Analisi e Progettazione del Software

Appunti non ufficiali - Theofilia Jessica
Prof: Riganelli (T1)

Anno 2023/2024

Contents

Chapter 1

Introduzione al corso e ai processi di sviluppo

Il software è un programma e la relativa documentazione: specifica requisiti, modelli, progetto, manuale utente. Un software può essere un prodotto:

1. **Generico**, ovvero per un ampio insieme di clienti
2. **Personalizzato**, per un singolo cliente

Le caratteristiche di un buon software sono:

- **Mantenibilità**: Il software dovrebbe essere facilmente comprensibile e modificabile da futuri collaboratori.
- **Affidabilità**: Deve essere robusto, sicuro e protetto da accessi non autorizzati o malfunzionamenti.
- **Efficienza**: Il software non dovrebbe sprecare risorse, dovrebbe reagire prontamente agli input e utilizzare CPU e memoria RAM in modo contenuto.
- **Accettabilità**: Deve essere compatibile e in grado di interagire con altri sistemi.

L'ingegneria del software è una disciplina ingegneristica che si occupa di tutti gli aspetti relativi alla produzione di software di alta qualità. Questo include ogni fase, dalle prime specificazioni del sistema fino alla sua manutenzione successiva alla messa in uso.

- *È una disciplina ingegneristica*: Implica l'uso di teorie e metodi specifici per risolvere problemi, considerando sempre i vincoli organizzativi e finanziari.
- *Copre tutti gli aspetti della produzione del software*: Non si limita al solo processo tecnico di sviluppo, ma include anche la gestione del progetto e lo sviluppo di strumenti e metodologie a supporto della creazione del software.

Le attività di processo fondamentali, comuni a tutti i processi software, sono le seguenti:

1. *Analisi dei requisiti*: Consiste nel definire in modo preciso i servizi che il software deve offrire.
2. *Progettazione*: Riguarda l'organizzazione della struttura complessiva del sistema software.

3. *Implementazione*: È la fase in cui il progetto si traduce in codice effettivo.

4. *Evoluzione*: Permette di modificare il software in base alle nuove esigenze o ai cambiamenti.

5. *Validazione*: Consiste nel verificare che il sistema sviluppato soddisfi le richieste e le aspettative del cliente.

In questo corso, ci concentreremo principalmente sulle prime due fasi: l'**analisi** (fare la cosa giusta) e la **progettazione** (fare la cosa bene).
Una descrizione dei processi ci viene fornita tramite:

- Artefatti: Questi sono gli elementi o i prodotti che devono essere realizzati durante le attività del processo.

- Ruoli: Indicano come le persone partecipano e interagiscono all'interno del processo.

- Pre e Post Condizioni: Si riferiscono alle regole o ai requisiti che devono essere soddisfatti prima di poter eseguire un'attività (pre-condizioni) e alle condizioni che devono essere vere dopo il completamento di un'attività (post-condizioni).

1.1 Le quattro fasi dell' Analisi e Progettazione

- L'**analisi** si concentra sull'indagine di un problema e dei suoi requisiti, piuttosto che sulla definizione di una soluzione.

- La **progettazione**, invece, si focalizza su una soluzione concettuale che risponde ai requisiti del problema.

- Nell'**analisi orientata agli oggetti**, l'attenzione si concentra sull'identificazione dei concetti, o degli oggetti, presenti nel contesto del problema.

- La **progettazione orientata agli oggetti**, invece, si focalizza sulla definizione di oggetti software che interagiscono tra loro per soddisfare i requisiti stabiliti.

In sintesi, l'analisi significa "fare la cosa giusta" (capire cosa serve), mentre la progettazione significa "fare la cosa bene" (realizzarla nel modo migliore). Sebbene analisi e progettazione abbiano obiettivi distinti e vengano perseguite con approcci diversi, sono attività profondamente sinergiche e complementari.
L'analisi e la progettazione del software possono includere quattro fasi principali:

1. Casi d'uso: Si tratta di descrizioni narrative di come il sistema verrà utilizzato dagli utenti.

2. Modello di dominio: Questo modello rappresenta i concetti o gli oggetti chiave del dominio del problema e le relazioni tra di essi.

3. Definizione dei diagrammi di interazione: Forniscono una visione dinamica, mostrando come gli oggetti software collaborano tra loro per eseguire specifiche operazioni.

4. Definizione dei diagrammi delle classi: Offrono una rappresentazione statica delle classi software, inclusi i loro attributi (dati) e metodi (funzioni).

1.2 Che cosa è UML

L'Unified Modeling Language (UML) è un linguaggio di modellazione visuale utilizzato per i sistemi software e non solo. Rappresenta una raccolta di best practice ingegneristiche che si sono dimostrate efficaci nella modellazione di sistemi complessi e di grandi dimensioni.
UML facilita la diffusione delle informazioni nella comunità dell'ingegneria del software, essendo diventato uno standard "de facto". È importante sottolineare che **UML non è una metodologia**, ma un linguaggio visuale.

UML modella i sistemi come insiemi di oggetti che collaborano tra loro. Si occupa di:

- ***Struttura Statica***: Definisce quali tipi di oggetti sono necessari e come sono correlati tra loro.
- ***Struttura Dinamica***: Descrive il ciclo di vita di questi oggetti e come collaborano per fornire le funzionalità richieste.

UML può essere impiegato a diversi livelli di dettaglio:

- *UML come abbozzo*: Utilizzato per creare diagrammi informali e incompleti.
- *UML come progetto*: Per la creazione di diagrammi di progetto relativamente dettagliati.
- *UML come linguaggio di programmazione*: Consente la specifica completamente eseguibile di un sistema software.

La modellazione agile, ad esempio, predilige l'uso di UML come abbozzo.

UML descrive tipi "grezzi" di diagrammi e non impone un punto di vista di modellazione specifico per il loro utilizzo. Possiamo adottare due punti di vista principali:

1. Punto di vista concettuale: I diagrammi descrivono oggetti del mondo reale o di un dominio di interesse.
2. Punto di vista software: I diagrammi descrivono astrazioni o componenti software.

È importante notare che entrambi i punti di vista utilizzano la stessa notazione UML.

Una **classe concettuale** rappresenta un oggetto o un concetto del mondo reale, focalizzandosi sull'essenza. Il Modello di Dominio contiene classi concettuali. Una **classe software** è una classe che rappresenta un componente software. Il Modello di Progetto, invece, contiene classi software.

Disegnare o leggere diagrammi UML significa lavorare in modo visuale. Questo sfrutta la capacità del nostro cervello di comprendere rapidamente simboli, unità e relazioni mostrati attraverso una notazione grafica (principalmente bidimensionale), come quelle rappresentate da rettangoli e linee.

1.3 Processi software

1.4 Processo di sviluppo

Un processo di sviluppo software definisce l'approccio per la sua costruzione, specificando:

- **Cosa** sono le attività da svolgere.
- **Chi** ricopre i ruoli e le responsabilità.
- **Come** vengono applicate le metodologie.
- **Quando** si organizza temporalmente il lavoro.

Le attività fondamentali di un processo software includono: Requisiti, Analisi, Progettazione, Implementazione, Validazione, Rilascio e installazione, Manutenzione ed Evoluzione, e Gestione del Progetto. L'analisi si concentra sull'*investigazione del problema e dei suoi requisiti*, mentre la progettazione si focalizza sulla *soluzione concettuale*.
Esistono diversi modelli di processo software a causa della complessità del progetto, della dimensione del team, del budget e delle tempistiche, e del rischio:

1. I **processi basati sul piano (o "waterfall")** prevedono una pianificazione dettagliata in anticipo, con fasi sequenziali e distinte. Questo modello è più adatto quando i requisiti sono ben compresi e stabili, poiché la sua inflessibilità rende difficile rispondere a cambiamenti.
2. I **processi agili**, invece, hanno una pianificazione incrementale e sono più flessibili nel recepire le mutevoli esigenze dei clienti. Molti processi pratici combinano elementi di entrambi gli approcci.
3. L'**integrazione e configurazione** è un modello basato sul riutilizzo di componenti o sistemi esistenti (COTS - Commercial-off-the-shelf). Questo approccio riduce costi e rischi e accelera la consegna, ma può comportare compromessi sui requisiti e una perdita di controllo sull'evoluzione degli elementi riutilizzati.

1.5 Sviluppo Iterativo, Incrementale ed Evolutivo

Lo sviluppo iterativo, incrementale ed evolutivo è una forma di sviluppo agile, dove le iterazioni hanno **durata fissa** (timeboxed, solitamente 2-6 settimane) e il sistema cresce in modo incrementale. Questo approccio **riduce i rischi** precocemente, offre progressi visibili e permette un feedback continuo. Richiede però che il software sia **flessibile**, **facilmente modificabile** e comprensibile.
La pianificazione è iterativa, il che significa che non si tenta di pianificare l'intero progetto in dettaglio fin dall'inizio. Piuttosto, si cerca di applicare un approccio basato su:

- *Guida dal rischio*: Gli obiettivi delle iterazioni iniziali vengono scelti per identificare e attenuare i rischi maggiori.
- *Guida dal cliente*: Si mira a costruire e rendere visibili le caratteristiche a cui il cliente tiene di più.

Il **Processo Unificato** (UP) è un processo iterativo diffuso per lo sviluppo software orientato agli oggetti. È flessibile e aperto, pilotato dai casi d'uso e dai fattori di rischio, incentrato sull'architettura e iterativo, incrementale ed evolutivo. UP organizza il lavoro in quattro fasi temporali:

1. **Ideazione**: Avvio del progetto, visione approssimativa, stime iniziali.
2. **Elaborazione**: Realizzazione del nucleo dell'architettura, risoluzione dei rischi maggiori e definizione dei requisiti.
3. **Costruzione**: Implementazione iterativa degli elementi rimanenti e preparazione al rilascio.
4. **Transizione**: Completamento del prodotto e rilascio.

Ogni iterazione in UP è un mini-progetto con **pianificazione**, **analisi**, **progettazione**, **costruzione**, **integrazione** e **test**, che porta a un rilascio. Una **Release** è un insieme di manufatti, previsti e approvati. Un incremento è la differenza tra una release e la successiva.

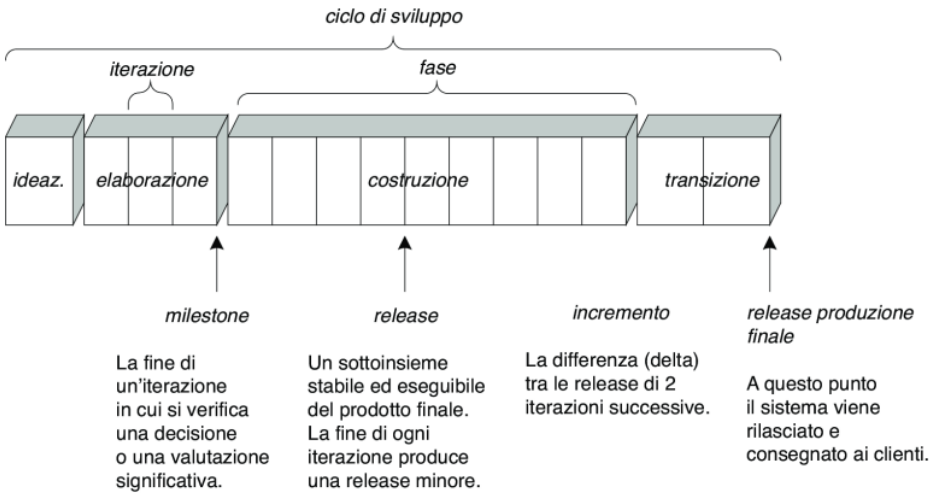


Figure 1.1: Struttura di UP

Quasi tutti gli elaborati e le pratiche del Processo Unificato (UP) sono **opzionali**. Alcune pratiche e principi di UP sono fissi, come lo sviluppo iterativo e guidato dal rischio e la verifica continua della qualità. Tutte le attività e i prodotti (modelli, diagrammi, documenti, ecc.) sono opzionali, con l'unica ovvia esclusione del codice.
La scelta delle pratiche e degli elaborati UP per un progetto può essere documentata in un breve scritto chiamato "**scenario di sviluppo**".

1.6 Sviluppo Agile

Lo sviluppo agile è una forma di sviluppo iterativo che promuove una risposta rapida e flessibile ai cambiamenti. I metodi agili applicano lo sviluppo iterativo con **iterazioni brevi** e a **durata fissa** (timeboxed), e una pianificazione iterativa. Essi promuovono le consegne incrementali e sostengono valori "agili" come la **semplicità**, la **leggerezza**, il

valore delle persone e la **comunicazione**. Inoltre, supportano pratiche "agili" come la programmazione a coppie, il TDD (Test-Driven Development) e il refactoring. Qualsiasi processo iterativo, incluso l'Unified Process (UP), può essere applicato con uno spirito agile.

Agile Modeling si basa su un insieme di pratiche e valori, tra cui l'idea che adottare un metodo agile non significa evitare del tutto la modellazione. Lo scopo dei modelli e della modellazione è **facilitare la comprensione** e la **comunicazione**. È consigliabile modellare e applicare UML solo alle parti del progetto che sono difficili, insolite e insidiose, usando lo strumento più semplice possibile. La modellazione non dovrebbe essere fatta da soli, ma piuttosto a coppie (o in tre), tenendo presente che tutti i modelli saranno incompleti e imprecisi.

UP agile è un'applicazione agile del Processo Unificato. UP non è stato concepito per essere un processo "pesante" o non agile, dato che molti dei suoi elementi sono opzionali e permettono la personalizzazione. Un UP agile si caratterizza per un piccolo insieme di attività ed elaborati. I requisiti e la progettazione non vengono completati prima dell'implementazione, ma emergono in modo adattivo durante una serie di iterazioni, anche sulla base del feedback. Questo include l'applicazione di UML secondo lo spirito della modellazione agile e una pianificazione iterativa e adattiva.

