per ogni posizione, riducendo il numero di test case a:

$$5 + 7 * 7 * 3 * 6 * 3 = 2651$$

2. *Vincoli di proprietà*

Si definiscono regole specifiche:

Definiamo dei vincoli **property/if-property** sui primi due vincoli:

- x_1 :
 - classe 1, classe 2, classe 3, classe 4 [property negativi]
 - classe 5, classe 6, classe 7 [property positivi]
 - (classe 8 [property error])
- x_2 :
 - classe 1, classe 3, classe 5, classe 7 [if negativi]
 - classe 2, classe 4, classe 6 [if positivi]
 - (classe 8 [property if error])

I vincoli di proprietà permettono di classificare i valori in gruppi significativi (es. valori “positivi”, “negativi”, “non validi”) riducendo il numero di test case a:

$$5 + (4 * 4 + 3 * 3) * 3 * 6 * 3 = 1355$$

Quindi il prodotto originale 7×7 viene ridotto a: **$4 \times 4 + 3 \times 3$**

3. *Vincoli single*

Si limita il test a un solo valore per uno o più parametri. Ad esempio, applicando il vincolo single al parametro x_4 , il numero di test case si riduce a:

$$5 + (4 * 4 + 3 * 3) * 3 * 1 * 3 = 230$$

Pairwise Testing

Il **pairwise testing** è una tecnica utilizzata quando non esistono vincoli evidenti tra i dati di input. Invece di generare tutte le possibili combinazioni, questa tecnica si concentra sulle combinazioni tra tutte le possibili coppie di variabili.

Riprendendo l'esempio di prima in cui abbiamo 5 parametri, le possibili combinazioni sono:

$$8^*8^*4^*7^*4 = 7.168$$

Applicando il pairwise testing otteniamo il seguente numero di combinazioni, cioè di test case:

$$8^*8 + 8^*4 + 8^*7 + 8^*4 + 8^*4 + 8^*7 + 8^*4 + 4^*7 + 4^*4 + 7^*4$$

che sono **376**, invece di **1768**.

EX.

Supponiamo di avere 3 parametri con i seguenti valori:

- Parametro A: {1, 2}
- Parametro B: {X, Y}
- Parametro C: {α, β}

Tutte le combinazioni possibili sarebbero:

1. (1, X, α)
2. (1, X, β)
3. (1, Y, α)
4. (1, Y, β)
5. (2, X, α)
6. (2, X, β)
7. (2, Y, α)
8. (2, Y, β)

Con il *Pairwise Testing*, ci focalizziamo solo sulle **coppie di valori**, riducendo il numero di test case. Ad esempio:

- Coppie A-B: (1, X), (1, Y), (2, X), (2, Y)
- Coppie A-C: (1, α), (1, β), (2, α), (2, β)
- Coppie B-C: (X, α), (X, β), (Y, α), (Y, β)

Ora costruiamo una suite di test che copre tutte queste coppie almeno una volta attraverso un algoritmo:

1. (1, X, α)
2. (1, Y, β)
3. (2, X, β)

4. (2, Y, α)

Il numero è stato ridotto da 8 combinazioni a 4 combinazioni significative.

Capitolo 20

Criteri white-box

I criteri *white-box* si basano sull'**analisi della struttura interna del codice** per identificare i casi di test da aggiungere.

Questi criteri non sostituiscono i test black-box, ma li integrano, aiutando a identificare malfunzionamenti che non emergono con i test basati esclusivamente sui requisiti funzionali.

L'obiettivo principale dei criteri white-box è rispondere alla domanda: *Quali parti del codice non sono ancora state esercitate dai test e necessitano di ulteriori verifiche?*

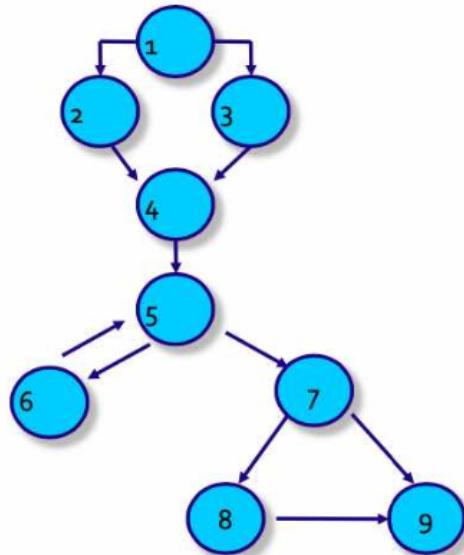
Un programma non può essere considerato adeguatamente testato se alcune sue componenti non sono mai state eseguite. Tali elementi possono essere comandi, decisioni, condizioni, cammini, etc.

