

Appunti Analisi e Progettazione del Software

Domande e Risposte

Capitolo 1 - Introduzione Analisi e Progettazione del Software

1) Cosa consistono le fasi di analisi e progettazione?

La fase di **analisi** ha l'obiettivo di determinare e di descrivere in maniera dettagliata tutte le componenti necessarie per lo svolgimento del progetto. Tale fase risulta essere di maggiore importanza dato che il risultato di questo lavoro diventa la base e la guida di tutto il progetto. La fase di **progettazione** ha l'obiettivo di determinare una soluzione concettuale che soddisfa i requisiti escludendo spesso i dettagli di basso livello o di seconda importanza.

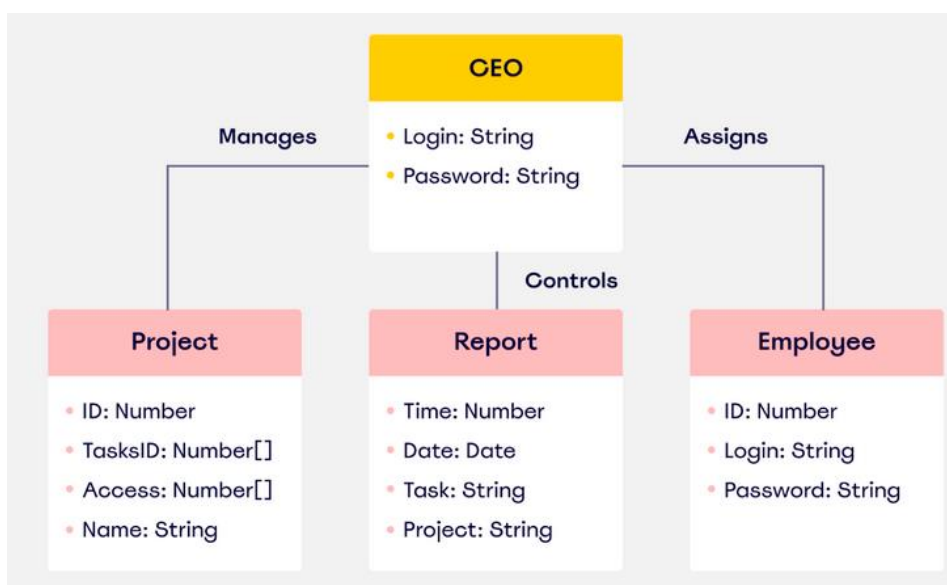
2) Cosa si intende per progettazione orientata agli oggetti e analisi orientata agli oggetti?

Si definisce **progettazione orientata agli oggetti (OOD)** un approccio alla progettazione dei **sistemi software** fondata sul **paradigma ad oggetti**. Si definisce invece **analisi orientata agli oggetti (OOA)** un approccio di analisi che consiste nel definire le **classi concettuali** (di **dominio**). Tali **classi** vengono utilizzate per descrivere i concetti o gli oggetti (del dominio di interesse/mondo reale) relativi al problema.

3) Cosa si intende per UML (Unified Modeling Language)?

UML (Unified Modeling Language) è un linguaggio visuale per la costruzione, la documentazione e la specifica degli elaborati di un sistema software.

Rappresenta uno **standard de facto** per la notazione dei diagrammi.



4) Quali sono i tre modi per applicare UML?

Ci sono tre modalità per applicare UML:

- come **linguaggio di programmazione**: rappresenta in modo pratico il comportamento e la logica;
- come **abbozzo**: visti come diagrammi informali e incompleti utili per l'espressività dei linguaggi visuali;
- come **progetto**: visti come diagrammi di progetto relativamente dettagliati e vengono utilizzati per il "reverse engineering", ovvero per visualizzare e comprendere il codice già esistente.

La **modellazione Agile** enfatizza il secondo utilizzo.

5) Quali sono i due punti di vista per applicare UML?

Il **diagramma UML** può essere utilizzato per due diversi punti di vista:

- **punto di vista concettuale**: i diagrammi sono scritti e interpretati come descrizioni di oggetti del mondo reale o nel dominio di interesse;
- **punto di vista software**: i diagrammi descrivono astrazioni o componenti software.

6) Come vengono rappresentate le classi in UML?

In UML, le **classi** sono rappresentate da rettangoli e racchiudono una varietà di significati. In un **modello di dominio**, i rettangoli sono definiti **concetti di dominio** o **classi concettuali** e rappresentano un concetto del mondo reale, mentre nel **modello di progetto** i rettangoli costituiscono le **classi di progetto** o **classi software** rappresentano un componente software.

7) Quali sono i vantaggi della modellazione visuale?

La **modellazione visuale** contribuisce ad un'esaminazione più specifica delle relazioni tra gli elementi nell'analisi del **software** astruendo da dettagli poco significativi.

Capitolo 2 - Processi iterativi ed evolutivi

8) Che cosa si intende per UP (Unified Process)?

Si dice **UP (Unified Process)** un processo iterativo diffuso per lo sviluppo del software per la costruzione di sistemi orientati agli oggetti.

9) Quali sono le caratteristiche di UP?

Le caratteristiche di **UP** sono l'ampia flessibilità, apertura del sistema e combina delle best practice comunemente accettate. **UP** risulta, inoltre, ad essere un'ottima scelta per tre motivi:

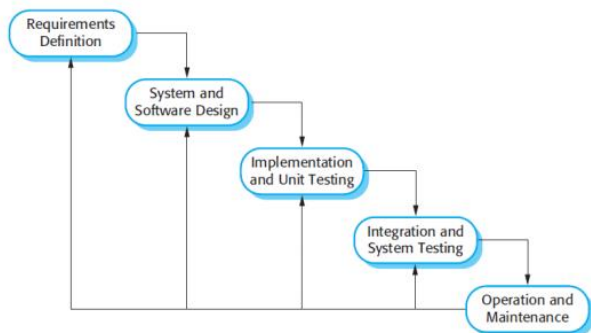
- a - lo **sviluppo iterativo** è il modo migliore per applicare l'OOA/D;
- b - fornisce una struttura di esempio sia per eseguire che per spiegare l'OOA/D;
- c - può essere applicato usando un approccio leggero e agile che comprende pratiche di altri metodi agili.

10) Quali sono le caratteristiche dei processi per lo sviluppo software?

Un processo per lo **sviluppo software** definisce un approccio disciplinato per la costruzione, il rilascio e la manutenzione del **software**. Descrive generalmente chi fa cosa, quando e come per raggiungere un determinato scopo.

11) Descrivere le caratteristiche del processo a cascata?

Il **processo software a cascata** è basato su uno svolgimento sequenziale delle diverse attività dello sviluppo software. Risulta essere spesso mediocre e poco efficace ed è potenzialmente associato ad una frequente percentuale di fallimenti e ad un basso indice di produttività.



12) Cosa si intende per sviluppo iterativo? Quali sono i suoi vantaggi?

Lo **sviluppo iterativo** è una pratica moderna organizzata in una serie di mini - progetti brevi di lunghezza fissa, chiamati **iterazioni**. Il risultato di ogni **iterazione** è un sistema eseguibile, testato e integrato che fornisce un feedback alle fasi successive. I vantaggi dello **sviluppo iterativo** sono:

- minore probabilità di fallimento del progetto;
- riduzione notevole dei rischi maggiori;
- il progresso viene già reso visibile alle prime fasi;
- coinvolgimento dell'utente e adattamento basato sui feedback, che conducono ad un sistema che soddisfa meglio le esigenze reali;
- gestione della complessità, evitando la "paralisi da analisi";
- l'apprendimento nel corso di un'iterazione può indurre a miglioramenti o progressi per il processo di sviluppo stesso;
- riduzione notevole dei termini di consegna e il **timeboxing** è l'idea chiave di

tale metodologia in cui ogni iterazione ha un quanto di tempo fissato e non consente ritardi di date delle iterazioni.

13) Cosa sono iterazioni timeboxed?

Un'iterazione si dice **timeboxed** se tale iterazione ha una durata fissa.

14) Quali sono le fasi di UP?

Un **progetto UP** organizza il lavoro e le iterazioni in quattro fasi temporali principali:

- 1 - **ideazione**: visione approssimativa, con stime dei costi e dei tempi;
- 2 - **elaborazione**: visione raffinata, implementazione iterativa del nucleo dell'architettura, identificazione della maggior parte dei requisiti;
- 3 - **costruzione**: implementazione iterativa degli elementi rimanenti, più facili e a rischio minore
- 4 - **transizione**: beta test e rilascio.

15) Quali sono le altre pratiche fondamentali di UP?

Ci sono ulteriori best practice e concetti chiave di **UP**:

- affrontare le problematiche di rischio maggiore e valore elevato nelle iterazioni iniziali;
- impegnare gli utenti continuamente sulla generazione dei feedback;
- creare un'architettura coesa;
- verificare continuamente la qualità e testare spesso e in modo realistico.

16) Cosa si intende per disciplina di UP?

Una **disciplina** è un insieme di attività e dei relativi elaborati in una determinata area. In **UP**, un **elaborato** è il termine generico che indica un qualsiasi prodotto di lavoro. La **disciplina infrastruttura** fa riferimento alla definizione degli strumenti, ovvero all'impostazione degli strumenti e del processo.

Capitolo 3 - Agile

17) Che cosa si intende per modellazione agile?

La **modellazione agile** è una modalità utilizzata per comprendere al meglio un problema e spaziare tra le soluzioni, esplorando le alternative. Tale concetto si basa su un insieme di pratiche e valori:

- considera la modellazione come la meno importante;
- lo scopo dei modelli è di agevolare la comprensione e la comunicazione;
- vengono utilizzati modelli solo per risolvere problematiche più difficili e insidiose rimandando i problemi semplici o diretti alla fase di programmazione;

- deve essere utilizzato per primo lo strumento più semplice possibile, che aiuti la creatività con il minimo dispendio, indipendentemente dalla tecnologia scelta;
- la modellazione non va fatta in solitario, ma piuttosto a coppie;
- non è importante essere rigidi e rigorosi nell'utilizzo dell'UML, purché i modellatori si capiscano l'uno con l'altro;
- tutti i modelli sono inevitabilmente imprecisi, e il codice o il progetto finale differiranno, a volte anche in modo notevole;
- la modellazione OO dovrebbe essere eseguita dagli stessi sviluppatori che si occupano della programmazione.

18) Che cosa si intende per UP agile?