Scrum è un altro metodo agile focalizzato sull'organizzazione del lavoro e la gestione dei progetti. Si basa su cicli chiamati "Sprint" (2-4 settimane) che producono un "Incremento di prodotto potenzialmente rilasciabile". Gli incontri giornalieri ("Daily Scrum") servono a esaminare i progressi e definire le priorità. I ruoli chiave includono lo ScrumMaster e il Product Owner. I termini scrum sono descritti nella Tabella 1.1.

Termine Scrum	**Descrizione**
Scrum	Un incontro giornaliero del team Scrum che esamina i progressi e definisce le priorità del lavoro da svolgere in quel giorno. Idealmente, è un breve incontro faccia a faccia che include l'intera squadra.
ScrumMaster	Responsabile di assicurare che il processo Scrum sia seguito e guida il team nell'uso efficace di Scrum. Si interfaccia con il resto dell'azienda e assicura che il team non sia deviato da interferenze esterne.
Sprint	Un'iterazione di sviluppo, solitamente di 2-4 settimane.
Velocity	Una stima di quanto lavoro rimanente un team può fare in un singolo sprint. Aiuta a stimare ciò che può essere coperto in uno sprint e fornisce una base per misurare il miglioramento delle prestazioni.
Team di sviluppo	Un gruppo auto-organizzato di sviluppatori software, idealmente non più di 7 persone. Sono responsabili dello sviluppo del software e di altri documenti essenziali del progetto.
Incremento potenzialmente rilasciabile	L'incremento software fornito da uno sprint. L'idea è che sia "potenzialmente trasportabile", ovvero in uno stato finito senza necessità di ulteriore lavoro, come il testing, per essere incorporato nel prodotto finale. In pratica, questo non è sempre realizzabile.
Product backlog	Un elenco di elementi "da fare" che il team Scrum deve affrontare. Possono essere definizioni di caratteristiche, requisiti software, storie utente o descrizioni di compiti supplementari.
Product owner	Un individuo (o un piccolo gruppo) il cui compito è identificare le caratteristiche o i requisiti del prodotto, prioritizzarli per lo sviluppo e rivedere continuamente il product backlog per assicurare che il progetto soddisfi le esigenze di business critiche.

Table 1.1: Descrizioni dei termini Scrum

Chapter 2

Analisi dei requisiti e casi d'uso

2.1 Analisi dei requisiti

2.1.1 Ideazione

La maggior parte dei progetti inizia con una breve fase iniziale per esaminare questioni fondamentali come: qual è la visione e lo studio economico del progetto? Il progetto è fattibile? Conviene comprare o costruire? Qual è la stima approssimativa dei costi (decine di migliaia, centinaia di migliaia o milioni di euro)? E soprattutto, dovremmo procedere o fermarci?

Lo scopo dell'ideazione è triplice:

- Stabilire una visione comune per gli obiettivi del progetto.
- Stabilire se il progetto è fattibile.
- Decidere se vale la pena di effettuare indagini più approfondite nella fase di elaborazione.

A tal fine, l'Ideazione richiede una certa esplorazione dei requisiti, ad esempio circa il **10% dei requisiti funzionali**.

È importante sottolineare che **l'Ideazione non è la fase dei requisiti**. La maggior parte dell'analisi dei requisiti avviene durante la fase di elaborazione, in parallelo alle prime attività di programmazione di qualità-produzione e di test.

Definizione: Requisito

Un Requisito è una capacità o condizione a cui il sistema deve essere conforme. Il 34% delle cause di fallimenti di progetti riguardano attività di requisiti. Possono essere:

- *Funzionali*: descrivono il comportamento del sistema in termini di funzionalità ed informazioni
- *Non Funzionali*: sono relativi al sistema: sicurezza, prestazioni ecc. Possono essere molto difficili da dimostrare con precisione e quindi è utile verificarli attraverso unità di misura.

La fase di ideazione può essere particolarmente breve e può includere il primo workshop sui requisiti e la pianificazione per la prima iterazione dell'elaborazione. Per quanto riguarda gli elaborati iniziati durante l'ideazione, il loro contenuto dovrebbe essere "leggero". Alcuni elaborati vengono iniziati (spesso semplicemente abbozzati) in questa fase. ***Non dovrebbero essere creati elaborati se non si ritiene che aggiungano un valore pratico effettivo***.

Una linea guida per la scrittura dei requisiti include le seguenti regole:

- Ideare un formato standard e utilizzarlo per tutti i requisiti.
- Utilizzare il linguaggio in modo consistente: "DEVE" per i requisiti obbligatori e "DOVREBBE" per quelli desiderabili.
- Utilizzare l'evidenziazione per identificare le porzioni più importanti dei requisiti.
- Evitare il gergo informatico.
- Includere una spiegazione (razionale) del motivo per cui è necessaria una disposizione.

2.1.2 Casi d'uso

I casi d'uso sono storie scritte ampiamente utilizzate per scoprire e registrare i requisiti. Un caso d'uso rappresenta un dialogo tra un attore e un sistema che svolge un compito. Sebbene non siano elaborati orientati agli oggetti, influenzano molti aspetti di un progetto, inclusa l'analisi e la progettazione orientata agli oggetti. Rappresentano un metodo semplice per descrivere i requisiti funzionali, essendo direttamente comprensibili dai clienti e permettendo il loro coinvolgimento nella definizione e revisione. I casi d'uso mettono in risalto gli obiettivi degli utenti e il loro punto di vista, sono utili per produrre la guida utente e per i test di sistema.

I **casi d'uso** sono requisiti, in particolare requisiti funzionali o comportamentali, che indicano cosa deve fare il sistema. Possono essere usati anche per altri tipi di requisiti se sono strettamente correlati a un caso d'uso. Un caso d'uso definisce un contratto relativo al comportamento di un sistema.

Un **attore** è un'entità, persona o sistema, che manifesta un comportamento. Ad esempio, un cassiere o un sistema di pagamento possono essere considerati attori. Esistono diversi tipi di attore:

- ***Attore primario***: raggiunge degli obiettivi usando il Sistema in Discussione (SuD).
- ***Attore finale***: vuole che il SuD sia 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 SuD.

Uno **scenario** (o istanza del caso d'uso) è una specifica sequenza di azioni e interazioni tra il sistema e uno o più attori. Descrive una storia particolare nell'uso del sistema, come un acquisto con pagamento in contanti. Gli scenari possono essere di successo o di fallimento, come un acquisto fallito a causa di un credito non autorizzato.

Un **caso d'uso** è una collezione di scenari correlati, sia di successo che di fallimento, che descrivono come un attore utilizza un sistema per raggiungere un obiettivo specifico.

Il **Modello dei Casi d'Uso** è l'insieme di tutti i casi d'uso scritti. Possono essere scritti usando diversi formati e livelli di formalità:

- *Formato breve*: un riepilogo conciso di un solo paragrafo, normalmente relativo al solo scenario principale di successo.
- *Formato informale*: più paragrafi scritti in modo informale, relativi a vari scenari.
- *Formato dettagliato*: tutti i passi e le variazioni sono scritti nel dettaglio, includendo sezioni di supporto come pre-condizioni e garanzie di successo.

È consigliabile scrivere i casi d'uso in uno stile essenziale, ignorando l'interfaccia utente e concentrandosi sullo scopo dell'attore. Devono essere concisi ma completi, con una chiara indicazione di soggetto, verbo ed eventuali frasi subordinate. La scrittura deve essere a **scatola nera**, specificando cosa il sistema deve fare (comportamento o requisiti funzionali) senza decidere come lo farà (progettazione), descrivendo le responsabilità senza il funzionamento interno. È fondamentale adottare un punto di vista dell'attore e dell'attore-obiettivo, concentrandosi sugli utenti o attori del sistema, i loro obiettivi e le situazioni tipiche, e ciò che l'attore considera un risultato di valore.

Per trovare i casi d'uso:

1. Scegliere il *confine di sistema*.
2. Identificare gli *attori primari*.
3. Identificare gli *obiettivi* per ogni attore primario.
4. Definire i *casi d'uso* che soddisfano questi obiettivi.

È importante stabilire i confini del sistema, comprendendo cosa ne è al di fuori. Una volta identificati gli attori esterni, i confini diventano più chiari. Ad esempio, la responsabilità delle autorizzazioni e dei pagamenti in un sistema POS rientra in un attore esterno, non completamente nei confini del sistema. Un altro approccio per identificare attori, obiettivi e casi d'uso è l'analisi degli eventi che sollecitano il sistema (quali eventi sono di interesse, chi li genera, perché).
In generale, va definito un caso d'uso per ciascun obiettivo utente, assegnandogli un nome simile all'obiettivo e facendolo iniziare con un verbo (es. "Elabora Vendita" per l'obiettivo "elaborare una vendita"). Un'eccezione comune riguarda gli obiettivi CRUD (Create, Retrieve, Update, Delete), che vengono normalmente raggruppati in un singolo caso d'uso "Gestisci X", come "Gestisci Utenti".

Per verificare l'utilità dei casi d'uso, invece di chiedere cosa sia un caso d'uso valido, è più pratico domandare quale sia il livello di attenzione utile per esprimere casi d'uso per l'analisi dei requisiti applicativi. Si applicano delle regole generali:

- **Test del capo**: Se alla domanda "Che fai tutto il giorno?" la risposta è "il login!", e il capo non è felice, il caso d'uso non è fortemente mirato a ottenere risultati con valore misurabile. Questo non significa ignorare sempre questi casi d'uso, poiché l'autenticazione utente potrebbe essere importante e complessa.

- **Test EBP (Elementary Business Process)**: Un processo di business elementare non è un singolo passo (il suo scenario principale conterrà probabilmente 5-10 passi), è un'attività eseguita in una sola sessione di lavoro (probabilmente tra alcuni minuti e un'ora), in risposta a un evento di business, aggiunge un valore di business osservabile e misurabile, e lascia il sistema e i dati in uno stato stabile e coerente. Ci si dovrebbe concentrare sui casi d'uso che corrispondono agli EBP.

- **Test della Dimensione**: Un buon caso d'uso non dovrebbe essere troppo breve, ma comprendere diversi passi e, nel formato dettagliato, richiedere spesso 3-10 pagine di testo.

Ci possono essere *violazioni ragionevoli* dei test: attività secondarie e piccoli passi possono diventare casi d'uso per evitare ripetizioni di testo, e in questi casi i tre test possono essere violati. Ad esempio, un "login utente" potrebbe non superare il test del capo ma essere abbastanza complesso da giustificare un'analisi accurata, come per una funzione "single sign-on".

I casi d'uso possono essere scritti a livelli diversi, ed è importante indicare il livello per ciascuno:

- *Livello di obiettivo utente*: il più interessante nell'analisi dei requisiti.

- *Livello di sotto-funzione*: utili per mettere a fattor comune parti di casi d'uso e/o per descrivere interazioni di dettaglio.

- *Livello di sommario*: un caso d'uso a questo livello comprende più casi d'uso a livello di obiettivo utente (utile per la loro identificazione e per comprendere il loro contesto).

I casi d'uso sono centrali nei processi iterativi, incluso UP (Unified Process). UP incoraggia lo sviluppo guidato dai casi d'uso: i requisiti sono registrati principalmente nel Modello dei casi d'uso, i casi d'uso hanno un ruolo importante nella pianificazione iterativa, la progettazione è guidata dalla loro realizzazione, influenzano l'organizzazione dei manuali d'uso, e i test funzionali e di sistema possono corrispondere agli scenari dei casi d'uso.

I punti di forza e usi dei casi d'uso includono: enfatizzare gli obiettivi degli utenti, decomporre la funzionalità del sistema in compiti discreti ("Divide et impera"), essere facili da capire per gli utenti, poter essere riutilizzati per la documentazione utente, consentire la derivazione diretta dei casi di test, ed essere indipendenti dalla tecnologia di implementazione.
Le associazioni tra attori e casi d'uso, rappresentate da una linea continua, indicano un canale di comunicazione. La direzione della linea può specificare chi dà inizio all'interazione, mentre l'assenza di direzione implica che entrambe le parti possono avviarla.
Esistono tre tipi principali di relazioni tra Casi d'Uso (Diagramma dei Casi d'Uso):

- **Include** (Inclusione): Questa relazione esiste tra un caso d'uso base e un caso d'uso che è incluso in quello base.
- **Extend** (Estensione): Questa relazione connette un caso d'uso esteso a un caso d'uso base. Aggiunge varianti a un caso d'uso base e viene inserita solo se la condizione di estensione è vera, in corrispondenza di specifici extension point.
- **Generalization** (Generalizzazione): Iaso d'uso genitore è una generalizzazione di un caso d'uso figlio. Il caso d'uso figlio viene eseguito se la condizione di generalizzazione è vera. I casi d'uso figli possono ereditare, aggiungere e sovrascrivere le funzioni del loro genitore. Può essere applicato anche per gli attori.

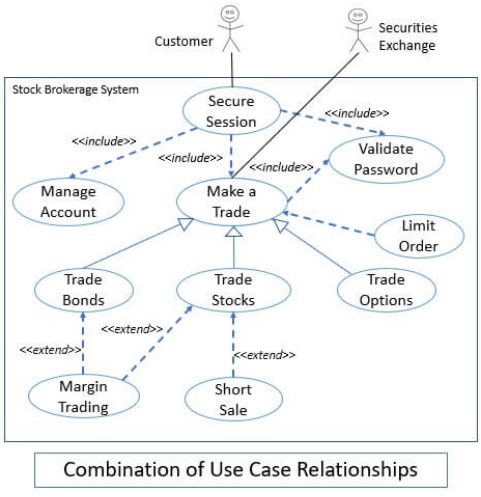


Figure 2.1: Include, Extend e Generalization in un Diagramma dei Casi d'Uso

Chapter 3

Analisi Orientata ad Oggetti

3.1 Modelli di Dominio

3.1.1 Seconda fase: Elaborazione

L'elaborazione rappresenta la fase iniziale delle iterazioni in un progetto tipico, durante la quale avvengono diverse attività chiave. In questo periodo, il nucleo dell'architettura software, spesso rischioso, viene programmato e verificato. Inoltre, la maggior parte dei requisiti viene scoperta e stabilizzata, e i rischi maggiori vengono mitigati o rientrano. Generalmente, l'elaborazione è composta da due o più iterazioni, ciascuna della durata di 2-6 settimane, e ogni iterazione è a tempo fisso (timeboxed).