Grafo di flusso

Un grafo di flusso definisce la struttura del codice identificandone le parti e come sono collegate tra loro.

EX.

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



Copertura dei comandi

E importante definire un insieme di possibili input che esercitino tutti i comandi del programma.

EX. In riferimento all'algoritmo eleva dell'esempio precedente

Le coppie:

- (x = -2, y = 3)
- (x = 4, y = 0)
- (x = 0, y = -5)

Esercitano tutti i comandi.

La *test suite* è generalmente *tanto più adeguata quanti più comandi esercita*.

Copertura delle decisioni

Bisogna avere casi di test che esercitino entrambi i rami di ogni condizione.

Copertura dei cammini

La copertura dei cammini richiede di percorrere tutti i cammini.

In caso di cicli, è bene definire test case che esercitino il ciclo:

- 0 volte;
- 1 volta;
- 2+ volte.

Fault based testing

Il *fault based testing* si basa sull'idea di **ipotizzare difetti potenziali nel codice e creare test suite per verificarne la capacità di rilevarli**.

Una tecnica comune è il *testing mutazionale*, che consiste nell'iniettare difetti artificiali nel codice (mutanti) per valutarne la capacità di essere individuati dai test.

EX.

```
int foo(int x, int y){
    if(x ≤ y)
        return x+y
    else
        return x*y
}

int foo(int x, int y){
    if(x<y) //Mutante ≤ → <
        return x+y
    else
        return x*y
}
```

Se la test suite non riesce a individuare i difetti nel codice originale o nella versione modificata (mutante), **si dice che il mutante non viene "ucciso"**. Questo indica che la **test suite è inefficace** e potrebbe richiedere miglioramenti o test più sofisticati.

Il *testing mutazionale* funziona come segue:

1. Si esegue il programma originale (P) con una test suite (T), verificando che P funzioni correttamente.
2. Si introducono dei difetti (mutazioni) nel codice, creando una versione modificata del programma (P').
3. Si esegue il programma mutato (P') sulla stessa test suite (T). Se la test suite rileva i difetti, significa che è efficace nel rilevare errori. Se non rileva i difetti,

significa che la test suite non è sufficientemente buona.

Il **testing mutazionale** si applica spesso in combinazione con altri criteri di test. I mutanti possono essere:

- **Invalido**: Se non è sintatticamente corretto (e.g., non passa la compilazione).
- **Valido**: Se è sintatticamente corretto.
- **Utile**: Se è difficile distinguere il mutante dal programma originale, cioè solo una piccola parte della test suite è in grado di rivelarlo, altrimenti è **Inutile**.
- **Equivalente**: Se il comportamento semantico del mutante è identico a quello del programma originale.

L'obiettivo è valutare l'efficacia della test suite nell'individuare mutanti e, quindi, migliorare la qualità dei test.

EX.

\original

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

\invalido

```
int foo(int x, int y)
{ if(x < "a")
    return x+y;
else return x*y; }
```

\inutile

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

\equivalente

```
int foo(int x, int y)
{ if(x < y)
    return x+y+1-1;
else return x*y; }
```

Oracolo

L'**oracolo** nel contesto del testing è lo **strumento che fornisce l'output atteso per un determinato input**, permettendo così di confrontare i risultati del test con quelli previsti.

Se il calcolo dell'output atteso richiede molta fatica manuale, il processo di testing perde di efficacia. Per questo motivo, l'oracolo è essenziale, e può essere implementato in vari modi:

1. *Risultati ricavati dalle specifiche*: Le specifiche stesse del sistema indicano quale dovrebbe essere l'output per un dato input, quindi l'oracolo si basa direttamente su di esse.
2. *Inversione delle funzioni*: Se è possibile calcolare facilmente la funzione inversa di quella applicata dal programma, l'oracolo può confrontare il risultato ottenuto con quello che ci si aspetta.
3. *Versioni precedenti dello stesso programma*: Può comparare l'output della versione corrente del programma con quello di una versione precedente correttamente funzionante.
4. *Versioni multiple indipendenti*: Si può confrontare l'output del programma con quello di altre implementazioni alternative.
5. *Semplificazione dei dati di ingresso*: Può anche funzionare su input semplificati per verificare la correttezza del programma in scenari più semplici.
6. *Semplificazione dell'output*: In alcuni casi, non è necessario avere un risultato esatto, ma solo una versione approssimativa che consenta di verificare che l'output sia plausibile.