L'**UP agile** è una versione semplificata dell'IBM **Rational Unified Process (RUP)** che descrive un approccio allo sviluppo di applicazioni software, semplice da comprendere e che utilizza tecniche e concetti agili attenendosi comunque ai principi di **RUP**.

19) Che cosa si intende per SCRUM?

Si definisce **SCRUM** un metodo agile che consente di sviluppare e rilasciare prodotti software con il più alto valore per i clienti, nel più breve tempo possibile. Si occupa principalmente dell'organizzazione del lavoro e della gestione dei progetti lasciando la possibilità agli sviluppatori di scegliere le tecniche e le metodologie specifiche da utilizzare.

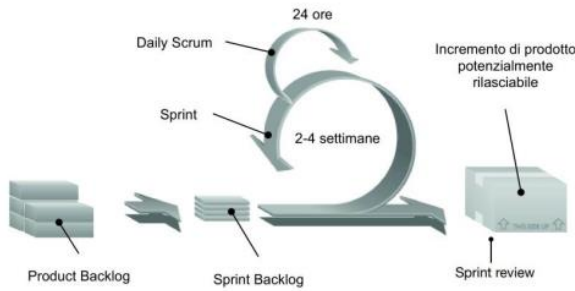
20) Caratteristica principale di SCRUM

SCRUM è un approccio iterativo e incrementale allo sviluppo del **software**. Ciascuna **iterazione**, chiamata uno **Sprint**, ha una durata fissata, per esempio di due settimane (iterazioni timeboxed e non vengono mai estese).

21) Quali sono i ruoli di SCRUM?

Ci sono vari ruoli:

- **ScrumMaster**: non è necessariamente il project manager; si prefigge di far applicare al meglio il metodo **SCRUM** e fa da tramite tra il team e altri team aziendali;
- **Team di sviluppo**: gruppo di sviluppatori, non più di 7; sono responsabili del software e dei vari elaborati;
- **Product owner**: il cliente (eventualmente è il product manager o rappresentante di altri stakeholder) oppure un piccolo gruppo di persone.



Capitolo 4 - Ideazione (Prima fase)

22) Cosa si intende per ideazione? Quali sono i suoi componenti?

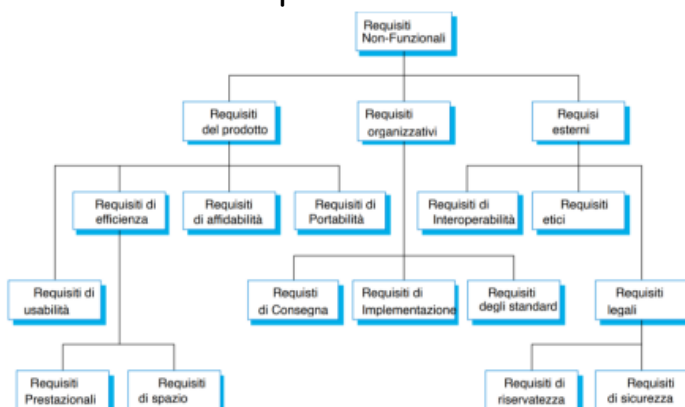
In UP l'**ideazione** è il passo iniziale che permette di definire la visione del progetto. Ha lo scopo di raccogliere informazioni per avere una visione comune e decidere se il progetto è fattibile. Essa è composta da un'unica iterazione, detta **iterazione zero**, differente dalle altre perché non punta a creare software eseguibile, ma si concentra sull'avvio di attività ed elaborati. Oltre a fare una stima approssimativa dei costi e dei tempi di sviluppo comprende anche l'analisi del circa 10% dei **requisiti funzionali** e **non funzionali più critici**.

23) Cosa si intende per requisito e quali sono i tipi di requisiti?

Si dice **requisito** una capacità o una condizione a cui un sistema, e quindi il progetto, deve essere conforme e devono essere completi rispetto alle richieste del cliente, non ambigui e coerenti fra loro. Ci sono due tipi di requisiti:

→ **Requisiti funzionali**: definiscono **funzionalità** o **servizi** che il sistema deve fornire, risposte che l'utente si aspetta in determinate condizioni e output attesi a fronte di determinati input. Gli elaborati prodotti in merito sono i **Casi d'Uso**.

→ **Requisiti non funzionali**: sono requisiti relativi alle proprietà del sistema. Possono riguardare vincoli del sistema a livello di affidabilità e rapidità o vincoli su I/O, oppure vincolano gli standard di qualità del progetto, o l'IDE da usare, ecc. Gli elaborati prodotti in merito sono le **Specifiche Supplementari**.



24) Tipologie di requisiti non funzionali: quali sono?

I **requisiti non funzionali** del sistema si dividono in due categorie:

- **Obiettivi: intenzioni generali** dell'utente (esempio: facilità d'uso), difficilmente quantificabili; rappresentano un'ideale da raggiungere, **non esprimibile oggettivamente**.
- **Requisito non funzionale verificabile: dichiarazione** che utilizza misure oggettivamente quantificabili e verificabili.

25) Come viene effettuata l'acquisizione e la redazione dei requisiti?

UP incoraggia un'acquisizione dei requisiti agile, tramite:

- scrittura di Casi d'Uso con i clienti (anche tramite interviste);
- workshop dei requisiti con sviluppatori e clienti, o rappresentanti degli stessi;
- feedback gathering dai clienti dopo ogni interazione.

26) Come devono essere gli elaborati dei requisiti?

Gli **elaborati dei requisiti** devono essere scritti tramite **frasi in linguaggio naturale (formato testuale)** comprensibili, integrati eventualmente con diagrammi, sia per gli sviluppatori che per i clienti. Va ideato un **formato standard** per la struttura testuale del requisito (per esempio, viene indicato il tipo di requisito? Comincia con un verbo? Ha un titolo?). Si utilizza il verbo **deve** per i **requisiti obbligatori**, **dovrebbe** per quelli **desiderabili** (risulta utile usufruire di enfasi testuale).

27) Casi d'Uso: definizione

I **Casi d'Uso** sono storie scritte, solitamente con lunghezza ridotta, che raffigurano **dialoghi** tra uno o più **attori** e un **sistema** che svolge un **compito**. Vengono utilizzati per scoprire e registrare i **requisiti funzionali** (eventualmente anche alcuni **requisiti non funzionali**). Un Caso d'Uso definisce un contratto relativo al comportamento del sistema.

28) Quali sono gli elaborati dei Casi d'Uso definiti da UP?

UP definisce come elaborato il **Modello dei Casi d'Uso**, ossia l'insieme di tutti i **Casi d'Uso** (detto testo dei **Casi d'Uso**) descritti ed eventualmente un **diagramma UML** degli stessi. Comprende anche, se presenti, i **Diagrammi di Sequenza di Sistema** e i **Contratti**.

29) Casi d'Uso: perché vengono utilizzati?

Vengono utilizzati perché:

- aiutano a scoprire e descrivere i **requisiti funzionali**;
- sono direttamente **comprensibili per i clienti**, permettendo di **coinvolgerli** nella definizione e revisione;
- mettono in risalto gli **obiettivi degli utenti** e il loro punto di vista;
- sono utili per produrre **test** e la **guida utente**.

30) Casi d'Uso: quali sono i componenti?

Un **Caso d'Uso** è costituito da:

- **Attore**: qualcosa o qualcuno **dotato di un comportamento**.
- **Scenario**: **sequenza specifica di azioni e interazioni** tra il sistema e alcuni attori che è costituito da una sequenza di passi, che possono essere di tre tipi:
 - a) un'interazione tra attori;
 - b) un cambiamento di stato da parte del sistema;
 - c) una validazione.

→ **Caso d'Uso**: è una **collezione di scenari** correlati, sia di successo che di fallimento.

All'interno di un Caso d'Uso possiamo distinguere diversi attori:

- **Attore primario**: utilizza direttamente i servizi del **SuD (System under Discussion)**, affinché vengano raggiunti degli obiettivi utente;
- **Attore finale**: vuole che il SuD venga utilizzato affinché vengano raggiunti dei suoi obiettivi;
- **Attore di supporto**: offre un servizio al SuD;
- **Attore fuori scena**: ha un interesse nel comportamento del Caso d'Uso, ma non è nessuno dei 3 tipi precedenti, né interviene all'interno del Caso d'Uso.

31) Quali sono le tipologie di Casi d'Uso?

Si possono scrivere i Casi d'Uso (scritti) a diversi livelli di dettaglio e formalità:

- **Formato breve**: riepilogo conciso di un solo paragrafo, relativo al solo scenario principale di successo;
- **Formato informale**: più paragrafi, relativi a vari scenari;
- **Formato dettagliato**: tutti i passi e tutte le variazioni vengono scritti nel dettaglio, include anche ulteriori sezioni. Un Caso d'Uso dettagliato dovrebbe essere lungo tra le 3 e le 10 pagine.

Una seconda distinzione dei **Casi d'Uso** permette di identificare la cosiddetta "scala" dei **Casi d'Uso**:

- **Livello di obiettivo utente**: consente all'utente di raggiungere un proprio obiettivo;

→ **Livello di sotto-funzione**: rappresenta solo una funzionalità nell'uso del sistema, sono interazioni di dettaglio;

→ **Livelli di sommario**: riguarda un obiettivo più ampio, non direttamente raggiungibile con un singolo utilizzo del sistema. Sono utili per comprendere il contesto di più **Casi d'Uso**.

Attenzione: nei Casi d'Uso viene specificato ciò che il sistema deve fare e non come lo deve fare.

Template Caso d'Uso

Nome del Caso d'Uso	Indicato con un verbo
Portata	Nome del SuD (confini del sistema).
Livello	Obiettivo utente/sotto-funzione/sommario.
Attore primario	Chi usa direttamente il sistema.
Parti interessate e interessi	A chi interessa questo Caso d'Uso e perché (attori coinvolti).
Pre-condizioni	Cosa dev'essere vero all'inizio del Caso d'Uso.
Garanzia di successo	Cosa dev'essere vero se il Caso d'Uso viene completato con successo.
Scenario principale di successo	Uno scenario comune di attraversamento del Caso d'Uso, di successo e incondizionato. Tale scenario soddisfa gli interessi delle parti interessate.
Estensioni	Scenari alternativi, di successo o fallimento.
Requisiti speciali	Requisiti non funzionali verificabili correlati.
Elenco delle varianti tecnologiche e dei dati	Varianti nei metodi di I/O e nel formato dei dati.
Frequenza di ripetizione	Frequenza prevista di esecuzione del Caso d'Uso.
Varie	Altri aspetti (problemi aperti etc.)