In una sola frase, lo scopo dell'elaborazione è:
Costruire il nucleo dell'architettura, risolvere gli elementi ad alto rischio, definire la maggior parte dei requisiti e stimare il piano di lavoro e le risorse complessive.

Alcune idee e best practice fondamentali per la fase di elaborazione includono:

- Eseguire iterazioni guidate dal rischio, brevi e timeboxed.
- Iniziare precocemente la programmazione.
- Progettare, implementare e testare attivamente le parti principali e più rischiose dell'architettura.
- Effettuare test precocemente, frequentemente e in modo realistico.
- Adattarsi in base al feedback proveniente da test, utenti e sviluppatori.

Scrivere la maggior parte dei casi d'uso e degli altri requisiti nel dettaglio, prevedendo un workshop dei requisiti per ogni iterazione.
I requisiti e le iterazioni vanno organizzati in base a rischio, copertura e criticità:

1. **Rischio**: include sia la complessità tecnica sia altri fattori come l'incertezza dello sforzo o l'usabilità.
2. **Copertura**: indica che le iterazioni iniziali devono considerare tutte le parti principali del sistema.

3. **Criticità** (valore): si riferisce alle funzioni che il cliente considera di elevato valore di business.

Questi aspetti vengono ordinati spesso con tecniche di votazione, e le prime iterazioni implementano gli scenari con il voto maggiore. Questo approccio è parte di una pianificazione adattiva e iterativa.

3.1.2 Modello di Dominio

Un modello di dominio è il modello più importante dell'Analisi Orientata agli Oggetti (OOA). Esso descrive le classi concettuali e le relazioni tra di esse. Serve come fonte di ispirazione per le classi di progetto e di implementazione. Questo modello viene sviluppato in modo iterativo e incrementale ed è limitato dai requisiti dell'iterazione corrente.

È importante sottolineare che i modelli si influenzano reciprocamente.

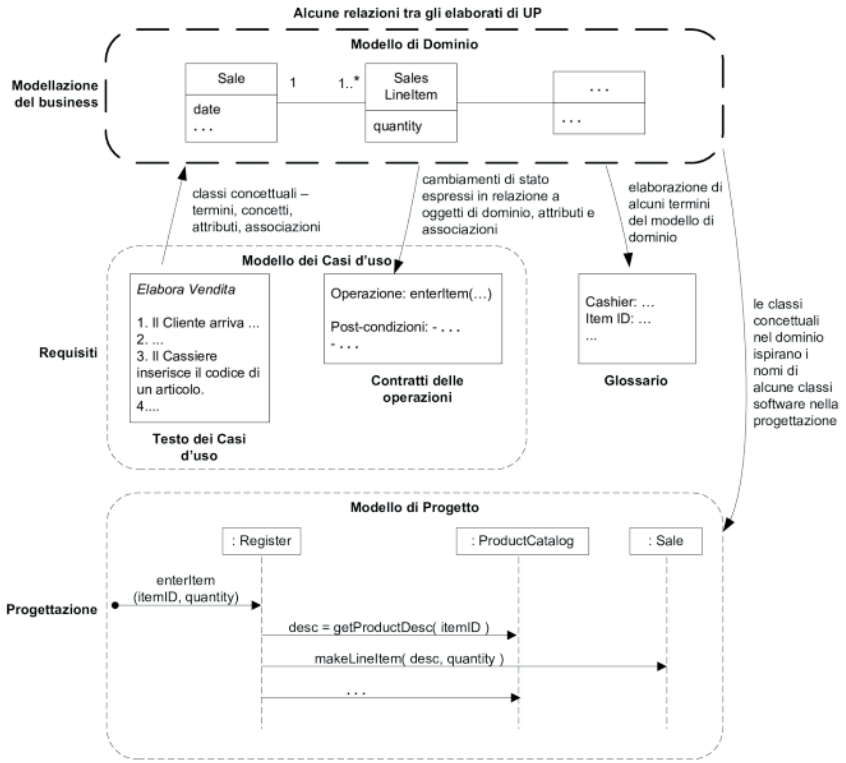


Figure 3.1: Relazioni tra gli elaborati di UP

Un modello di dominio è una rappresentazione visuale di classi concettuali o di oggetti del mondo reale e delle relazioni tra di essi, all'interno di un dominio di interesse. Applicando UML, un modello di dominio può essere realizzato come uno o più diagrammi delle classi che non definiscono operazioni, ma mostrano: classi concettuali o oggetti di dominio, associazioni tra classi concettuali e attributi di classi concettuali.
Il modello di dominio funge da "dizionario visuale". È un dizionario visuale delle astrazioni significative, della terminologia e del contenuto informativo del dominio di interesse. Esso mostra i vocaboli e la struttura del linguaggio proprio del dominio. Le classi concettuali in questo contesto sono astrazioni di oggetti reali.

È importante sottolineare che il modello di dominio ***non è una raffigurazione di oggetti software***. Non mostra elementi software, metodi o responsabilità.

Un modello di dominio si distingue da un modello dei dati:

- Un **modello dei dati** mostra i dati che devono essere memorizzati in modo persistente.
- Un **modello di dominio** descrive informazioni che devono essere gestite nel sistema in discussione e può contenere classi concettuali senza attributi o classi concettuali che hanno un ruolo puramente comportamentale e non solo informativo.

Una classe concettuale è un'idea, una cosa o un oggetto che può essere considerata in termini di:

- **Simbolo**: una parola o un'immagine usata per rappresentare la classe concettuale.
- **Intenzione**: la definizione (in linguaggio naturale) della classe concettuale.
- **Estensione**: l'insieme degli oggetti descritti dalla classe concettuale.

In UML, una **classe** è il descrittore per un insieme di oggetti che possiedono le stesse caratteristiche (attributi, operazioni, metodi, relazioni e comportamento). Una classe concettuale rappresenta un concetto del mondo reale e un insieme di oggetti che possiedono caratteristiche strutturali (attributi e relazioni) simili.

In UML, un'**associazione** è la relazione tra due o più classificatori che comporta connessioni tra le rispettive istanze. Un'associazione è una relazione tra classi (più precisamente, tra le istanze di queste classi) che indica una connessione significativa e interessante.

In UML, un **attributo** è la descrizione di una proprietà in una classe. Un attributo è un valore logico (ovvero un dato, una proprietà elementare) degli oggetti di una classe.

Perché creare un modello di dominio? Per comprendere il (dominio del) sistema da realizzare e il suo vocabolario (analisi). Serve come fonte di ispirazione per lo strato del dominio.

Come creare un modello di dominio? Rimanendo vincolati ai requisiti scelti per la progettazione nell'iterazione corrente, si devono affrontare i seguenti passi:

1. Trovare le classi concettuali.
2. Disegnarle come classi in un diagramma delle classi UML.
3. Aggiungere associazioni e attributi.

Per identificare le classi concettuali, si possono usare tre strategie:

1. Riusare o modificare modelli esistenti.
2. Utilizzare un elenco di categorie comuni.
3. Identificare nomi e locuzioni nominali.

3.1.3 Modello di Dominio in UML

In UML (Unified Modeling Language):

- Una **classe** è il descrittore per un insieme di oggetti che possiedono le stesse caratteristiche.
- Un'**associazione** è la relazione tra due o più classificatori che comporta connessioni tra le rispettive istanze. Un'associazione può anche essere multipla o riflessiva.
- Un **attributo** è la descrizione di una proprietà in una classe.
- Ciascuna estremità di un'associazione è chiamata **ruolo**.
- La **molteplicità** di un ruolo indica quante istanze di una classe possono essere associate a una istanza dell'altra classe.

Come tipi di relazioni, oltre all'associazione classica, abbiamo:

- **Aggregazione** (Legame Debole "Tutto-Parte"): Rappresenta una relazione in cui le parti possono esistere indipendentemente dall'aggregato, e più aggregati possono condividere la stessa parte. L'aggregato può essere incompleto senza alcune delle sue parti, ma la distruzione dell'aggregato non implica la distruzione delle parti. Un esempio è la relazione tra un Computer e una Stampante, dove la Stampante può esistere autonomamente.
- **Composizione** (Legame Forte "Tutto-Parte"): È una forma più forte di aggregazione in cui la vita della parte è vincolata a quella del composto. Ogni parte appartiene a un solo composito, e il composito è l'unico responsabile della creazione e distruzione delle sue parti. Se il composito viene distrutto, le sue parti devono essere distrutte o la loro responsabilità trasferita. È identificabile quando c'è un ovvio gruppo fisico o logico intero-parte, una dipendenza di creazione/cancellazione, e proprietà o operazioni dell'intero si propagano alle parti (es. posizione, movimento). Un esempio è la relazione tra un Mouse e un Pulsante, dove il Pulsante non ha esistenza propria senza il Mouse.

Il tipo degli attributi deve essere un tipo di dato. In UML, un tipo di dato è un insieme di valori in cui l'identità univoca non è significativa. Non è opportuno utilizzare gli attributi come chiavi esterne.

3.2 Diagramma di sequenza di sistema (SSD) e contratti

3.2.1 Diagramma di Sequenza di Sistema

Un **Diagramma di Sequenza di Sistema** (SSD) mostra gli eventi di input e output dei sistemi in discussione per un particolare corso di eventi all'interno di un caso d'uso. Include gli attori esterni che interagiscono direttamente con il sistema, il sistema stesso (considerato come una scatola nera), e gli eventi di sistema generati dagli attori.

Gli **eventi di sistema** sono eventi esterni di input, generati da un attore per interagire con il sistema, mentre le operazioni di sistema sono le trasformazioni o interrogazioni che il sistema deve eseguire per gestire tali eventi. UML non definisce esplicitamente "eventi di sistema" o "operazioni di sistema" o "diagrammi di sequenza di sistema", ma fornisce gli elementi generici "evento", "operazione" e "diagramma di sequenza". La qualifica "di sistema" enfatizza l'applicazione di questi elementi ai sistemi, considerati a scatola nera.

Gli SSD sono importanti perché il software deve essere progettato per gestire gli eventi di sistema. Un sistema software reagisce a tre tipi di eventi: eventi esterni da **attori** (umani o informatici), eventi **temporali**, e **guasti** o **eccezioni**.

Per descrivere le funzioni e il comportamento a "scatola nera" del sistema, senza spiegare come lo fa, si utilizzano modelli come i casi d'uso, i diagrammi di sequenza di sistema e i contratti delle operazioni di sistema. Gli elementi mostrati negli SSD (operazioni, parametri, valori restituiti) sono concisi, e una spiegazione più dettagliata può essere fornita nel Glossario del progetto.

È consigliabile disegnare un SSD per lo scenario principale di successo di ciascun caso d'uso, nonché per gli scenari alternativi più frequenti o complessi. Gli SSD vengono creati in modo evolutivo e iterativo, focalizzandosi solo sugli scenari dell'iterazione corrente. Fanno parte del Modello dei casi d'uso, visualizzando le interazioni implicate, e sono maggiormente creati durante la fase di elaborazione. Sebbene non siano un elaborato ufficiale di UP, la loro utilità è riconosciuta.

3.2.2 Contratti

Il comportamento del sistema è descritto dai casi d'uso e dai requisiti funzionali. I contratti delle operazioni di sistema offrono un modo più dettagliato per descrivere tale comportamento, utilizzando **pre-condizioni** e **post-condizioni** per indicare i cambiamenti agli oggetti in un modello di dominio.

Un'operazione di sistema è un'operazione pubblica del sistema la cui esecuzione è richiesta da un evento di sistema. L'esecuzione di un'operazione di sistema modifica lo stato del sistema, rappresentato dallo stato degli oggetti nel modello di dominio. L'insieme di tutte le operazioni di sistema costituisce l'interfaccia del sistema.

Un contratto relativo a un'operazione di sistema descrive il cambiamento dello stato degli oggetti del Modello di dominio causato dall'esecuzione di tale operazione. Le operazioni di sistema gestiscono gli eventi di sistema di input.

Le **post-condizioni** descrivono i cambiamenti nello stato degli oggetti del Modello di dominio. Non descrivono le azioni eseguite durante l'esecuzione, ma sono dichiarazioni sullo stato degli oggetti dopo l'esecuzione dell'operazione, e sono scritte al passato. I possibili cambiamenti di stato includono la creazione o cancellazione di un oggetto (istanza di una classe), il cambiamento di valore di un attributo, o la formazione o rottura di un collegamento (istanza di un'associazione). Le post-condizioni sono importanti perché descrivono i cambiamenti richiesti dall'esecuzione dell'operazione di sistema senza speci-

ficare come ottenerli. Lo spirito delle post-condizioni è paragonabile a fare una "foto" dello stato del sistema prima e dopo l'operazione, esprimendo i cambiamenti osservati. Le post-condizioni vanno espresse con un verbo al passato.

Le **pre-condizioni** descrivono ipotesi significative sullo stato del sistema prima dell'esecuzione dell'operazione. Offrono una descrizione sintetica dello stato di avanzamento del caso d'uso. Ad esempio, per l'operazione enterItem, una pre-condizione potrebbe essere "è in corso una vendita".

È fondamentale adottare un punto di vista concettuale e della conoscenza. I cambiamenti nello stato del sistema, causati dall'esecuzione di un'operazione, vanno espressi in termini di oggetti, collegamenti e attributi nel dominio di interesse che il sistema necessita di conoscere e ricordare. Ad esempio, "È stata creata un'istanza X" va interpretato come "è iniziata la conoscenza da parte del sistema dell'istanza X". Durante la creazione dei contratti, è normale che emerga la necessità di registrare nuove classi concettuali, attributi o associazioni nel modello di dominio. Nei metodi iterativi ed evolutivi, tutti gli elaborati di analisi e progettazione sono considerati parziali e imperfetti, ed evolvono in risposta a nuove scoperte.

L'utilità dei contratti risiede nel loro essere complementari ai casi d'uso. Sebbene i casi d'uso siano il principale modo per descrivere i requisiti funzionali e il comportamento del sistema in UP, i contratti non sono sempre necessari. Essi permettono di rappresentare situazioni dettagliate o complesse che non è opportuno descrivere nei casi d'uso.