Per scrivere un **Caso d'Uso** efficace è necessario quindi:

→ scegliere i confini di sistema (anche attraverso attori esterni);

→ identificare gli attori primari;

→ identificare gli obiettivi di ciascun attore primario;

→ definire i Casi d'Uso che soddisfano gli obiettivi degli utenti; il loro nome va scelto in base all'obiettivo.

32) Come posso verificare l'utilità di un Caso d'Uso?

Per verificare l'utilità dei Casi d'Uso ci sono 3 test:

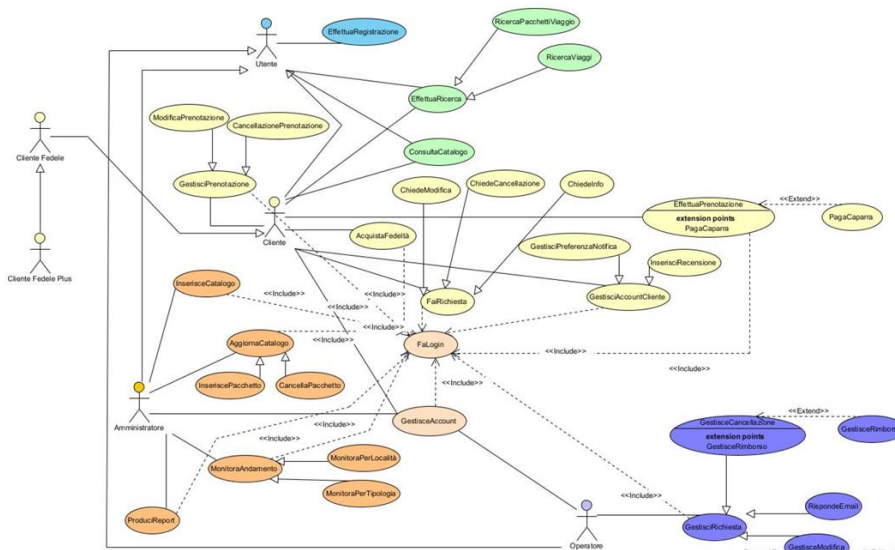
- **test del capo**: Se un Caso d'Uso non supera il test del capo, ma è un Caso d'Uso complicato, vale comunque la pena tenerlo;
- **test EBP (Elementary Business Process)**: si concentra sul valore aziendale misurabile, nel senso che si tratta di un'attività eseguibile in una sola sessione di lavoro, in risposta a un evento business che genera valore business misurabile e osservabile e che lasci il sistema in uno stato stabile e coerente;
- **test della dimensione**: deve avere tra le 3 e le 10 pagine di testo.

33) Cosa sono i diagrammi dei casi d'uso?

Un **diagramma dei Casi d'Uso** costituisce un buon diagramma di contesto, che serve a mostrare il quadro generale e i confini del sistema, ciò che giace al suo esterno e come viene utilizzato.

Le relazioni che i **Casi d'Uso** possono avere sono tre:

- **Include**: relazione tra un **Caso d'Uso** e un **Caso d'Uso** incluso nel caso base. Indica un **Caso d'Uso** che viene sempre e completamente eseguito all'interno di un altro.
- **Extend**: **Caso d'Uso** che estende un **Caso d'Uso** base. Indica un **Caso d'Uso** che, se viene soddisfatta una certa condizione, viene eseguito per poi tornare al punto di estensione.
- **Generalizzazione**: funziona praticamente come in **Java** per i **Casi d'Uso** (ereditano tutto, possono aggiungere di tutto, e possono sovrascrivere tutto tranne relazioni e punti di estensione), mentre per gli attori ereditano ruoli e relazioni dell'antenato che per buona pratica dovrebbe essere astratto.



Capitolo 5 - Modellazione del Business

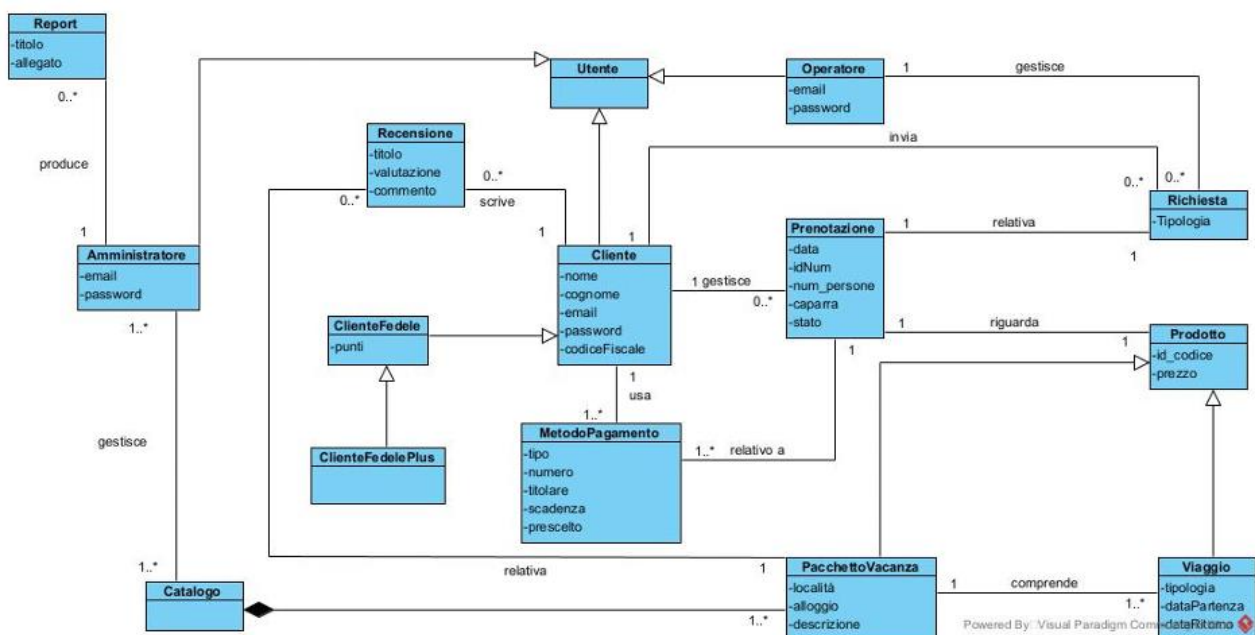
34) Cosa si intende per analisi ad oggetti?

L'**analisi orientata agli oggetti** è basata sull'identificazione dei concetti nel dominio del problema che modella 3 aspetti di un sistema:

- le informazioni da gestire (**Modello di dominio**);
- le funzioni (**Diagrammi di Sequenza di Sistema**);
- il comportamento, ossia i cambiamenti di stato del sistema conseguenze delle funzioni (**Contratti**).

35) Modello di Dominio: cosa si intende?

Il **Modello di Dominio** è il documento principale dell'analisi a oggetti. È una rappresentazione visuale (diagramma) delle classi concettuali (non software), che mostrano i concetti del mondo reale nonché le loro relazioni tra di essi nel dominio di interesse. Esso è costituito da classi concettuali, i loro attributi e le relazioni che intercorrono tra i modelli di dominio.



36) Modello di Dominio: perché viene utilizzato?

Si usa il **Modello di Dominio** per:

- 1) comprendere il dominio del sistema da realizzare e per definire un linguaggio comune sulle parti interessate al sistema;
- 2) la progettazione dello strato del dominio.

37) Modello di Dominio: classe concettuale

Una **classe concettuale** è un'idea, un oggetto del mondo reale; formalmente è definita da:

- **un simbolo**: parola o immagine che rappresenta la classe;

→ **un'intensione**: definizione in linguaggio naturale della classe e delle sue proprietà;

→ **un'estensione**: l'insieme degli oggetti descritti dalla classe.

I **nomi** delle classi concettuali sono di solito al singolare; sono nomi esistenti nell'area di interesse, non sono fuori dalla portata dell'iterazione e non sono sovrabbondanti.

38) Modello di Dominio: attributi

Un **attributo** è una proprietà elementare degli oggetti di una classe; ogni oggetto assegna un valore ai propri attributi. La sintassi nelle **classi UML** è: visibility name: type multiplicity = default {property-string}. Gli attributi non devono correlare classi concettuali. (non vanno usati come attributi di chiave esterna).

Le **visibilità** si indicano in questo modo: + public, ~ package, # protected, - private.

39) Modello di Dominio: classe descrizione

Una classe descrizione è una speciale classe concettuale che contiene informazioni su qualcos'altro. Conviene introdurre una classe descrizione quando:

→ è necessaria una descrizione di un articolo o servizio, indipendentemente dall'attuale esistenza di istanze di tali articoli/servizi;

→ l'eliminazione delle istanze degli oggetti che sono descritti darebbe luogo a una perdita di informazioni;

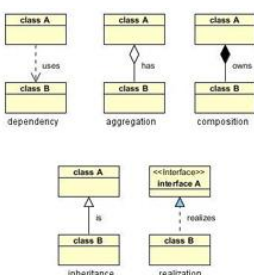
→ si vuole ridurre la ridondanza di informazioni.

(Nota: è identificato da **DataType**)

40) Modello di Dominio: associazioni tra classi concettuali

Un'**associazione** è una relazione semantica tra classi, che comporta connessioni tra le rispettive istanze. Le istanze di un'**associazione** sono dette **collegamenti**.

Le associazioni vengono mostrate se le stesse devono essere memorizzate nel sistema, indipendentemente dalla loro durata, e sono significative (non ridondanti o derivabili).



Ci sono associazioni "speciali" offerte da **UML**: sono l'aggregazione, la composizione, la generalizzazione (più la dipendenza e la realizzazione, che però non trattiamo a livello di dominio).

→ **Aggregazione**: tipo di **associazione/relazione intero-parte**. L'aggregato può esistere indipendentemente dalle parti e viceversa;

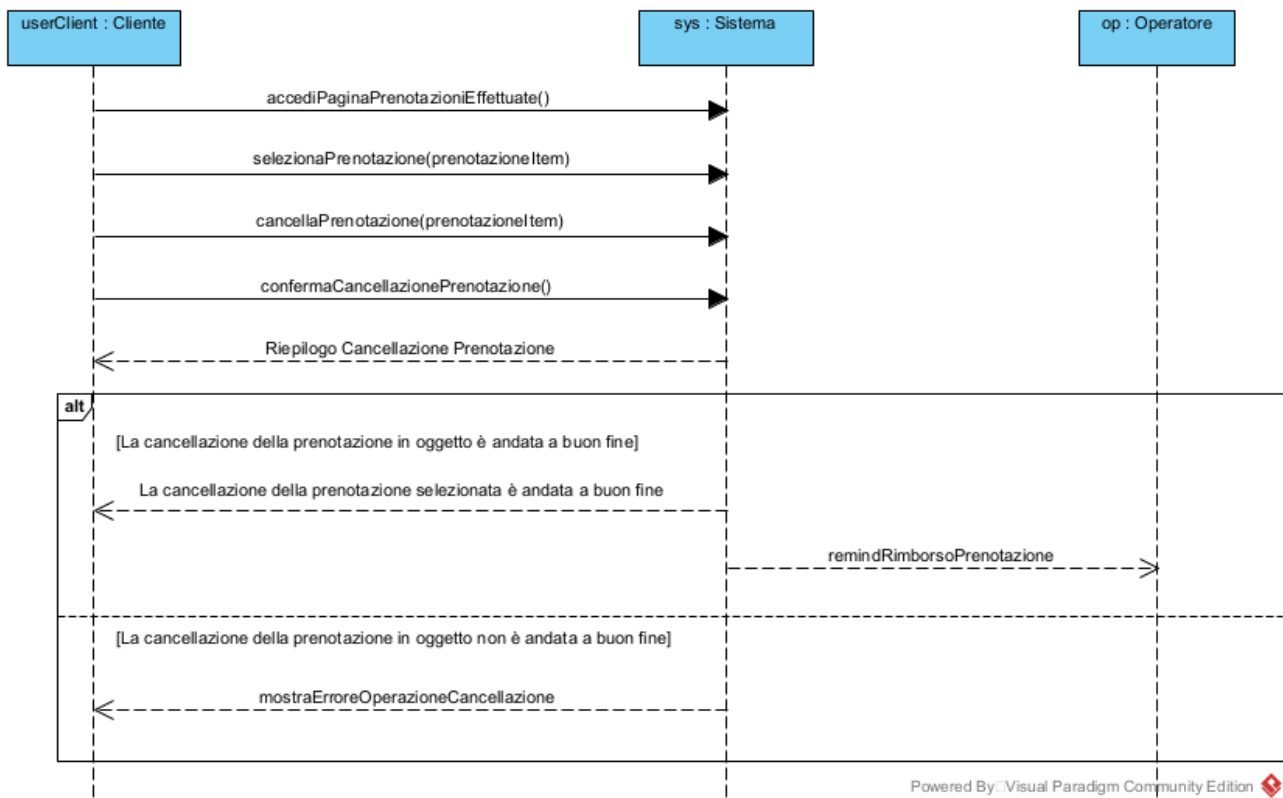
→ **Composizione**: la **composizione** è un tipo forte di aggregazione intero-parte e implica che:

- a) ogni istanza della parte deve appartenere sempre e solamente a una istanza del composto;
- b) il composto ha la responsabilità (operazioni CRUD) rispetto alle sue parti;
- c) il composto può rilasciare una delle parti se un altro composto se ne prende la responsabilità.

Capitolo 6 - Requisiti

41) Cosa si intende per SSD (State Sequence Diagram)?

Un **SSD (State Sequence Diagram)** è un elaborato che illustra eventi di input e output relativi ai **SuD** e descrive come gli attori (o più sistemi) interagiscono con il sistema. Il testo dei Casi d'Uso e gli eventi di sistema da essi sottintesi costituiscono l'input per la creazione degli **SSD**; quindi si adotta, coerentemente, il ragionamento a scatola nera per il sistema e gli attori esterni.



Come mostrato in figura gli eventi di sistema dovrebbero essere espressi a un livello astratto e cominciare con un verbo.

42) Cosa sono i contratti?

I **contratti** sono delle operazioni descrivono nel dettaglio i cambiamenti agli oggetti di dominio come risultato dell'esecuzione di un'operazione di sistema, ossia una funzionalità che il sistema (a scatola nera) offre tramite la sua interfaccia. I principali input per i contratti sono le operazioni di sistema identificate negli **SSD** e il **Modello di Dominio**.

Stesura di un Contratto

Contratto <nomeContratto>	
Operazione	Nome e parametri (signature) dell'operazione, con tipo.
Riferimenti ai Caso d'Uso	Casi d'Uso in cui può verificarsi.
Pre-condizioni	Condizioni necessarie affinché l'operazione si possa eseguire, ipotesi sullo stato del sistema. Espresse al presente o al passato.
Post-condizioni	Condizioni vere al termine dell'esecuzione dell'operazione, cambiamenti di stato degli oggetti nel Modello di Dominio. Espresse al passato e al passivo (è il sistema a cambiare gli oggetti a scatola nera, è sempre soggetto implicito).

Capitolo 7 - Dai Requisiti alla Progettazione

43) Cosa si intende per architettura software?

L'**architettura software** è un insieme delle decisioni significative sull'organizzazione di un sistema software, la scelta degli elementi strutturali, relative interfacce, il loro comportamento, la composizione di questi elementi strutturali e lo stile architettonico che guida questa organizzazione. È a larga scala.

44) Cosa si intende per architettura a N+1 viste?

Si dice **architettura a N+1 viste** un insieme delle decisioni significative sull'organizzazione di un sistema software in cui si descrivono N viste diverse più una fissa, la vista dei Casi d'Uso, che serve a descrivere l'intenzione

generale e lo scopo del progetto attraverso dei Casi d'Uso significativi a livello di sistema. È molto comune l'architettura a 6 viste:

- **Architettura logica**: si preoccupa di dividere le classi software in strati/package/sottosistemi in base alla loro semantica e appartenenza a un determinato strato/package/sottosistema a livello software;
- **Architettura di processo**: classi UML e Diagrammi di Interazione che mostrano i processi di sistema;
- **Architettura di rilascio**: specifica come viene rilasciato fisicamente il sistema e su quali dispositivi risiedono i vari nodi del sistema;
- **Architettura dei dati**: dove e come vengono salvati in modo persistente i dati;
- **Architettura di implementazione**: è la definizione del modello di implementazione, sostanzialmente l'attuale codice eseguibile, inclusi gli elaborati consegnabili ai clienti;
- **Architettura dei Casi d'Uso**: sommario dei Casi d'Uso più significativi a livello architetturale, ossia quei Casi d'Uso che, attraverso la loro implementazione, illustrano ampiamente le funzionalità del sistema.

45) Cosa si intende per architettura logica?

L'**architettura logica** si occupa di dividere le classi software in **strati/packages/sottosistemi** in base alla loro appartenenza e semantica ad un determinato strato a livello software. Viene definita **logica** dato che non vengono prese decisioni su come questi elementi siano distribuiti su processi/computer.

46) Quali sono le tre tipologie dell'architettura logica?

L'**architettura logica** viene suddivisa in tre strati:

- **Livello (tier)**: solitamente indica un nodo fisico di elaborazione (pc, tablet, server, router, switch);
- **Strato (layer)**: una sezione verticale dell'architettura (struttura Sistema Operativo, Modello ISO/OSI per le reti di telecomunicazione o la struttura del middleware);
- **Partizione (partition)**: una divisione orizzontale di sottosistema di uno strato.

47) Definizione e caratteristiche dell'architettura logica a strati

In un'architettura logica a strati, uno strato è un set di sottosistemi che ha delle responsabilità coese rispetto ad un aspetto importante del sistema. Gli

strati inferiori sono servizi generali e di basso livello, facilmente riusabili, quelli superiori sono orientati a servizi specifici per l'applicazione. Gli strati più alti ricorrono ai servizi degli strati più bassi.

48) Tipologie di architetture logiche a strati

Ci sono due tipi di **architetture a strati**:

→ **A strati stretta**: uno strato può richiamare solamente i servizi di uno strato immediatamente sottostante;

→ **A strati rilassata**: uno strato può richiamare servizi di strati più bassi di diversi livelli.

Solitamente un'**applicazione software** è costituita da tre macro - componenti:

→ **UI (User Interface)**: interfaccia software;

→ **Domain**: concetti che rappresentano il dominio e che costituiscono il cuore dell'applicazione;

→ **Servizi Tecnici** (che implementano ad esempio i protocolli di rete come HTTP, FTP, POP, IMAP, ecc.).

49) Perché vengono usati gli strati?

L'uso degli strati contribuisce ad affrontare ed evitare diversi problemi:

→ modifiche del codice che si estendono a tutto il sistema;

→ intrecci tra logica applicativa e UI (**separation of concerns**);

→ intrecci tra servizi / logica di business con logica applicativa (**separation of concerns**);

→ accoppiamenti tra diverse aree di interesse (**low coupling**).

50) Quali sono i vantaggi di tale architettura?

I vantaggi di tale **architettura** sono:

→ separations of concerns;

→ miglior coesione;

→ incapsulamento della complessità;

→ sostituibilità degli strati;

→ riusabilità delle funzioni degli strati bassi;

→ distribuzione di alcuni strati;

→ sviluppo degli strati da parte di team di sviluppo differenti.

51) Cos'è la logica applicativa?

La **logica applicativa** è lo strato che riguarda gli oggetti di dominio che viene suddivisa in due strati:

- **Strato del dominio**: gli oggetti di dominio;
- **Strato applicazione**: gli oggetti che gestiscono il workflow da e per gli oggetti di dominio, soprattutto dall'UI.

Capitolo 8 - Progettazione

52) Quali sono le possibili modalità di approccio alla progettazione?

Gli sviluppatori possono approcciarsi in diversi modi alla progettazione:

- **Codifica**: in cui viene effettuata la progettazione durante il processo codifica, con TTD e refactoring possibilmente.
- **Disegno, poi codifica**: il diagramma UML viene trasformato in codifica.
- **Solo disegno**: uno strumento che converte disegno in codice (in qualche modo).

53) Differenza tra modelli statici e dinamici

Ci sono due tipi di modelli per gli oggetti:

- **Modelli statici**: Diagrammi delle Classi, di package e di deployment (per progettare i package, attributi e firme dei metodi);
- **Modelli dinamici**: Diagrammi di Sequenza, di Comunicazione, delle Macchine a Stati e di Attività (per progettare la logica, il comportamento e corpo dei metodi). In tale modellazione vengono applicati i pattern e la progettazione guidata dalle responsabilità.