Contratto CO2: enterItem

Operazione:	enterItem(itemID: ItemID, quantity: integer)
Riferimenti:	casi d'uso: Elabora Vendita
Pre-condizioni:	è in corso una vendita s
Post-condizioni:	- è stata creata un'istanza sli di SalesLineItem (*creazione di oggetto*).
	- sli è stata associata con la Sale (vendita) corrente s (*formazione di collegamento*).
	- sli è stata associata con una ProductDescription, in base alla corrispondenza con itemID (*formazione di collegamento*).
	- sli.quantity è diventata quantity (*modifica di attributo*).

Figure 3.2: Esempio di Contratto

Per creare e scrivere i contratti:

- Identificare le operazioni di sistema dagli SSD.
- Creare i contratti per le operazioni più complesse o non chiare dai casi d'uso.
- Per descrivere le post-condizioni, utilizzare categorie come creazione o cancellazione di oggetto, modifica di attributo, formazione o rottura di collegamento. Le post-condizioni descrivono cambiamenti di stato al passato e occorre ricordare di formare collegamenti necessari tra oggetti esistenti e quelli appena creati. Le pre-condizioni sono normalmente implicate dall'esecuzione di operazioni di sistema "precedenti" e sono utili per indicare quali oggetti sono noti al sistema.

In UML, un'**operazione** è la specifica di una trasformazione o interrogazione che un oggetto può essere chiamato ad eseguire. Un metodo è l'implementazione di un'operazione. Un'operazione ha una firma (signature) ed è associata a un insieme di vincoli (Constraint) classificati come pre-condizioni e post-condizioni che specificano la semantica dell'operazione. OCL (Object Constraint Language) è un linguaggio formale usato in UML per esprimere vincoli, e in teoria, pre-condizioni e post-condizioni possono essere espresse in OCL.

I contratti delle operazioni in UP possono essere usati per descrivere operazioni di sistema nel Modello dei casi d'uso. Non sono richiesti durante l'ideazione e vengono scritti principalmente durante l'elaborazione, solo per operazioni complesse.

Chapter 4

Progettazione orientata agli oggetti

4.1 Dai requisiti alla progettazione e architettura logica

L'analisi dei requisiti e l'analisi a oggetti si concentrano sul *"fare la cosa giusta"*. Il successivo lavoro di progettazione, invece, enfatizza il *"fare la cosa bene"*. Nello sviluppo iterativo, attività di analisi dei requisiti, analisi e progettazione orientata agli oggetti, implementazione e test vengono svolte in ogni iterazione. L'enfasi su progettazione e implementazione aumenta gradualmente.

È naturale e salutare scoprire e modificare alcuni requisiti durante le fasi iniziali di progettazione e implementazione. I metodi iterativi ed evolutivi abbracciano il cambiamento, ma cercano di provocarlo nelle iterazioni iniziali per avere obiettivi più stabili nelle iterazioni successive.

Il tempo richiesto per l'analisi vera e propria è realisticamente di alcune ore o giorni. Tuttavia, dall'inizio del progetto, potrebbero essere trascorse alcune settimane di preparazione per attività come la ricerca di risorse, la pianificazione e l'allestimento dell'ambiente.

4.2 Architettura Logica

Passando dall'analisi alla progettazione del software, si inizia a pensare su larga scala. La progettazione di un tipico sistema orientato agli oggetti si basa su diversi strati architetturali, come uno strato dell'interfaccia utente e uno strato della logica applicativa.

L'architettura logica di un sistema software è l'organizzazione su **larga scala** delle classi software in **package** (o namespace), **sottoinsiemi** e **strati**. Uno stile comune per l'architettura logica è l'**architettura a strati**.

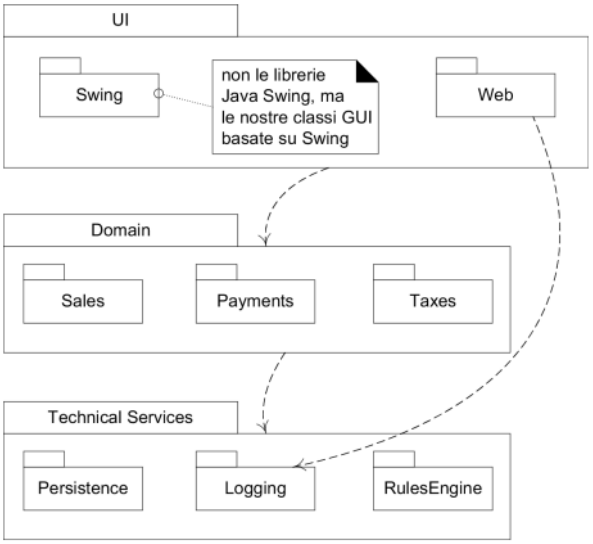


Figure 4.1: Caption

In un'architettura a strati, uno strato è un gruppo di classi, package o sottosistemi, con responsabilità coese rispetto a un aspetto importante del sistema. Generalmente, gli strati più alti utilizzano i servizi degli strati più bassi.
Gli strati di un'applicazione software tipicamente includono:

- **Presentazione**: che gestisce l'interfaccia utente.
- **Logica applicativa** o **strato del dominio**: che contiene gli elementi che rappresentano i concetti del dominio.
- **Servizi tecnici**.

Esistono due tipi di architettura a strati:

- In un'architettura a strati **stretta**, uno strato può richiamare solo i servizi dello strato immediatamente sottostante.
- In un'architettura a strati **rilassata**, uno strato più alto può richiamare i servizi di strati più bassi di diversi livelli.

Questo corso si concentra principalmente sullo strato della logica applicativa (o strato del dominio), con discussioni secondarie sugli altri strati. Le lezioni di progettazione orientata agli oggetti apprese in questo contesto sono applicabili anche a tutti gli altri strati o componenti.

L'architettura logica può essere illustrata mediante un diagramma dei package di UML. Un package UML può raggruppare classi, altri package o casi d'uso, ed è comune l'annidamento di package. Le dipendenze tra package sono mostrate con una dipendenza UML, e un package UML rappresenta un namespace che consente classi con lo stesso nome in package diversi (es. java::util::Date).

La progettazione degli strati implica l'organizzazione della struttura logica in strati separati con responsabilità distinte e correlate, garantendo una netta e coesa separazione

degli interessi. Gli strati inferiori offrono servizi generali e di basso livello, mentre quelli superiori sono più specifici per l'applicazione. Le collaborazioni e gli accoppiamenti vanno dagli strati più alti a quelli più bassi. L'obiettivo è suddividere il sistema in elementi software che possano essere sviluppati e modificati il più possibile indipendentemente l'uno dall'altro.

I vantaggi dell'uso a strati includono:

- Separazione degli interessi, che riduce l'accoppiamento e le dipendenze, migliora la coesione, aumenta la riusabilità e la chiarezza.
- Incapsulamento della complessità, che può essere decomposta.
- Possibilità di sostituire alcuni strati con nuove implementazioni.
- Funzioni riusabili negli strati più bassi.
- Possibilità di distribuire alcuni strati.
- Sviluppo facilitato per i team grazie alla segmentazione logica.

Le responsabilità coese e la separazione degli interessi implicano che in uno strato, ***le responsabilità degli oggetti devono essere fortemente coese e non mescolate con quelle di altri strati***. Ad esempio, gli oggetti dell'interfaccia utente (UI) devono concentrarsi sulle attività UI (creare finestre, catturare eventi), mentre gli oggetti dello strato della logica applicativa (o del dominio) devono concentrarsi sulla logica applicativa (calcolare totali o imposte). Gli oggetti UI non dovrebbero implementare logica applicativa, e le classi della logica applicativa non dovrebbero catturare eventi UI.

Per progettare la logica applicativa con gli oggetti, si creano oggetti software con nomi e informazioni simili al dominio del mondo reale e si assegnano loro responsabilità della logica applicativa. Un oggetto software di questo tipo è chiamato oggetto di dominio, che rappresenta una cosa nello spazio del dominio del problema e ha una logica applicativa o di business correlata (es. un oggetto Sale che calcola il suo totale). Questo approccio porta a uno strato della logica applicativa che può essere chiamato più precisamente strato del dominio dell'architettura, contenente oggetti di dominio per gestire la logica applicativa.

Il **principio di separazione Modello-Vista** stabilisce che gli oggetti di dominio (modello) non devono essere connessi o accoppiati direttamente agli oggetti UI (vista). Ad esempio, un oggetto di dominio Sale non dovrebbe avere un riferimento a un oggetto finestra JFrame di Java Swing. La logica applicativa non dovrebbe essere inclusa nei metodi di un oggetto dell'interfaccia utente; gli oggetti UI dovrebbero solo inizializzare gli elementi UI, ricevere eventi UI e delegare le richieste di logica applicativa a oggetti non UI. Un legittimo rilassamento di questo principio è il pattern Observer, dove gli oggetti del dominio inviano messaggi a oggetti della UI indirettamente, tramite un'interfaccia come PropertyListener, senza conoscere la classe UI concreta dell'oggetto.

Per quanto riguarda le definizioni:

1. **Livello** (*tier*): solitamente indica un nodo fisico di elaborazione.

2. **Strato** (*layer*): una sezione verticale dell'architettura.

3. **Partizione** (*partition*): una divisione orizzontale di sottosistemi di uno strato.

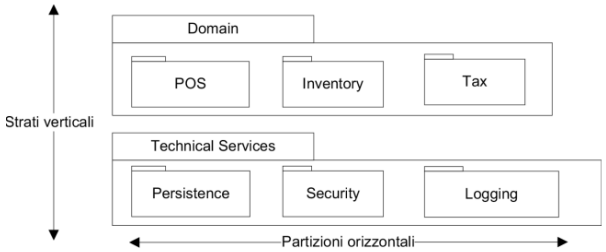


Figure 4.2: Architettura Logica

Gli sviluppatori possono progettare gli oggetti in diversi modi: codificando direttamente (spesso con TDD e refactoring), disegnando diagrammi UML su lavagna o con strumenti CASE prima di passare alla codifica, o affidandosi completamente alla generazione automatica da diagrammi.

L'Agile Modeling e il disegno leggero di UML mirano a ridurre i costi aggiuntivi del disegno, focalizzandosi sulla modellazione per comprendere e comunicare, anziché per documentare. Le pratiche includono la modellazione collaborativa e la creazione parallela di diversi modelli (es. diagrammi di interazione e classi per brevi sessioni di 5 minuti). Gli strumenti CASE per UML possono essere complementari alla modellazione alla parete, ed è consigliabile sceglierne uno che si integri con IDE diffusi (come Eclipse) e che sia in grado di eseguire il reverse-engineering non solo per i diagrammi delle classi, ma anche per i diagrammi di interazione.

Per un'iterazione di 3 settimane, si dovrebbero dedicare *alcune ore* o al *massimo un giorno* al disegno UML, per poi passare alla codifica per il resto dell'iterazione. Sessioni di disegno più brevi possono essere previste nel corso dell'iterazione, e prima di ogni sessione successiva, è utile eseguire il reverse-engineering e fare riferimento ai diagrammi aggiornati.

Esistono due tipi di modelli per gli oggetti: **statici** e **dinamici**, da creare in parallelo.

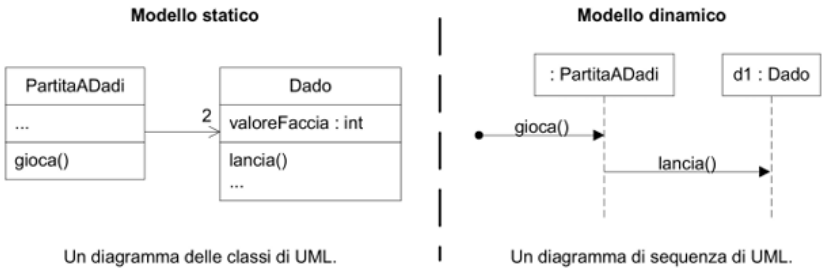


Figure 4.3: Modelli statici e dinamici in parallelo

I **modelli dinamici** (diagrammi di sequenza e comunicazione) sono i più importanti e difficili da creare, e per essi si applicano la progettazione guidata dalle responsabilità e

i principi GRASP. I **modelli statici** (diagrammi delle classi) sono utili come sintesi e base per la struttura del codice.

È cruciale dedicare tempo sufficiente alla creazione dei diagrammi di iterazione. L'importanza della capacità di progettazione a oggetti supera la mera conoscenza della notazione UML. La notazione è utile, ma l'aspetto fondamentale è sapere "come pensare e progettare a oggetti" e applicare le best practice e i design pattern della progettazione a oggetti. Il disegno con UML è una conseguenza delle decisioni di progetto, e la progettazione a oggetti richiede la conoscenza di principi di assegnazione di responsabilità e design pattern. Altre tecniche di progettazione includono le schede CRC (Class, Responsibility, Collaboration).