54) Scheda CRC (Class, Responsibility, Collaboration)

Una **scheda CRC** è un foglio che riporta le responsabilità e i collaboratori di una classe (una scheda corrisponde a una classe).

55) Diagramma di Interazione (definizione e tipologie)

Un **Diagramma di Interazione** è una rappresentazione grafica che rappresenta il modo in cui gli oggetti interagiscono attraverso lo scambio di messaggi.

Un'**interazione** è una specifica di come alcuni oggetti si scambiano messaggi nel tempo per eseguire un compito nell'ambito di un certo contesto. Tale interazione è motivata dalla **necessità** di eseguire un determinato compito, dato da un messaggio (detto **messaggio trovato**) che dà il via all'interazione tra un oggetto e altri partecipanti.

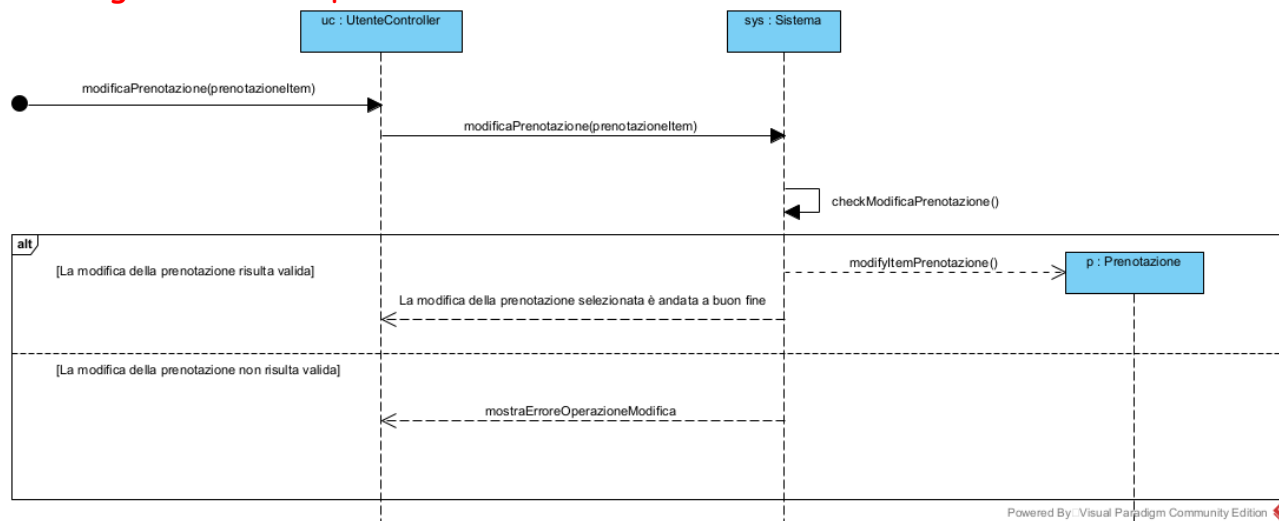
Ci sono 4 modalità di diagrammi di interazione:

- Diagrammi di Sequenza (SD);
- Diagrammi di Comunicazione (CD);

- Diagrammi di temporizzazione;
- Diagrammi di Interazione generale.

56) Diagramma di Sequenza (SD) : caratteristica principale

Un **diagramma di sequenza** mostra l'interazione tra le linee di vita verticale.

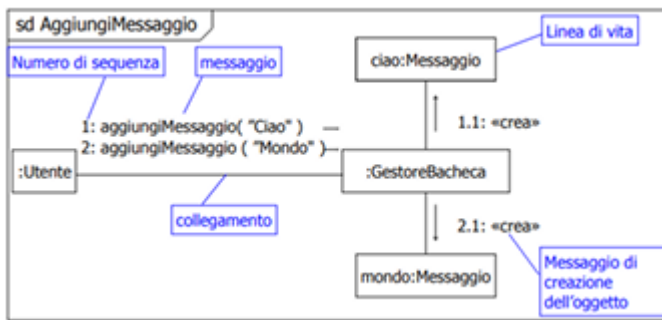


Operatore	Nome	Significato
opt	Option	C'è un singolo operando che viene eseguito se la condizione è vera (if...then)
alt	Alternatives	Viene eseguito l'operando la cui condizione è vera. Si può usare la parola chiave "altrimenti" o "else" per il caso di default (switch...case)
loop	Loop	Ha una sintassi speciale: loop min, max, [condizione]. Esegui loop min volte, poi mentre la condizione è vera esegui loop fino al massimo a max volte in totale.
break	Break	Se la condizione di guardia è vera viene eseguito l'operando ma non il resto dell'interazione
ref	Reference	Il frammento combinato fa riferimento ad un'altra interazione.

57) Diagramma di Comunicazione (CD)

Un **diagramma di comunicazione** è un diagramma di interazione alternativo al diagramma di sequenza in cui le interazioni tra gli oggetti vengono rappresentati in un formato a grafo o a rete. Le linee di vita sono formate nel momento in cui c'è un messaggio, che porta la firma del metodo e sono numerati (tranne il messaggio trovato iniziale se si vuole semplificare) per poterli leggere

nel giusto ordine.



58) Diagramma delle classi Software : definizione

I **Diagrammi delle Classi Software** illustrano le classi, le interfacce e le relative associazioni; sono usati per la modellazione statica delle classi e sono stati già introdotti per la realizzazione del Modello di Dominio.

59) Diagramma delle classi Software : oggetto

Un **oggetto** è un'istanza di una classe, che definisce l'insieme comune di caratteristiche (attributi e operazioni) condivisi da tutte le sue istanze. Tutti gli oggetti hanno:

- ID univoco;
- I valori degli attributi - la parte dei dati (detto stato);
- Le operazioni - la parte comportamento.

60) Diagramma delle classi Software : classificatore

Un **classificatore** è "un elemento di modello che descrive caratteristiche comportamentali e strutturali". I **classificatori** più diffusi sono **classi** e **interfacce**, ma anche i Casi d'Uso e attori sono **classificatori** (naturalmente non nel diagramma delle classi). Le **proprietà strutturali** di un **classificatore** comprendono i suoi **attributi** di classe e le **estremità delle associazioni**.

61) Diagramma delle classi Software : proprietà

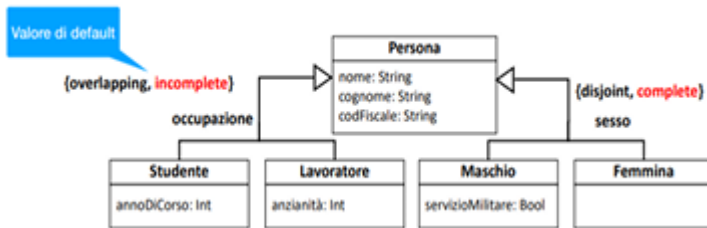
Una **proprietà** è un valore con un nome, che denota una caratteristica di un elemento. Le **proprietà** di una classe sono dette **attributi**, la visibilità è una proprietà delle operazioni.

62) Diagramma delle classi Software : parola chiave

Una **parola chiave** è un decoratore testuale per classificare un elemento di modello. Tra le parole chiave definite in UML troviamo «actor», «interface», {abstract}, {ordered}.

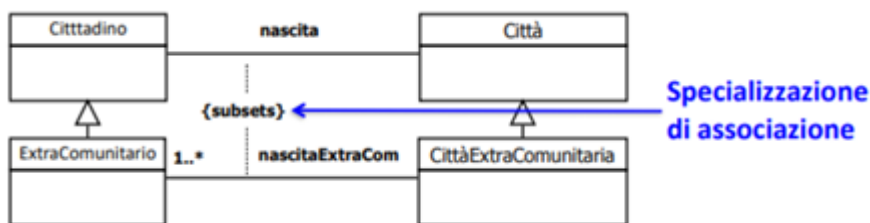
63) Diagramma delle classi Software : Overlapping/Disjoint

Si parla di una **proprietà** in cui i classificatori possono/non possono avere istanze in comune.



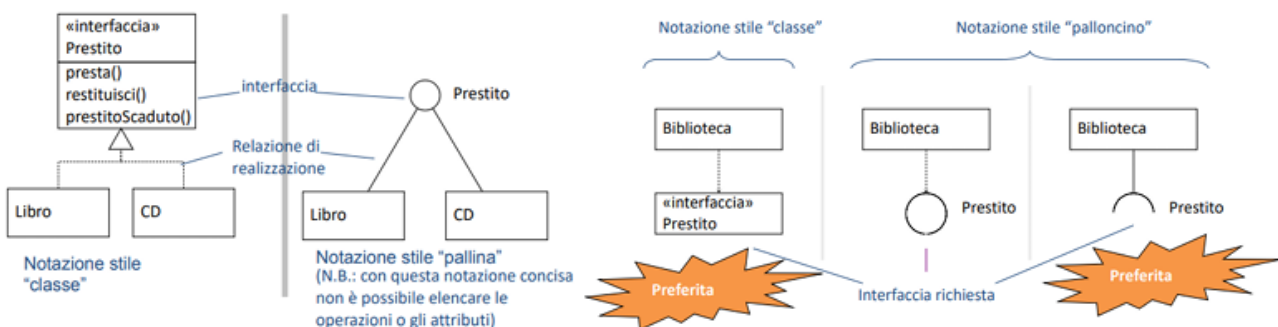
64) Diagramma delle classi Software : Complete/incomplete

Si parla di una **proprietà** in cui l'unione delle istanze delle sottoclassi è/non è uguale all'insieme delle istanze del superclasse.



65) Diagramma delle classi Software : Interfacce

Un'**interfaccia** è un insieme di funzionalità pubbliche identificate da un nome; separa le specifiche di una funzionalità dall'implementazione della stessa. Essa definisce un contratto e tutti i classificatori che la realizzano lo devono rispettare.