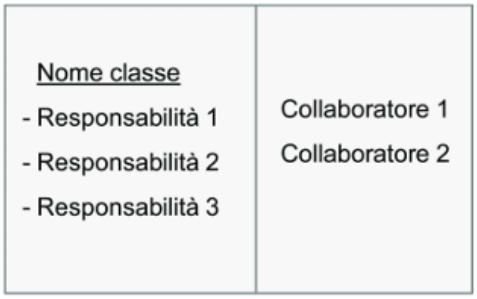


Figure 4.4: Scheda CRC

4.3 Diagrammi di interazione

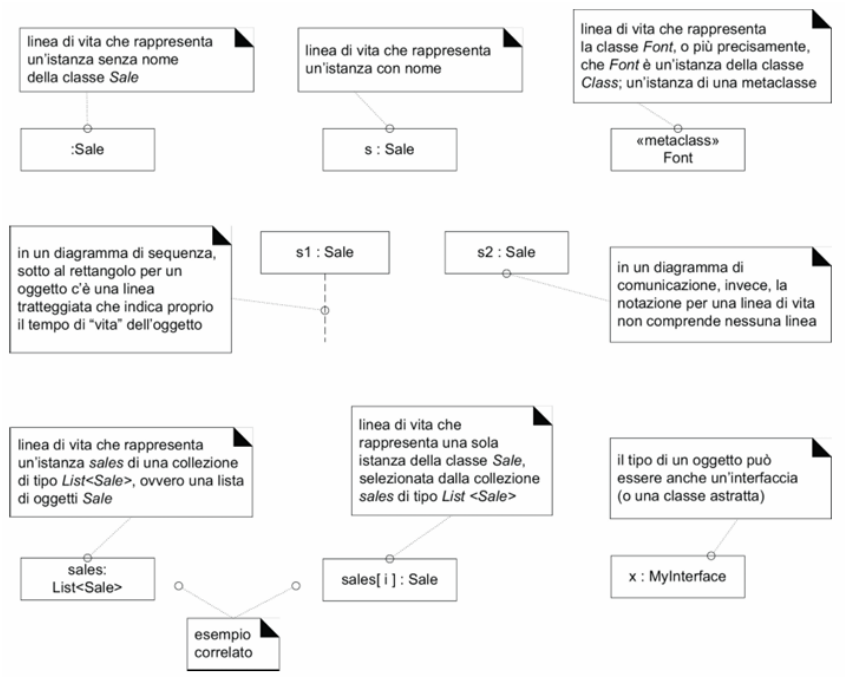
Un **oggetto** è un'unità riusabile che incapsula dati e funzioni. Ogni oggetto possiede un'identità (un identificatore unico), uno stato (determinato dai valori dei suoi dati in un dato istante) e un comportamento (l'insieme delle operazioni che può eseguire).

L'**incapsulamento** (o data hiding) è un principio fondamentale: i dati sono nascosti all'interno dell'oggetto e l'unico modo per accedervi è *tramite le operazioni che l'oggetto espone*. Questo promuove la coesione e riduce la dipendenza tra le parti del sistema.

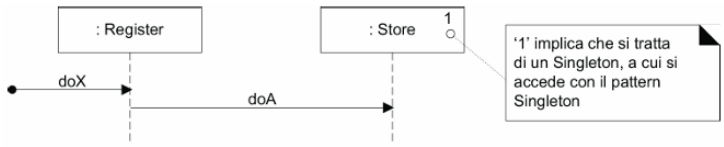
Gli oggetti collaborano tra loro scambiandosi **messaggi** per eseguire le funzioni del sistema. Un messaggio causa l'invocazione di un'operazione da parte dell'oggetto destinatario.

I diagrammi di interazione in UML sono utilizzati per modellare il comportamento dinamico degli oggetti, illustrando come questi interagiscono attraverso lo scambio di messaggi. Catturano un'interazione tramite:

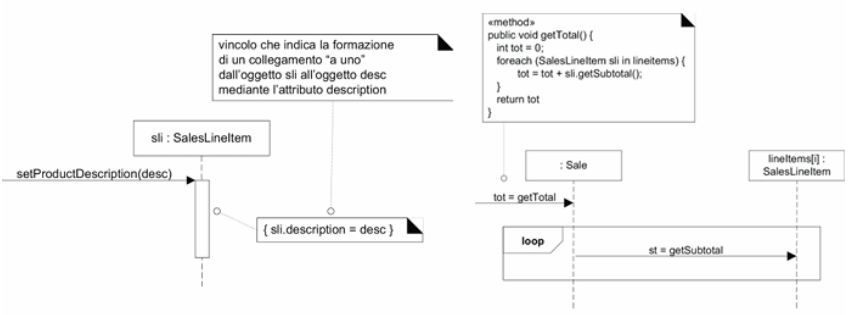
- **Linee di vita**: Rappresentano un singolo partecipante (un'istanza di un classificatore) all'interazione. Hanno un nome, un tipo e un selettore (una condizione booleana per selezionare un'istanza specifica).



- **Messaggi**: Rappresentano la comunicazione tra due linee di vita.
- **Oggetto Singleton**



- **Note**



Esistono diversi tipi di diagrammi di interazione, ognuno con un focus specifico: n

- **Diagrammi di sequenza**: Enfatizzano la sequenza temporale degli scambi di messaggi, mostrando interazioni ordinate nel tempo. Non mostrano esplicitamente le relazioni tra gli oggetti, ma queste possono essere dedotte dagli scambi di messaggi. Sono disegnati in un formato "a steccato".
- **Diagrammi di comunicazione**: Enfatizzano le relazioni strutturali tra gli oggetti, mostrando come le linee di vita sono collegate. Sono disegnati in un formato "a grafo" o "a rete".

- **Diagrammi di interazione generale**: Illustrano come comportamenti complessi sono realizzati da interazioni più semplici.
- **Diagrammi di temporizzazione**: Enfatizzano gli aspetti in tempo reale di un'interazione.

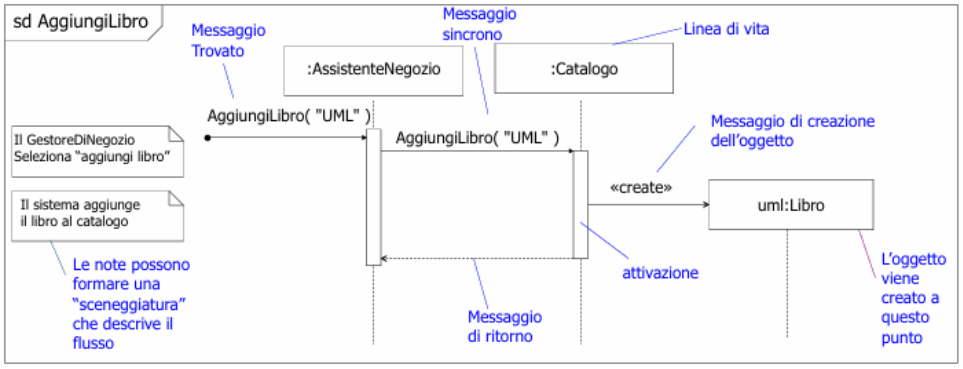
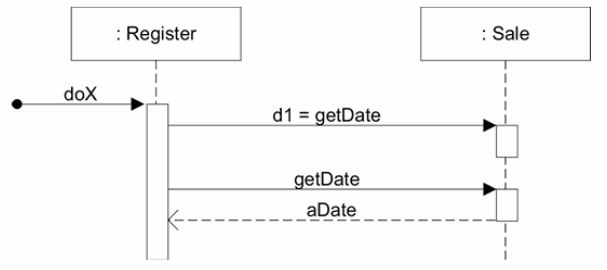


Figure 4.5: Esempio dei Diagrammi di Sequenza

Nei diagrammi di interazione, si usano diverse notazioni:

- **Attivazioni**: Indicano quando una linea di vita ha il focus di controllo.



- **Creazione e distruzione di istanze**: La creazione di un oggetto è indicata, così come la sua distruzione (terminando la linea di vita con una grossa croce).

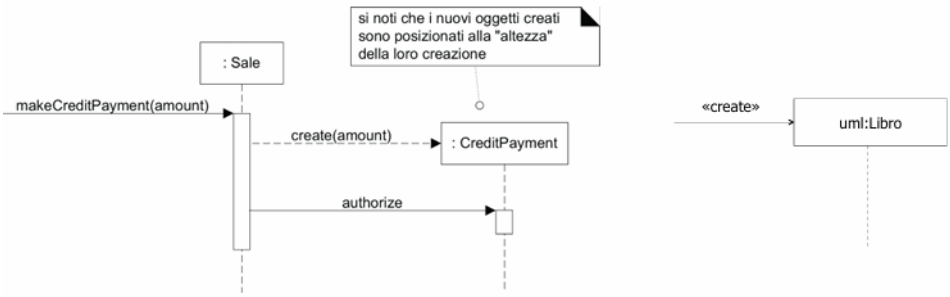


Figure 4.6: Creazione di istanze

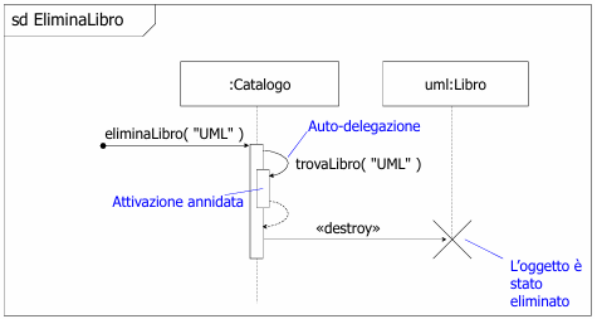


Figure 4.7: Distruzione di istanze e auto-delega

- **Auto-delegazione**: Quando una linea di vita invia un messaggio a se stessa, generando un'attivazione annidata.
- **Invarianti di Stato**: Descrivono le condizioni o situazioni in cui un oggetto si trova durante la sua vita.

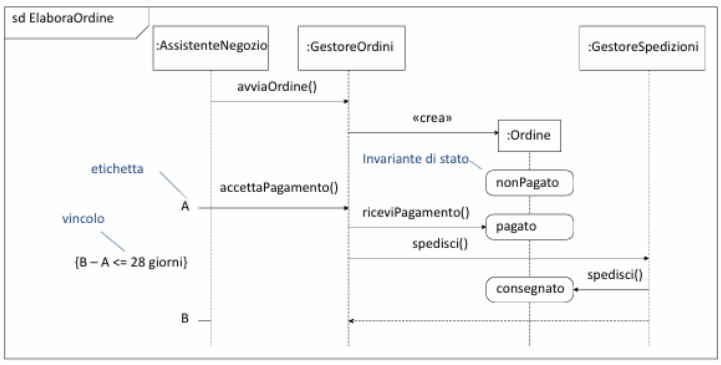


Figure 4.8: Invarianti di Stato

I **frammenti combinati** (combined fragments) dividono i diagrammi di sequenza in aree per rappresentare comportamenti complessi. Hanno uno o più operandi e un operatore che ne determina l'esecuzione, spesso con condizioni di guardia.
Gli operatori più comuni includono:

Operatore	Descrizione
opt	Un singolo operando viene eseguito se la condizione è vera (come un if...then).
alt	Viene eseguito l'operando la cui condizione è vera (come un select... case).
loop	Esegue un operando più volte, con sintassi specifica per min/max ripetizioni e condizioni.
break	Se la condizione di guardia è vera, l'operando viene eseguito e il resto dell'interazione viene interrotto.
ref	Fa riferimento a un'altra interazione, utile per riutilizzare schemi di interazione.
Altri operatori	Gestiscono il parallelismo (par, seq, strict), le interazioni che non devono accadere (neg) e le regioni critiche (critical).

Idiomi più comuni:

- loop *→ Continuare il ciclo sempre
- loop n→ Ripetere n volte
- loop [espressioneBooleana] → Ripetere fino a che espressioneBooleana è vera
- loop 1,*espressioneBooleana → Eseguire 1 volta e poi fino a che espressione N è vera
- loop [collezione] → Foreach di collezione
- loop [classe] → Foreach di classe

Spesso non si presta sufficiente attenzione alla modellazione dinamica con i diagrammi di interazione, privilegiando i diagrammi delle classi (che modellano gli aspetti statici). È fondamentale modellare in parallelo aspetti statici e dinamici del sistema per una progettazione completa ed efficace.

4.4 Diagrammi delle Classi

I diagrammi delle classi sono il cuore della modellazione strutturale in UML. Descrivono le classi, i loro componenti interni e le relazioni statiche tra di esse.
Componenti di una Classe in UML Una **classe** è rappresentata da un rettangolo diviso in tre sezioni:

- **Nome della Classe**: La sezione superiore, in grassetto e centrata.
- **Attributi**: La sezione centrale. Ogni attributo modella una proprietà locale della classe ed è caratterizzato da un nome e dal tipo dei valori associati. Ha:
  - **Visibilità**: Essendo in un ambiente con incapsulamento, solitamente sono private. Se non specificata, la visibilità è privata di default.
    - + (public): Accessibile da qualsiasi classe.
    - - (private): Accessibile solo dalle operazioni della classe stessa.
    - # (protected): Accessibile dalla classe e dalle sue sottoclassi.
    - (package/default): Accessibile solo all'interno dello stesso package.
  - **Nome**: Obbligatorio.
  - **Tipo**: Il tipo di dati associato all'attributo.
  - **Molteplicità**: Se non indicata, il valore predefinito è uno. Se un attributo B di tipo T di una classe C ha molteplicità x..y, allora B associa ad ogni istanza di C al minimo x e al massimo y valori di tipo T.
  - **Valore Iniziale**: Opzionale. Oltre agli attributi di classe, esistono anche attributi di associazioni e collegamenti.
- **Operazioni** (*Metodi*): La sezione inferiore. Un'operazione di UML è una dichiarazione di funzionalità con un nome, parametri, tipo di ritorno, elenco di eccezioni e opzionalmente vincoli di precondizioni e postcondizioni. La sua sintassi è: visibility name(parameter-list) : return-type property-string.

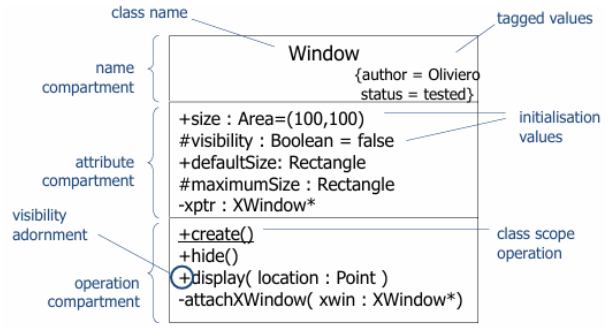


Figure 4.9: Esempio di classe in UML

Un metodo è l'implementazione di un'operazione. L'operazione create corrisponde a un costruttore. Le operazioni get e set non vengono spesso mostrate per il basso rapporto disturbo-valore. Un'operazione polimorfica ha molte implementazioni.