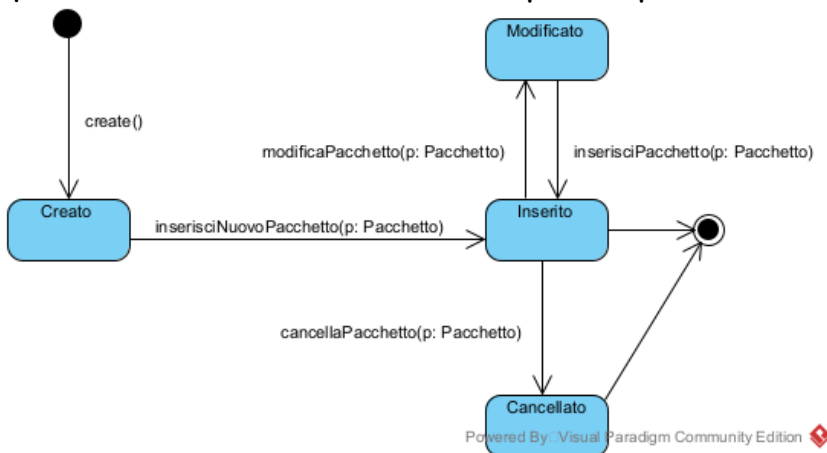
66) Macchina a Stati: definizione

Una **Macchina a Stati** è un diagramma che può essere utilizzato per modellare il comportamento dinamico di classificatori quali classi, casi d'uso, sottosistemi e interi sistemi. Mostra essenzialmente il ciclo di vita di un oggetto, modellato attraverso una successione di stati (condizione di un oggetto in un time slice), transizioni (relazioni etichettate tra gli stati; quando si verifica l'evento l'oggetto cambia stato) ed eventi. Gli oggetti possono dipendere dallo stato, ossia rispondono in modo diverso a seconda dello stato in cui si trovano, o non

dipendere dallo stato, ossia rispondono nello stesso modo ad un determinato evento.

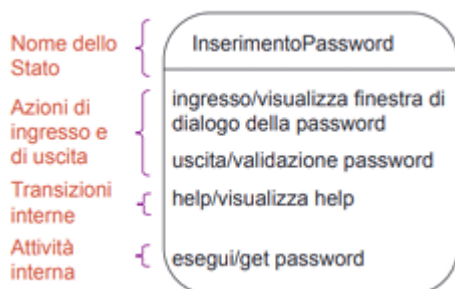
67) Macchina a Stati: rappresentazione

Un **Diagramma di Macchina a Stati** di UML gli **stati** di un **oggetto** vengono rappresentati attraverso dei rettangoli arrotondati, insieme al comportamento dell'oggetto in termini di transizioni - frecce etichettate con delle azioni istantanee non interrompibili - fra gli stati in risposta agli eventi durante la sua vita. Un **ciclo** comincia di solito dallo stato iniziale (pallino pieno) e finisce quando arriva nello stato finale (pallino pieno con bordo).



68) Macchina a Stati: caratteristiche degli stati

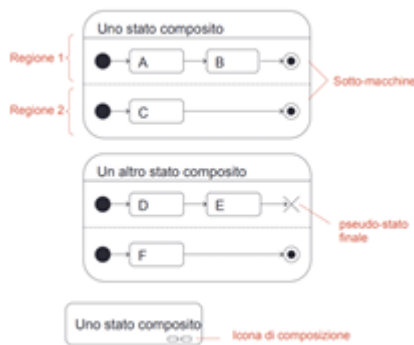
Uno **stato** ha un nome, delle azioni di ingresso e di uscita (archi entranti e uscenti), può avere delle transizioni interne (come se fossero transizioni a cappio, non cambiano lo stato) e delle attività interne, che richiedono un tempo finito e sono interrompibili (sono ciò che viene fatto mentre l'oggetto è in quello stato).



69) Macchina a Stati: tipologie degli stati

Gli **stati** possono essere anche più complessi e tali stati possono essere **compositi** o **pseudostati**. Gli **stati compositi** sono stati che includono una (stati compositi semplici) o più regioni (stati compositi ortogonali) che hanno all'interno delle sotto-macchine. Lo stato finale ferma solo la regione in cui si trova; lo stato finale di stato composito (vedere gli **pseudostati**) termina

l'intero stato composito.



In una **macchina a stati ortogonale** gli stati del sistema sono indipendenti l'uno dall'altro, cioè ogni stato è rappresentato da una variabile distinta che può cambiare indipendentemente dalle altre variabili e se le regioni hanno tutte uno stato finale la terminazione è sincrona, altrimenti la terminazione avviene quando una delle regioni entra nel suo pseudostato di uscita.



Uno **pseudostato** indica uno stato che un oggetto non possiede mai:

- **giunzione**: ricongiunge più archi diretti verso lo stesso stato, si indica con un piccolo pallino pieno;
- **selezione**: è uno stato con un solo input e vari output mutuamente esclusivi con guardia, nel momento in cui il flusso arriva viene preso il ramo con la condizione vera;
- **di ingresso**: pallino vuoto sul bordo di uno stato composito che indica l'entrata nello stesso;
- **di uscita**: pallino vuoto con una x che funge da punto di uscita per uno stato composito;
- **con memoria semplice**: stato che ricorda l'ultimo sotto-stato quando si è lasciato lo stato composito, si indica con un pallino vuoto con all'interno una H;
- **con memoria multilivello**: come lo stato di memoria semplice, ma può ricordarsi i sotto-sotto-stati di eventuali sotto-stati dello stato composito (quindi memorizza gli ultimi stati di tutti gli stati composti presenti nello stato composito in cui è collocato, quest'ultimo compreso); la H diventa H*;
- **finale per gli stati composti**: lo stato finale di un intero stato composito è indicato con una croce.

Inoltre possiamo avere delle linee di **fork** e **join**, che ci permettono di suddividere e ricongiungere (sincronizzandole) linee di flusso diverse.

70) Macchina a Stati: eventi

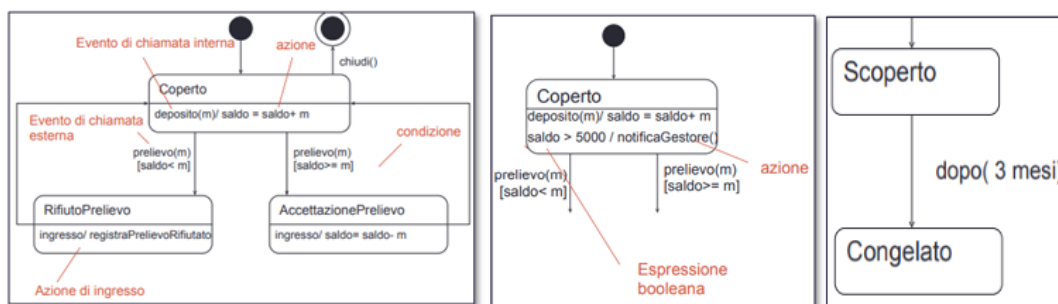
Vi sono altri tipi di **eventi** a disposizione nelle **Macchine a Stati**:

→ **eventi di chiamata**: sono chiamate per specifiche operazioni o una sequenza di operazioni;

→ **eventi di segnale**: comporta l'invio o l'attesa della ricezione (entrambi asincroni) di un segnale, viene indicato con una bandierina;

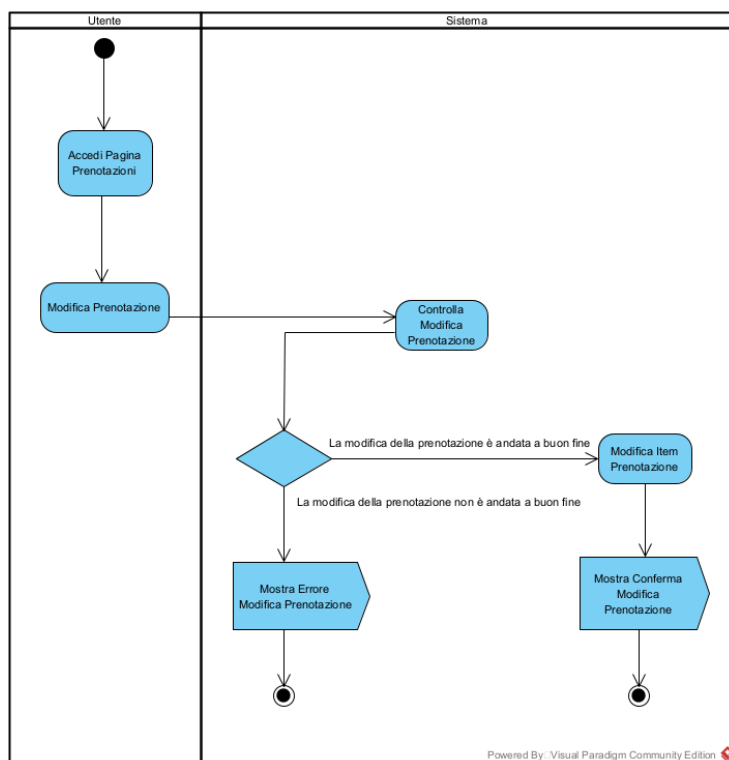
→ **eventi di variazione**: nel momento in cui una condizione viene verificata all'interno di uno stato, viene eseguita l'azione indicata dopo lo slash; non cambia di stato;

→ **evento temporale**: verificano condizioni temporali, dopo() o quando().



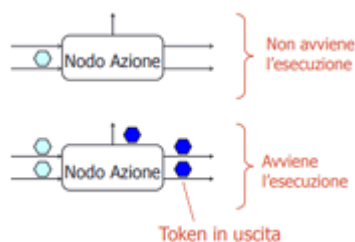
71) Diagramma delle attività: definizione e caratteristiche

Un **Diagramma di Attività** mostra le attività di un processo rappresentate da reti di nodi connessi da archi.





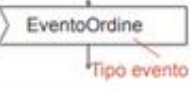
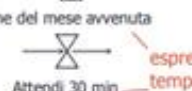
72) Diagramma delle attività: token game

Un **token** è un oggetto, un insieme di dati o un flusso di controllo. Viene passato da nodo a nodo attraverso le transizioni per rappresentare il flusso.










73) Diagramma delle attività: nodo azione

Un **nodo azione** rilascia dei **token** su tutti gli archi in uscita quando tutti gli archi in entrata hanno un **token** e rappresenta unità discrete di lavoro atomiche all'interno dell'attività. Quando è presente un **token** per ogni arco in entrata viene eseguita l'azione del nodo e mandato un **token** in ogni nodo d'uscita. Può invocare un'attività (nodo rastrello), un comportamento o un'operazione.

Sintassi	Semantica
	Nodo azione di chiamata – invoca un'attività, un comportamento o un'operazione. Tipo di nodo azione più comune
	Invia segnale di azione – invia un segnale in modo asincrono. Può accettare i parametri di input necessari per creare il segnale
	Accetta un evento – aspetta gli eventi individuati dal suo oggetto proprietario ed emette l'evento sull'arco in uscita. E' attivo quando riceve un token nel suo arco in entrata. Se non c'è alcun arco entrante, inizia quando la sua attività contenente inizia
	Accetta un evento temporale: risponde al tempo. Genera eventi temporali secondo la sua espressione temporale

74) Diagramma delle attività: nodo controllo

Un **nodo controllo** ispeziona il flusso mediante l'attività.

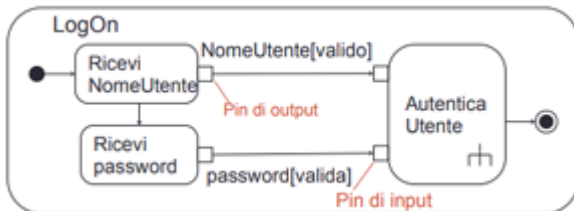
Sintassi	Semantica	Nodi finali
	Nodo iniziale – indica dove inizia il flusso quando invocata un'attività	
	Nodo finale dell'attività – termina un'attività	Nodi finali
	Nodo finale del flusso – termina un flusso specifico all'interno di un'attività. Gli altri flussi non vengono influenzati	
	Nodo decisione – viene attraverso l'arco in uscita la cui condizione di guardia è soddisfatta	
	Nodo fusione – copia token in ingresso nel suo unico arco in uscita	
	Nodo biforcazione – divide il flusso in più flussi concorrenti	
	Nodo ricongiunzione – Sincronizza più flussi concorrenti. Facoltativamente può avere una specifica di ricongiunzione per modificare la sua semantica.	

75) Diagramma delle attività: nodo oggetto

Il **nodo oggetto** rappresenta pile/code di token oggetti usati nell'attività.

76) Diagramma delle attività: pin

Un **pin** è un particolare nodo oggetto che rappresenta un input o un output di un'azione.



77) Diagramma delle attività: partizioni

Una **partizione** è un raggruppamento ad alto livello di azioni correlate; possono essere gerarchiche, verticali o orizzontali, e riferirsi a diverse entità.



Capitolo 9 - Progettazione guidata dalla responsabilità (Pattern)

78) Pattern GRASP: definizione e caratteristiche

I **Pattern GRASP** (**General Responsibility Assignment Software Patterns**)

descrivono dei principi di base per la progettazione di oggetti e l'assegnazione di responsabilità. Un **pattern** è una coppia problema/soluzione ben conosciuta e con un nome (per facilitare comprensione, memorizzazione e comunicazione).

Essi si basano su soluzioni e principi già applicati e dimostratisi corretti in diverse situazioni, si fondano quindi sull'esperienza e non sulla sperimentazione.

79) Pattern GRASP: tipologie

I **Pattern GRASP** si distinguono in 9 tipologie:

- **Creator**: assegna la responsabilità di creare un'istanza di classe alla classe che utilizza direttamente l'oggetto creato, ha i dati necessari per inizializzare l'oggetto, detiene e gestisce l'oggetto ed è strettamente collegata all'oggetto.
- **Information Expert**: assegna una responsabilità a quella classe che ha l'informazione necessaria per soddisfarla.

- **Low Coupling**: assegna le responsabilità in modo da ridurre al minimo l'accoppiamento tra classi per rendere il sistema più modulare e facile da modificare.
- **Controller**: assegna la responsabilità di gestire le richieste del sistema a un oggetto controller, che agisce come intermediario tra l'interfaccia utente e il sistema e può essere rappresentato da una classe di interfaccia utente, un gestore di casi d'uso, o una classe coordinatrice del sistema.
- **High Cohesion**: assegna le responsabilità in modo da mantenere alta la coesione delle classi e una classe è coesa se le sue responsabilità sono strettamente correlate e mirano a uno scopo specifico.
- **Pure Fabrication**: crea una classe che non rappresenta un concetto del dominio reale per assegnare una responsabilità che non può essere assegnata con efficacia utilizzando Information Expert o altri pattern ed è utile per mantenere il basso accoppiamento e l'alta coesione.
- **Polymorphism**: assegna responsabilità per comportamenti che variano a classi tramite l'uso del polimorfismo e le classi possono implementare la stessa interfaccia o estendere una classe base, permettendo la flessibilità nelle implementazioni.
- **Indirection**: assegna responsabilità a un intermediario per evitare un accoppiamento diretto tra due o più classi e introduce una classe intermedia per gestire l'interazione tra le classi.
- **Protected Variations**: protegge le variazioni nel comportamento del sistema dai cambiamenti e si ottiene usando interfacce, classi astratte, e altre tecniche per isolare le parti che cambiano dalle parti stabili del sistema.

80) Pattern GoF (Design Pattern): definizione e caratteristiche

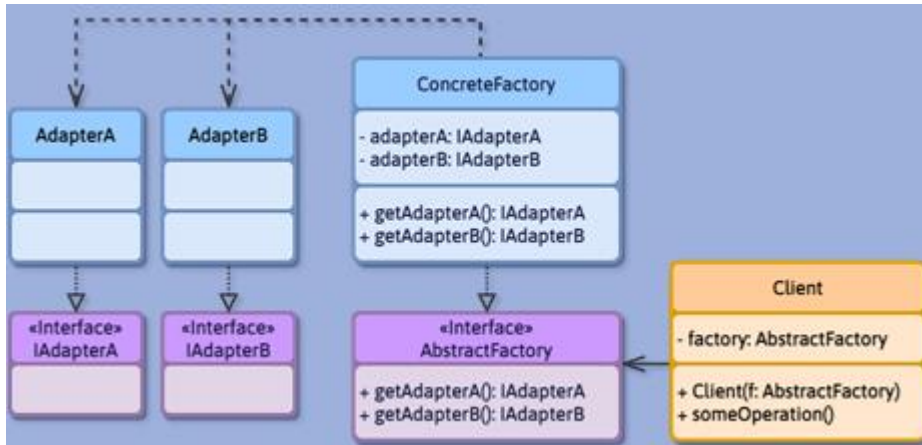
I **Design Pattern** sono una soluzione progettuale comune a un problema di progettazione ricorrente; molti design pattern possono essere analizzati in termini dei pattern **GRASP**. A differenza di questi ultimi, che sono "solo" consigli, i **Design Pattern** mostrano anche lo schema delle classi per l'implementazione.

81) Pattern GoF (Design Pattern): tipologie

I **Design Pattern** si suddividono in tre categorie principali: creazionali, strutturali e comportamentali. Di seguito una panoramica delle tipologie principali di design pattern all'interno di ciascuna categoria.

Pattern Creazionali

1. Factory Method: Fornisce un'interfaccia per creare oggetti in un superclasse, ma permette alle sottoclassi di alterare il tipo di oggetti che verranno creati.

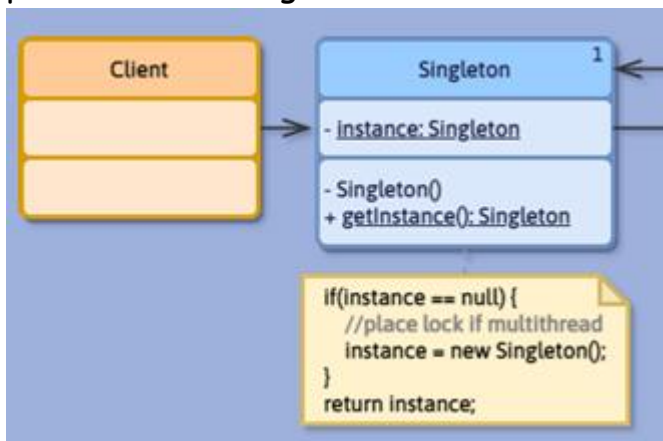


2. Abstract Factory: Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.

3. Builder: Separare la costruzione di un complesso oggetto dalla sua rappresentazione, in modo che lo stesso processo di costruzione possa creare rappresentazioni diverse.

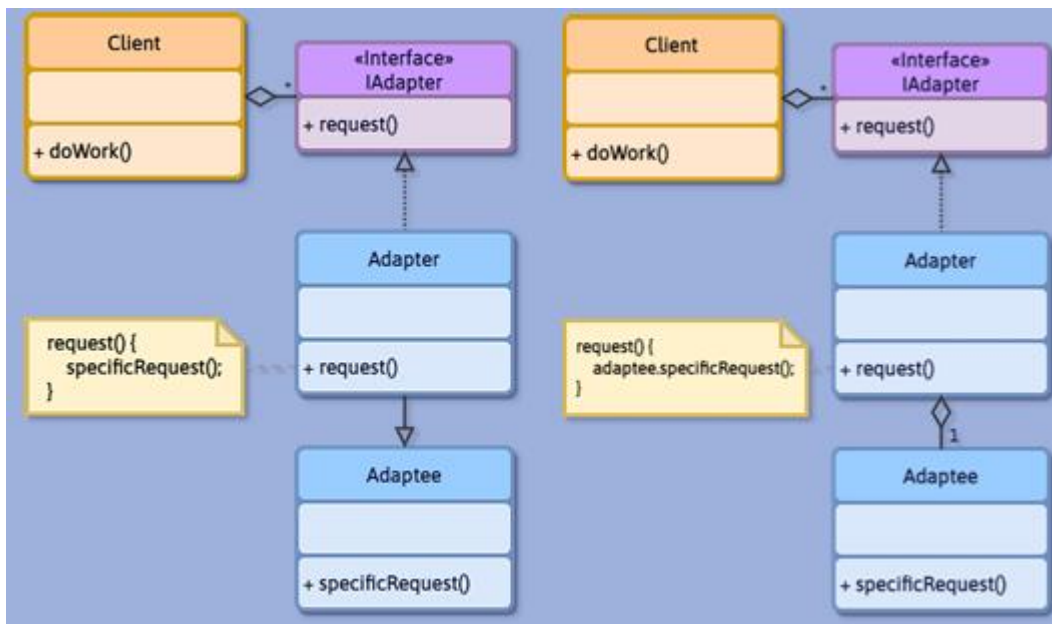
4. Prototype: Specifica i tipi di oggetti da creare utilizzando un'istanza prototipica, e crea nuovi oggetti copiando questo prototipo.