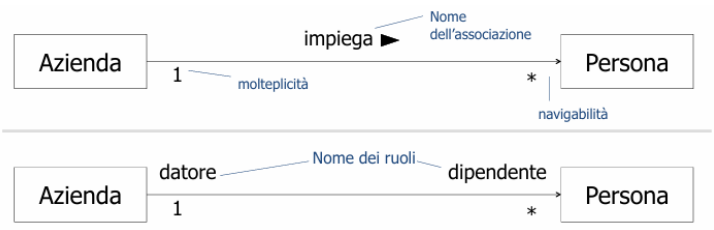
Le **relazioni** descrivono come le classi interagiscono e sono connesse.

Definizione: Associazione

Associazione: Modella una relazione matematica tra l'insieme delle istanze di due o più classi.

Può specificare un **verso** per il nome dell'associazione (una freccia sul nome). È possibile assegnare **due nomi** con i relativi versi alla stessa associazione. Si può aggiungere un'informazione che specifica il **ruolo** che una classe gioca nell'associazione (testo sul lato della classe, vicino all'estremità della linea di associazione).Può avvenire tra più classi (associazione n-aria) (Figura 4.10).

- La **navigabilità** (freccia sull'estremità della linea) indica che è possibile spostarsi da un oggetto della classe origine a uno o più oggetti della classe destinazione.
- La **molteplicità** (vedi sopra per gli attributi) limita il numero di oggetti di una classe che possono partecipare in una relazione in un dato istante.



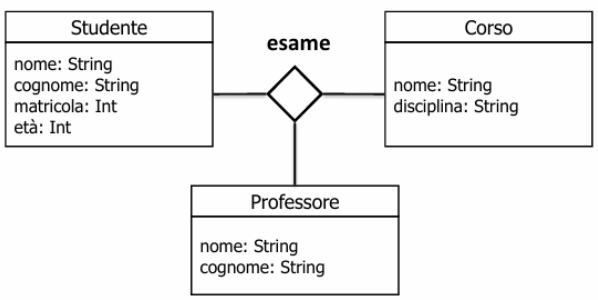
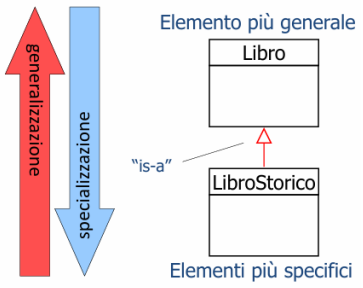


Figure 4.10: Associazione n-aria

Ci sono casi in cui l'associazione insiste più volte sulla stessa classe, ma il ruolo non è significativo, in particolare quando l'associazione rappresenta una relazione simmetrica.

Generalizzazione (*Ereditarietà*): La relazione "is-a". Ogni istanza di ciascuna sottoclasse è anche istanza della superclasse. Ogni proprietà della superclasse è anche una proprietà della sottoclasse, e non si riporta esplicitamente nel diagramma.



La superclasse può generalizzare diverse sottoclassi rispetto a un unico criterio; in questo caso, la freccia è unita. Si può generalizzare anche sulle associazioni.

Vincoli di generalizzazione:

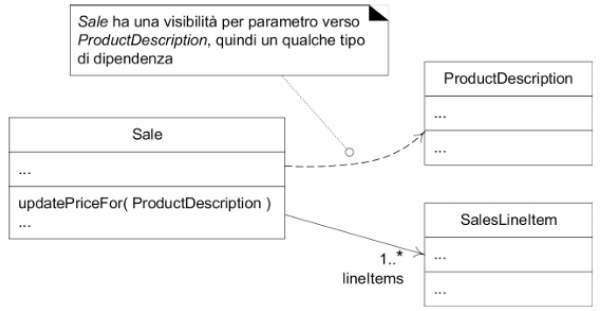
- {*disjoint*}: I classificatori specifici non hanno istanze comuni.
- {*overlapping*}: Le istanze possono appartenere a più sottoclassi contemporaneamente.
- {*complete*}: L'unione delle istanze delle sottoclassi è uguale all'insieme delle istanze della superclasse (non ci sono altre sottoclassi possibili).
- {*incomplete*}: Esistono altre sottoclassi non rappresentate.

Specializzazione:

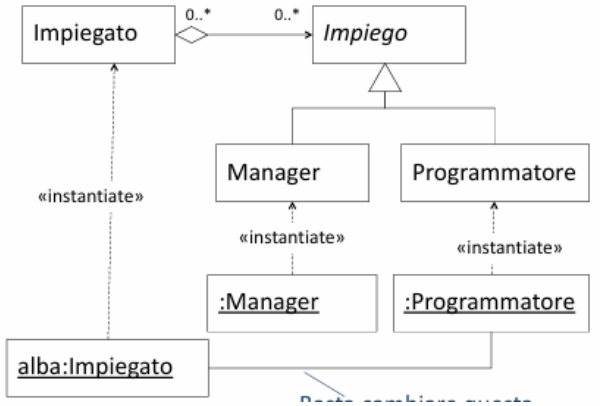
- **Specializzazione attributo**: Una sottoclasse può specializzare le proprietà (anche la molteplicità) ereditate dalla superclasse.
- **Specializzazione associazione**: Indicata con subsets.
- **Specializzazione operazioni**: Un'operazione si specializza specializzando i parametri e/o il tipo di ritorno.

Una classe con una o più operazioni astratte non può essere istanziata (si scrive il nome della classe in corsivo).

Dipendenza: Una relazione più debole (freccia tratteggiata) che indica che un elemento cliente è a conoscenza di un elemento fornitore, o un elemento è accoppiato o dipende da un altro. Un cambiamento nel fornitore può influenzare il cliente.



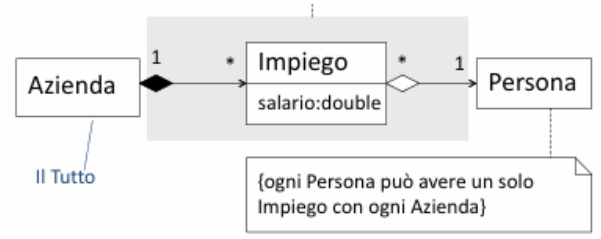
Aggregazione: Una associazione che rappresenta una relazione "intero-parte" (rombo vuoto sul lato dell'intero). La parte può esistere indipendentemente dal tutto.



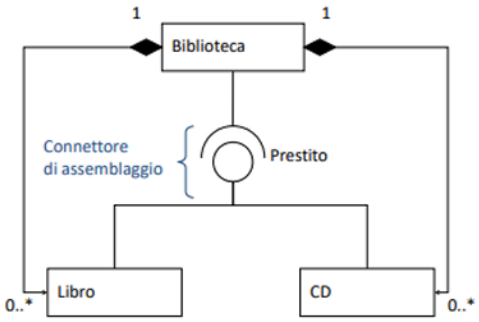
Composizione: Una forma forte di aggregazione (rombo pieno sul lato dell'intero) dove:

- Una parte appartiene a un solo composto alla volta.
- Ciascuna parte appartiene sempre a un composto.

Il composto è responsabile della creazione e cancellazione delle sue parti (la distruzione del composto implica la distruzione delle parti).



Un'**interfaccia** è un insieme di funzionalità pubbliche identificate da un nome. Separa le specifiche di una funzionalità dall'implementazione. Rappresentata con un cerchio (interfaccia Prestito).



Si possono connettere interfacce richieste (semicerchio) e interfacce fornite (cerchio) utilizzando il connettore di assemblaggio (pallino che si inserisce nel semicerchio).

I profili si usano per personalizzare UML per un uso specifico. Un profilo è un insieme di:

- **Stereotipi**: Definiscono un nuovo elemento di modellazione UML basandosi su un esistente.
- **Tagged Values** (Tag): Permettono di estendere la definizione di un elemento tramite l'aggiunta di nuove informazioni specifiche (es. ordered). Sono scritti tra {}.
- **Vincoli**: Estendono la semantica di un elemento consentendo di aggiungere nuove regole (es. ordered per gli attributi).

Chapter 5

Diagrammi di Attività e Diagrammi di Macchina a Stati

5.1 Macchine a Stati

Le macchine a stati possono essere utilizzate per modellare il comportamento dinamico di classificatori quali classi, casi d'uso, sottosistemi e interi sistemi. Esistono sempre nel contesto di un particolare classificatore che:

1. Risponde a eventi esterni;
2. Ha un ciclo di vita definito, rappresentabile come una sequenza di stati, transizioni ed eventi;
3. Può avere un comportamento corrente che dipende da comportamenti precedenti.
4. Tali diagrammi sono utili soprattutto per rappresentare il comportamento dinamico degli oggetti nel tempo.

È utile distinguere tra:

- Oggetti indipendenti dallo stato, che rispondono sempre nello stesso modo a un determinato evento;
- Oggetti dipendenti dallo stato, che rispondono in modo diverso a seconda dello stato in cui si trovano.

Le macchine a stati sono utili per:

- Modellare il comportamento di un oggetto reattivo complesso in risposta agli eventi;
- Modellare le sequenze valide delle operazioni (specifiche di protocollo o di linguaggio).

Ogni macchina a stati ha:

1. Uno **stato iniziale**, che rappresenta l'inizio della sequenza;
2. Uno **stato finale**, che chiude la sequenza (a meno di cicli infiniti).

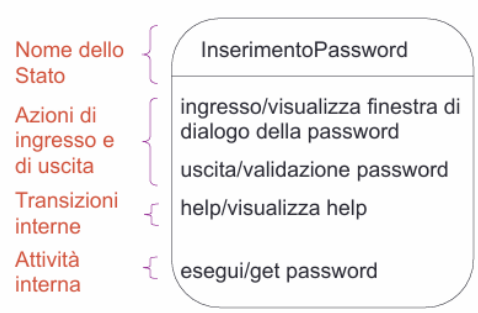
Transizioni tra stati sono causate da eventi, e possono essere condizionate da guardie (condizioni booleane) e accompagnate da azioni.

Stati - Uno **stato** rappresenta una condizione della vita di un oggetto durante la quale:

- L'oggetto soddisfa una condizione;
- Esegue un'attività;
- Attende un evento.

È determinato dai valori degli attributi, relazioni con altri oggetti, e attività in corso. Ogni stato può avere:

- Azioni di ingresso/uscita
- Attività interne
- Transizioni interne



Transizioni - Le transizioni collegano gli stati in risposta agli eventi. Possono essere:

- *Semplici*;
- *Condizionate* (con guardie);
- *Con azioni associate.*
- *Pseudo-stati*: Giunzioni e Selezioni
  - Giunzione: fusione o ramificazione di più transizioni (Figura 5.1).
  - Selezione: diramazione condizionata (es. pagamento parziale, completo, ecc.) (Figura 5.2).

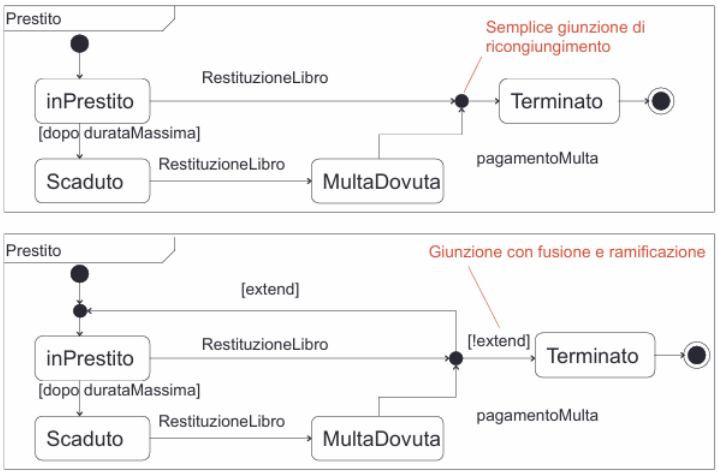


Figure 5.1: Pseudo stato Giunzione

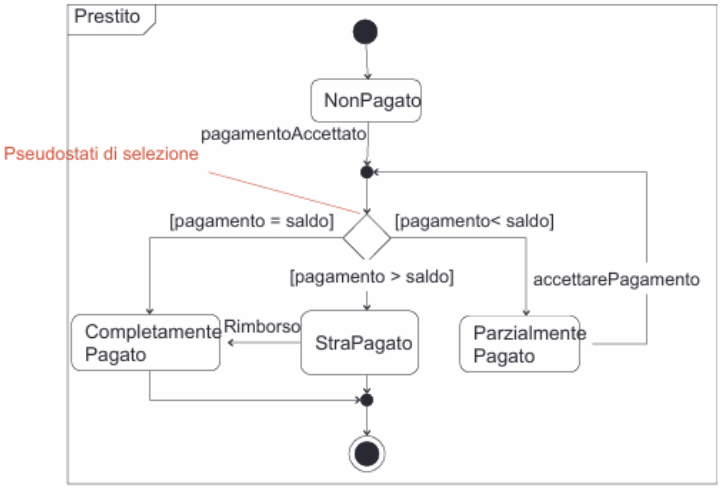


Figure 5.2: Pseudo stato Selezione

Eventi - Gli **eventi** attivano le transizioni. Possono essere di diversi tipi:

- *Evento di chiamata*: richiama un'operazione;
- *Evento di segnale*: pacchetto asincrono tra oggetti;
- *Evento di variazione*: attivato da una condizione booleana che passa da falso a vero;
- *Evento temporale*: attivato dal trascorrere del tempo (es. dopo(3 mesi)).

Stati Compositi - Gli **stati compositi** contengono una o più sotto-macchine. Possono essere:

- *Semplici* (una sola regione);
- *Ortogonali* (più regioni eseguite in parallelo) (Figura 5.3).

Ogni regione può contenere i propri stati e transizioni. L'uscita può essere sincronizzata (tutte le regioni devono terminare) o asincrona.