5. Singleton: Garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a essa.



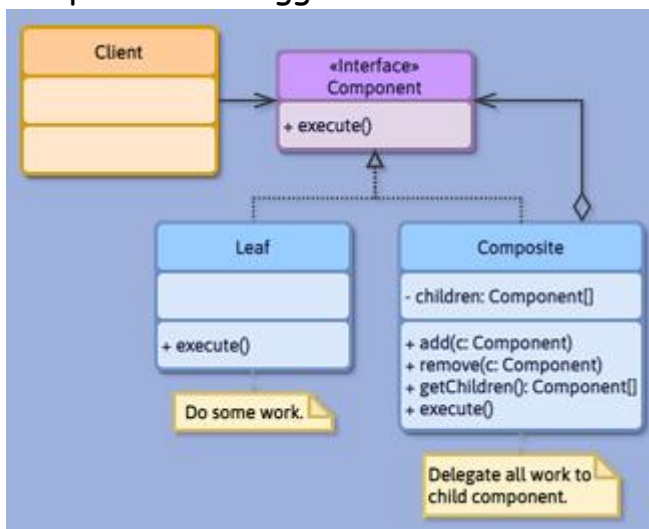
Pattern Strutturali

1. Adapter: Permette a classi con interfacce incompatibili di lavorare insieme, convertendo l'interfaccia di una classe in un'altra interfaccia che i client si aspettano.



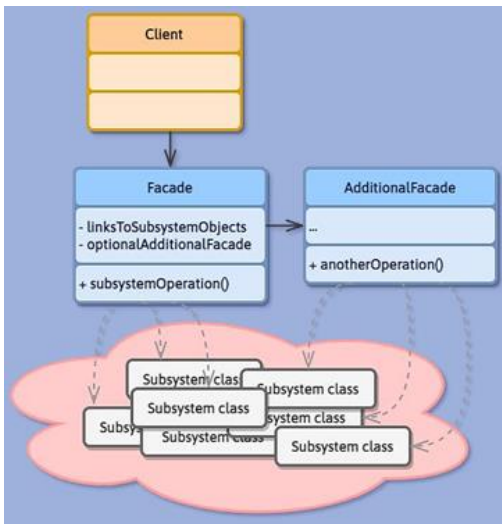
2. Bridge: Separa un'astrazione dalla sua implementazione in modo che le due possano variare indipendentemente.

3. Composite: Compone oggetti in strutture ad albero per rappresentare gerarchie parte-tutto. Permette ai client di trattare oggetti singoli e composizioni di oggetti in modo uniforme.



4. Decorator: Aggiunge dinamicamente responsabilità aggiuntive a un oggetto. I decorator forniscono una flessibile alternativa alla sotto classificazione per estendere le funzionalità.

5. Facade: Fornisce un'interfaccia unificata per un insieme di interfacce in un sottosistema. Fa apparire un sottosistema più facile da usare.



6. Flyweight: Usa la condivisione per supportare un gran numero di oggetti a granularità fine in modo efficiente.

7. Proxy: Fornisce un surrogato o un segnaposto per controllare l'accesso a un altro oggetto.

Pattern Comportamentali

1. Chain of Responsibility: Evita di accoppiare il mittente di una richiesta con il suo destinatario, dando a più oggetti la possibilità di trattare la richiesta. Collega gli oggetti riceventi e passa la richiesta lungo la catena fino a quando un oggetto la tratta.

2. Command: Incapsula una richiesta come un oggetto, permettendo di parametrizzare i client con richieste, accodare richieste o registrare richieste.

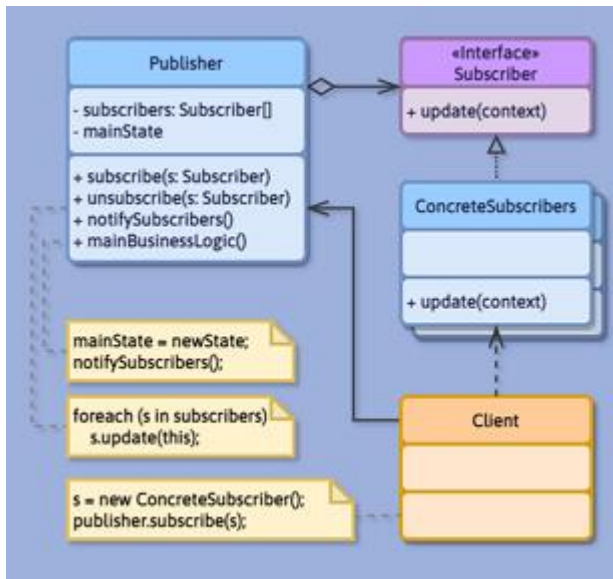
3. Interpreter: Dato un linguaggio, definisce una rappresentazione della sua grammatica insieme a un interprete che usa la rappresentazione per interpretare le frasi del linguaggio

4. Iterator: Fornisce un modo per accedere agli elementi di un aggregato sequenziale senza esporre la sua rappresentazione sottostante.

5. Mediator: Definisce un oggetto che incapsula come un insieme di oggetti interagisce. Promuove il disaccoppiamento riducendo le dipendenze dirette tra gli oggetti.

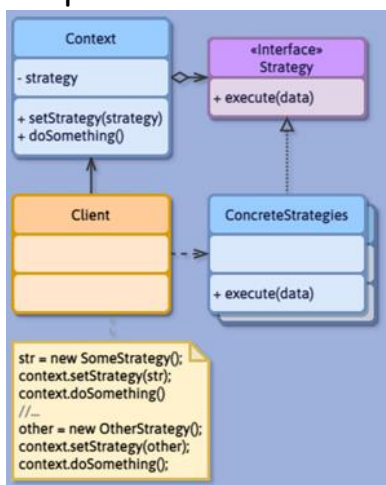
6. Memento: Cattura e externalizza lo stato interno di un oggetto senza violare l'incapsulamento, permettendo all'oggetto di essere ripristinato a questo stato in un momento successivo.

7. Observer: Definisce una dipendenza uno-a-molti tra oggetti in modo tale che quando un oggetto cambia stato, tutti i suoi dipendenti vengono notificati e aggiornati automaticamente.



8. State: Permette a un oggetto di alterare il proprio comportamento quando il suo stato interno cambia. Sembra che l'oggetto cambi classe.

9. Strategy: Definisce una famiglia di algoritmi, incapsula ciascuno di essi, e li rende intercambiabili. Permette di selezionare l'algoritmo da usare in modo indipendente dal client che lo utilizza.



10. Template Method: Definisce la struttura di un algoritmo, delegando la definizione di alcuni passaggi alle sottoclassi. Permette alle sottoclassi di ridefinire certi passaggi di un algoritmo senza cambiare la struttura dell'algoritmo stesso.

11. Visitor: Rappresenta un'operazione da eseguire sugli elementi di una struttura di oggetti. Permette di definire una nuova operazione senza cambiare le classi degli elementi su cui opera.

Capitolo 10 - Testing e Refactoring

82) Quali sono i tipi di test?

Si hanno i seguenti tipi di test:

- **test di unità**: test relativi a singole classi e metodi, per verificare il funzionamento delle piccole parti ("unità") del sistema;
- **test di integrazione**: per verificare la comunicazione tra varie parti (elementi architetturali) del sistema;
- **test di sistema**: detto anche test end-to-end, per verificare il collegamento complessivo tra tutti gli elementi del sistema;
- **test di accettazione**: per verificare il funzionamento complessivo del sistema a scatola nera e dal punto di vista dell'utente, con riferimenti a scenari di casi d'uso.

83) Test di unità: definizione e caratteristiche

Un **test di unità** consiste in:

- **preparazione**: creare l'oggetto da verificare (quest'ultimo chiamato **fixture**) e preparare altri oggetti/risorse necessari per l'esecuzione del test;
- **esecuzione**: far eseguire operazioni alla **fixture**, richiedendo lo specifico comportamento da verificare;
- **verifica**: valutare che i risultati ottenuti corrispondano a quelli previsti;
- **rilascio**: opzionalmente rilasciare/ripulire oggetti e risorse utilizzate nel test, per evitare che altri test vengano corrotti.

84) Tipi di cicli per i test unitari

Il **TDD** si basa su cicli di lavorazione molto brevi, guidati dalle seguenti regole:

- scrivere un test unitario che fallisce, per dimostrare la mancanza di una funzionalità o di codice;
- scrivere il codice più semplice possibile per far passare il test;
- riscrivere o ristrutturare (refactor) il codice, migliorandolo, oppure passare a scrivere il prossimo test unitario (il refactor può essere effettuato anche dopo diversi test).



Il ciclo di base del TDD per i test unitari.

Il doppio ciclo del TDD.

85) Refactoring: definizione e caratteristiche

Il **refactoring** è un metodo strutturato e disciplinato, utilizzato per riscrivere/ristrutturare del codice esistente senza modificarne il

comportamento esterno. Applica piccoli passi di trasformazione (uno alla volta), in combinazione con la ripetizione dei test dopo ciascun passo, per dimostrare che il refactoring non abbia provocato una regressione (fallimento). I vantaggi sono il miglioramento continuo del codice e la preparazione alle modifiche (o introduzione di nuove funzionalità).

86) Code Smell: definizioni e caratteristiche

Un **code smell** indica una serie di caratteristiche nel codice che possono essere indice di cattive pratiche di programmazione (**difetti di programmazione**).

Un **code smell** può essere:

→ **Bloaters**: metodi, classi o in generale codice di dimensione sproporzionata (troppo grandi):

a) **Long Method**: un metodo ha troppe righe di codice.

b) **Large Class**: una classe ha troppi metodi.

c) **Long Parameter List**: un metodo ha troppi parametri.

→ **Object-Orientation Abusers**: implementazioni errate, incomplete o deboli dei principi della programmazione OO:

a) **Switch Statement**: si ha uno switch molto lungo o più if...else.

b) **Refused Bequest**: la sottoclasse non usa tutti i metodi e le proprietà ereditate, la gerarchia è errata.

→ **Change Preventers**: parti di codice altamente accoppiate che impediscono cambiamenti rapidi:

a) **Shotgun Surgery**: per modificare il codice vanno apportate minime modifiche a molte classi diverse.

→ **Dispensables**: codice inutile, che sarebbe meglio togliere o integrare in altre parti per rendere il codice più pulito, efficiente e comprensibile:

a) **Duplicated Code**: lo stesso codice appare in molti luoghi.

b) **Data Class**: classe che funge solo da contenitore per i dati.

c) **Comments**: troppi commenti.

→ **Couplers**: classi che hanno eccessivi accoppiamenti o classi che delegano eccessivamente:

a) **Feature Envy**: Un metodo fa accedere ai dati di un altro oggetto più che ai propri dati.