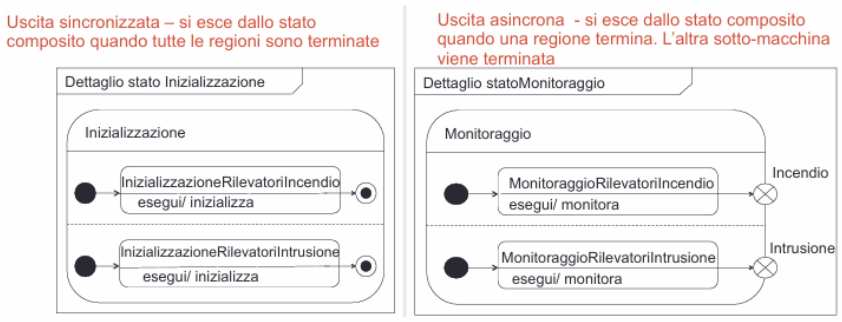


Figure 5.3: Stati Compositi Ortogonali

Stati della Sotto-Macchina - Per evitare diagrammi troppo complessi, è possibile riferirsi a una sotto-macchina come a un unico stato. Il suo contenuto può essere incluso solo quando necessario.

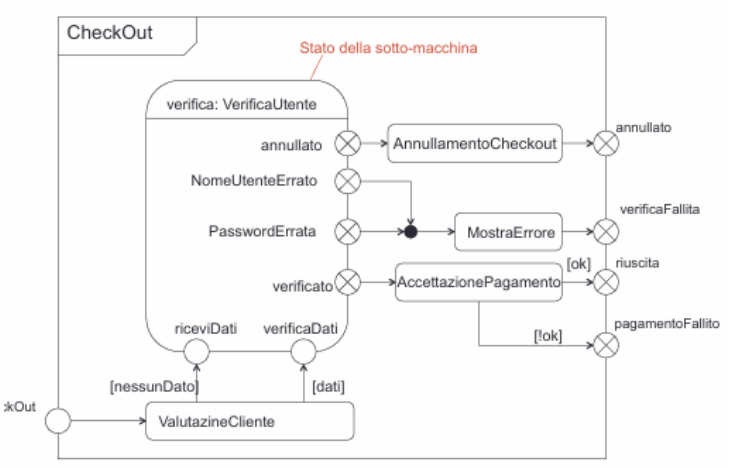


Figure 5.4: Stati della Sotto-Macchina

Comunicazione tra Sotto-Macchine - La comunicazione tra sotto-macchine può avvenire tramite:

- *Flag* (variabili condivise tra regioni) (Figura 5.5);
- *Stati di sync*, che fungono da code per sincronizzare esecuzioni parallele (Figura 5.6).

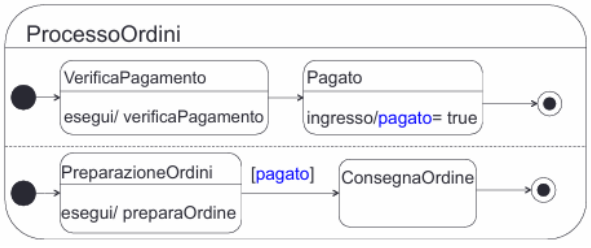


Figure 5.5: Comunicazione tra sotto-macchine

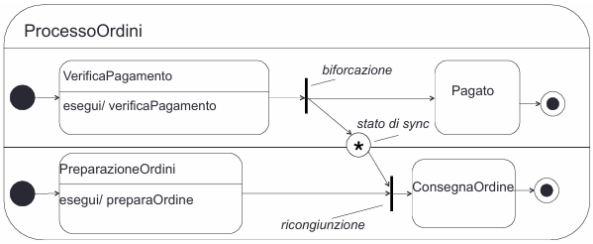


Figure 5.6: Comunicazione tramite stato di sync

Stati con Memoria:

- *Memoria semplice* (H): ricorda l'ultimo sottostato attivo al rientro;
- *Memoria multilivello* (H*): ricorda anche i sottostati annidati.

Nell'Unified Process (UP), non esiste un modello dedicato alle macchine a stati, ma ogni elemento (classe, caso d'uso, ecc.) può avere associata una macchina per descrivere il proprio comportamento dinamico.

5.2 Diagrammi di Attività

I diagrammi di attività spesso vengono chiamati "diagrammi di flusso OO". Consentono di modellare un processo come un'attività costituita da un insieme di nodi connessi da archi.
Le attività vengono di solito associate a:

- Casi d'uso
- Classi
- Interfacce componenti
- Collaborazioni
- Operazioni
- Processi di business

Le attività sono reti di nodi connessi ad archi. Esistono tre categorie di nodi:

- *Nodi azione*: rappresentano unità discrete di lavoro atomiche all'interno dell'attività.
- *Nodi controllo*: controllano il flusso attraverso l'attività.
- *Nodi oggetto*: rappresentano oggetti usati nell'attività.

Esistono due categorie di archi:

- *Flussi di controllo*: rappresentano il flusso di controllo attraverso le attività.
- *Flussi di oggetti*: rappresentano il flusso di oggetti attraverso l'attività.

Le attività sono reti di nodi connessi da archi. Il flusso di controllo è un tipo di arco. Le attività spesso iniziano con un singolo nodo di controllo, il nodo iniziale. Le attività possono avere pre e post condizioni. Quando un nodo azione finisce, esso emette un token che potrebbe attraversare un arco per dare inizio alla prossima azione.

Il token game: un token è un oggetto, alcuni dati o flusso di controllo. Descrive il flusso di token attorno a una rete di nodi e archi. I token si spostano da un nodo sorgente a un nodo destinazione attraverso un arco. Il nodo sorgente, l'arco e il nodo destinazione possono avere vincoli che controllano il flusso dei token. Il movimento del token può verificarsi solo quando tutte le condizioni sono soddisfatte.Un nodo inizia l'esecuzione quando ci sono token su tutti i suoi archi di entrata.
L'esecuzione dei nodi azione avviene quando:

- Esiste un token simultaneamente su ciascun arco entrante
- I token in ingresso soddisfano tutte le precondizioni locali del nodo azione

I **nodi azione**:

1. Eseguono un AND logico sui loro token di entrata
2. Eseguono una fork implicita su tutti i suoi archi uscenti quando l'esecuzione è terminata

Nodi di controllo

- Nodo iniziale: inizio del flusso
- Nodo finale attività: termina tutta l'attività
- Nodo finale del flusso: termina un flusso, altri proseguono
- Nodo biforcazione: divide il flusso in più flussi concorrenti
- Nodo ricongiunzione: sincronizza più flussi
- Nodo decisione: ha condizioni di guardia
- Nodo fusione: unisce flussi alternativi

Nodi Oggetto

- Indicano che sono disponibili istanze di un classificatore.
- Possono contenere più token (fino a un limite superiore)
- Ordinamento: FIFO, LIFO, criterio di selezione

I nodi oggetto possono essere parametri di input e output per le attività.
Un **Pin** è un nodo oggetto che rappresenta un input in un'azione o output da un'azione.
Un diagramma di attività può modellare casi d'uso come "Elabora Vendita", visualizzando l'intero processo con nodi, archi, oggetti, decisioni e flussi.

Chapter 6

GRASP

Progettazione guidata dalle responsabilità (Responsibility-Driven Design, RDD) La progettazione guidata dalle responsabilità è un approccio che incoraggia a pensare in termini di ruoli, responsabilità e collaborazioni tra oggetti. Piuttosto che focalizzarsi inizialmente sulla struttura statica del software, questo metodo parte dalle azioni e dalle conoscenze che gli oggetti devono gestire per soddisfare i requisiti del sistema.

Cosa si intende per responsabilità?
Nel linguaggio UML, una responsabilità è definita come un "contratto o obbligo di un classificatore". In altre parole, rappresenta ciò che un oggetto fa o ciò che conosce nel contesto dell'applicazione.
Le responsabilità si suddividono in due categorie:

1. Responsabilità di fare:
   Un oggetto può essere responsabile di:

   - Eseguire operazioni internamente, come un calcolo o un'azione su dati propri.
   - Delegare operazioni ad altri oggetti, chiedendo loro di eseguire azioni specifiche.
   - Controllare e coordinare le attività di altri oggetti, fungendo da "regista" della collaborazione.

2. Responsabilità di conoscere:
   Un oggetto può essere responsabile di:

   - Conoscere i propri dati privati, ovvero le informazioni incapsulate al suo interno.
   - Conoscere altri oggetti con cui ha una relazione significativa.
   - Conoscere o calcolare informazioni che può derivare dai dati in suo possesso.

Le responsabilità sono quindi implementate attraverso oggetti e metodi che agiscono da soli o collaborano tra loro per soddisfare le richieste del sistema.

6.1 Il flusso di lavoro nella progettazione RDD

La progettazione guidata dalle responsabilità segue tre passaggi fondamentali:

1. Identificare le responsabilità da gestire all'interno del sistema, considerando un compito alla volta.

2. Assegnare ciascuna responsabilità a un oggetto software appropriato. Questo oggetto può già esistere oppure può essere definito appositamente.

3. Valutare come l'oggetto assegnatario soddisferà la responsabilità: potrà farlo autonomamente o collaborando con altri oggetti.

6.2 I pattern GRASP

Definizione: Pattern

Un pattern è una soluzione tipica e collaudata per un problema ricorrente, accompagnata da un nome che ne facilita la discussione e il riutilizzo. I pattern non sono semplici frammenti di codice, ma rappresentano idee progettuali consolidate, nate dall'esperienza e riconosciute come valide in molteplici contesti.

I **GRASP patterns** (General Responsibility Assignment Software Patterns) forniscono una raccolta di linee guida per assegnare correttamente le responsabilità agli oggetti.

I principali pattern GRASP sono:

- **Creator**: Indica quale oggetto dovrebbe essere responsabile della creazione di altri oggetti.

- **Information Expert**: Assegna la responsabilità a chi ha le informazioni necessarie per svolgerla.

- **Low Coupling**: Favorisce una bassa dipendenza tra classi, migliorando la manutenibilità del sistema.

- **Controller**: Assegna a un oggetto di controllo la gestione delle operazioni di sistema.

- **High Cohesion**: Incoraggia la coerenza e il focus delle classi, evitando oggetti con troppe responsabilità eterogenee.

Questi pattern costituiscono la base per una progettazione a oggetti robusta, manutenibile e ben strutturata, facilitando decisioni architetturali coerenti nel tempo.

6.2.1 Creator

Problema: Chi dovrebbe essere responsabile della creazione di una nuova istanza di una classe?
Soluzione: Assegna la responsabilità della creazione dell'oggetto A a una classe B se:

- B contiene istanze di A
- B registra istanze di A
- B usa frequentemente oggetti A
- B ha i dati necessari per inizializzare A

In pratica, se B è concettualmente vicino a A (es. lo contiene, lo aggrega, lo conosce), allora è un buon candidato.

***Esempio**: Nel Sistema POS il Creator è Sale in quanto contiene molti oggetti SalesLineItem*

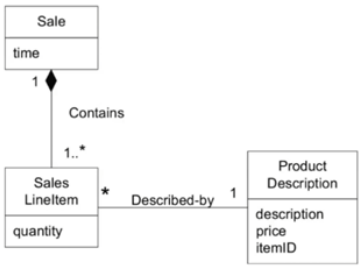


Figure 6.1: Esempio di applicazione del Pattern Creator

Discussione: Questo pattern aiuta a mantenere un accoppiamento basso, poiché l'oggetto creato è già conosciuto dal creatore. È particolarmente utile nei contesti dove un oggetto "padre" compone o gestisce oggetti "figlio".

Controindicazioni: Se la creazione dell'oggetto è complessa o varia, può essere più adatto un pattern creazionale come Abstract Factory (GoF).

Pattern correlati: Low Coupling, Whole-Part, Design Pattern Creazionali (Factory Method, Builder)

6.2.2 Information Expert

Problema: Come decidere a quale oggetto assegnare una responsabilità? **Soluzione**: Assegna la responsabilità all'oggetto che possiede le informazioni necessarie per svolgere il compito. Se un oggetto conosce i dati utili per realizzare un comportamento, è il candidato naturale.

***Esempio**: Nel Sistema POS l'Information Expert è Sale in quanto ha le conoscenze per calcolare il totale di una vendita. Dunque:*

1. *Aggiungo una classe sw Sale al modello di progetto;*
2. *Assegno alla classe la responsabilità di calcolare il suo totale.*

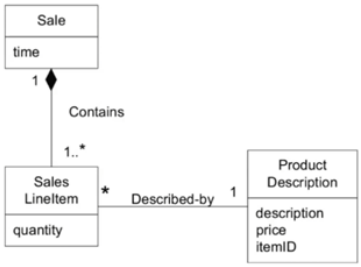


Figure 6.2: Esempio di applicazione del Pattern Information Expert

Discussione: Spesso è necessario che più esperti parziali collaborino. Questo principio riflette anche il mondo reale: si assegna un compito a chi ha le informazioni per svolgerlo.

Controindicazioni: A volte l'esperto ha troppe responsabilità, compromettendo la coesione o aumentando l'accoppiamento.

Vantaggi:

- Favorisce l'incapsulamento
- Supporta la coesione alta
- Mantiene l'accoppiamento basso

Pattern correlati: Low Coupling, High Cohesion, Single Responsibility Principle (SRP)

6.2.3 Low Coupling

Problema: Come ridurre la dipendenza tra le classi per facilitare la manutenzione e il riutilizzo?
Soluzione: Assegna responsabilità in modo che l'accoppiamento tra oggetti rimanga basso. Meno una classe dipende da altre, più sarà flessibile e riusabile.

***Esempio**: Supponiamo di dover creare un'istanza di CashPayment e di associarla alla Sale. Il responsabile sarà Register che invia un messaggio setPayment a Sale, passando il nuovo CashPayment come parametro.*

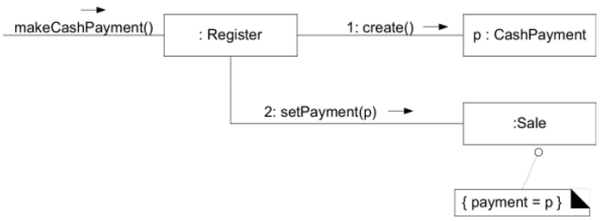


Figure 6.3: Esempio di applicazione del Pattern Low Coupling

Discussione: Ogni tipo di dipendenza (attributi, parametri, metodi, sottoclassi, creazione diretta) aumenta l'accoppiamento. Low Coupling guida le scelte per minimizzare queste dipendenze, senza annullare la collaborazione tra oggetti.

Controindicazioni: L'accoppiamento con classi stabili (poco soggette a cambiamenti) è generalmente accettabile.

Vantaggi:

- Facilità di modifica e test
- Codice riusabile e isolabile
- Progetti più flessibili

Pattern correlati: Protected Variations, Interface Segregation Principle, Dependency Injection

6.2.4 High Cohesion

Problema: Come mantenere oggetti focalizzati, comprensibili e gestibili?
Soluzione: Assegna responsabilità in modo che la coesione degli oggetti rimanga alta, ovvero che le responsabilità siano logicamente correlate. Una classe coesa svolge un insieme di compiti affini e focalizzati in una singola area di responsabilità.

***Esempio**: Register assume non solo la responsabilità di ricevere operazione sistema makeCashPayment ma anche parte della responsabilità di soddisfarla.*

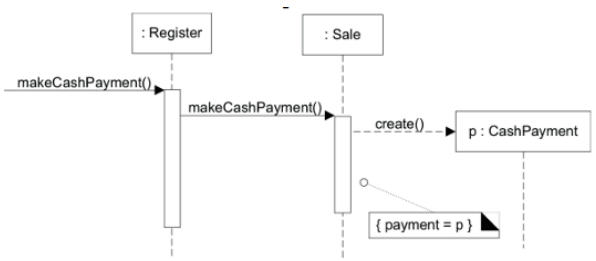


Figure 6.4: Esempio di applicazione del Pattern High Cohesion

Livelli di coesione:

- Molto bassa: responsabilità scollegate in aree diverse
- Bassa: compiti complessi ma in un'unica area
- Moderata: compiti leggeri in aree collegate
- Alta: compiti focalizzati in un'area funzionale, delega il resto

Discussione: Una classe con coesione alta è più semplice da mantenere e riusare.

Controindicazioni: In casi speciali (es. efficienza nei sistemi distribuiti), può essere utile un oggetto con responsabilità più ampie.

Vantaggi:

- Maggiore leggibilità e comprensione

- Migliore manutenibilità
- Migliore riuso per funzionalità specifiche

Pattern correlati: Single Responsibility Principle, Low Coupling

6.2.5 Controller

Problema: Chi riceve e coordina l'esecuzione di un'operazione di sistema?
Soluzione: Assegna la responsabilità a un oggetto controller, che può essere:

- Un facade controller: rappresenta il sistema, un punto di accesso o un sottosistema
- Un use-case controller: rappresenta un caso d'uso specifico

È l'intermediario tra UI e dominio.

***Esempio**: Nel Sistema POS un buon Controller sarà Register.*

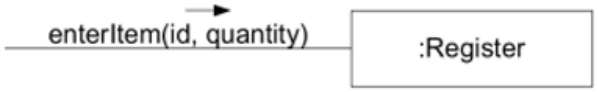


Figure 6.5: Esempio di applicazione del Pattern Controller

Discussione: Il controller non esegue direttamente il lavoro, ma coordina e delega. Attenzione ai controller gonfi, con troppe responsabilità (bassa coesione).

Problemi tipici:

- Un unico controller per troppe operazioni
- Controller che fa troppo, invece di delegare

Soluzioni:

- Aggiungere più controller
- Delegare alle classi del dominio
- Separare lo strato UI dal dominio (principio Layers)

Vantaggi:

- Favorisce la riusabilità del dominio
- Permette UI intercambiabili
- Supporta coerenza nei casi d'uso

Pattern correlati: Command Pattern, Facade, Layers, Pure Fabrication (Use Case Controller)

6.3 Altri pattern grasp

Oltre ai pattern fondamentali, GRASP include altri pattern progettuali che aiutano a gestire la complessità, favorire l'adattabilità e contenere l'impatto dei cambiamenti.

6.3.1 Pure Fabrication

Problema: A chi assegnare una responsabilità quando le soluzioni suggerite da Expert o altri principi violerebbero coesione e basso accoppiamento?
Soluzione: Crea una classe artificiale (non parte del dominio), che raggruppa in modo coeso una responsabilità, allo scopo di:

- Mantenere alta coesione
- Ridurre l'accoppiamento
- Favorire il riuso

Questa classe non rappresenta un concetto del mondo reale, ma è progettata per organizzare meglio il comportamento.

Vantaggi:

- Permette di mantenere le entità del dominio focalizzate
- Incoraggia la separazione delle responsabilità

Svantaggi:

- Un uso eccessivo porta a molti oggetti "tecnici", con rischio di frammentazione del codice
- Una Pure Fabrication deve comunque essere progettata con attenzione: deve collaborare correttamente per svolgere il proprio ruolo

6.3.2 Polymorphism

Problema: Come gestire comportamenti alternativi che variano in base al tipo?
Soluzione: Assegna le responsabilità direttamente ai tipi specifici, utilizzando operazioni polimorfe.
Le decisioni condizionali basate sul tipo possono essere sostituite da invocazioni polimorfe che distribuiscono il comportamento alle sottoclassi.

Vantaggi:

- Le estensioni sono facili da aggiungere
- I client restano indipendenti dalle implementazioni concrete

Svantaggi: Non bisogna progettare polimorfismo in anticipo senza un reale bisogno (per evitare complicazioni inutili)

6.3.3 Indirection

Problema: Come ridurre l'accoppiamento diretto tra componenti che devono collaborare?
Soluzione: Introduce un intermediario tra le componenti, che agisce da mediatore. Questo livello di indirezione isola le parti e:

- Riduce il numero di dipendenze dirette
- Aumenta la flessibilità

L'intermediario può essere un adattatore, un delegato, un proxy, ecc.
Vantaggi:

- Riduce l'accoppiamento tra componenti
- Facilita manutenzione, test, estensioni

Pattern correlati: Adapter, Mediator, Service Layer

6.3.4 Protected Variations

Problema: Come evitare che le variazioni in alcune parti del sistema si propaghino ad altre?
Soluzione: Isola le parti soggette a cambiamento dietro interfacce stabili o punti di protezione.
È una strategia generale per contenere l'instabilità del sistema, anticipando variazioni future.
Vantaggi:

- Facilita l'aggiunta di nuove varianti
- Riduce il costo delle modifiche
- Promuove Low Coupling

Svantaggi:

- È inutile o dannoso proteggere contro variazioni improbabili
- In certi casi è più economico riprogettare che introdurre un'infrastruttura di protezione

Pattern correlati: Strategy, Interface Segregation Principle, Bridge

Chapter 7

Design Pattern

I Design Pattern sono soluzioni progettuali collaudate a problemi ricorrenti nella progettazione software. I pattern GoF (Gang of Four) sono i 23 pattern classici descritti da Gamma, Helm, Johnson e Vlissides, classificati in tre categorie principali:

- *Creazionali*: gestione dell'istanziazione degli oggetti
- *Strutturali*: organizzazione di classi e oggetti
- *Comportamentali*: gestione del comportamento e dell'interazione

7.1 Adapter

Problema: Come integrare componenti con interfacce incompatibili?

- Un oggetto client vuole usare un servizio offerto da un server, ma l'interfaccia è diversa.
- Esistono più implementazioni simili con API differenti.

Soluzione: Si introduce un Adapter che converte l'interfaccia dell'oggetto server in quella attesa dal client.

Strutture:

- *Adapter per classe*: usa l'ereditarietà

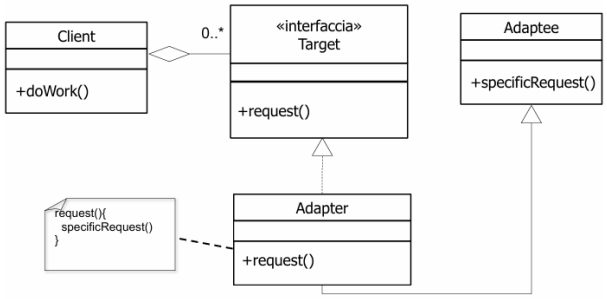


Figure 7.1: Adapter (Classe) Struttura

- *Adapter per oggetto*: usa la composizione

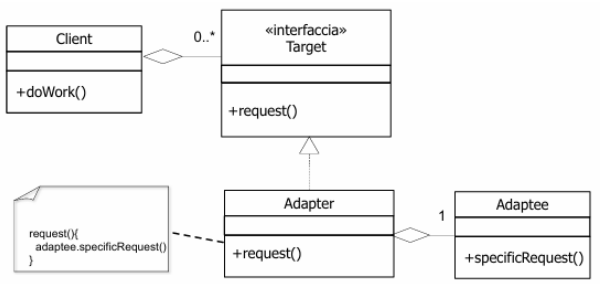


Figure 7.2: Adapter (Oggetto) Struttura

Relazione con i GRASP:

- **Protected Variations**: protegge il sistema da modifiche alle interfacce esterne
- **Indirection**: l'adapter funge da intermediario
- **Polymorphism**: gli adapter condividono un'interfaccia comune

Pattern correlati: Strategy, Facade, Bridge

7.2 Factory

Problema: Chi crea oggetti come gli Adapter, quando la creazione è complessa o dinamica?
Soluzione: Utilizzare una Factory, ovvero un oggetto responsabile di astrarre la logica di creazione.

Vantaggi:

- Separa la logica di istanziazione
- Nasconde i dettagli della creazione
- Permette gestione avanzata (es. caching, riciclo)

Pattern correlati:

- Singleton per l'accesso globale alla Factory
- Pure Fabrication: la Factory è una classe artificiale coesa

7.3 Singleton

Problema: Come garantire che una classe abbia una sola istanza, accessibile globalmente?
Soluzione: Definire un metodo getInstance() che restituisce sempre la stessa istanza.

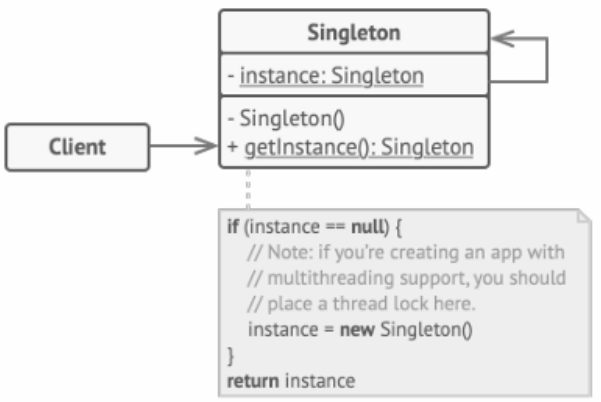


Figure 7.3: Struttura di Singleton

Vantaggi:

- Centralizza e controlla l'accesso
- Alternativa più sicura alle variabili globali

Controindicazioni:

- Rischio di accoppiamento globale
- Problematico in ambiente distribuito

Pattern correlati: Factory, Facade

7.4 Strategy

Problema: Come gestire algoritmi alternativi (es. sconti, tassazione, calcoli)?
Soluzione: Definire un'interfaccia comune e implementare diverse strategie concrete intercambiabili.

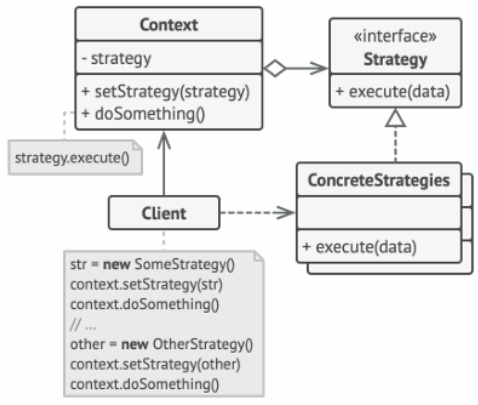


Figure 7.4: Struttura del Strategy

Vantaggi:

- Algoritmi facilmente sostituibili

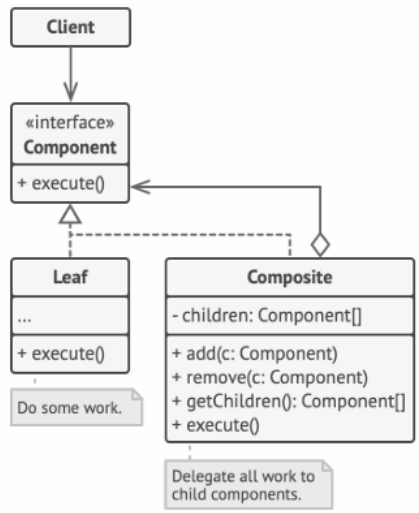


Figure 7.5: Struttura di Composite

- Separazione tra comportamento e utilizzatore

Relazione con GRASP:

- Usa Polymorphism
- Supporta Protected Variations

Pattern correlati: Factory (per creare strategie), Composite (per combinare strategie)

7.5 Composite

Problema: Come trattare in modo uniforme comportamenti singoli o composti?
Soluzione: Usare strutture ad albero in cui oggetti composti e oggetti semplici condividono la stessa interfaccia.
Vantaggi:

- Flessibilità nella composizione
- Uniformità nel trattamento

Pattern correlati: Strategy, Command, Decorator

7.6 Facade

Problema: Come semplificare l'accesso a un sottosistema complesso?
Soluzione: Fornire una facciata: un oggetto front-end che incapsula e coordina l'accesso a più componenti interni.
Relazione con GRASP:

- Supporta Low Coupling
- Usa Indirection
- Spesso realizzato come Facade Controller

Pattern correlati: Singleton, Adapter, Controller

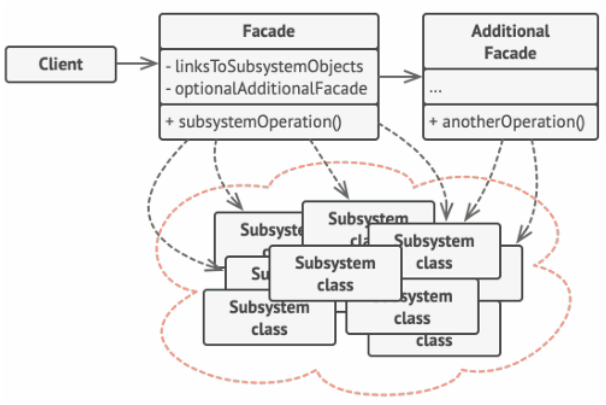


Figure 7.6: Struttura del Facade

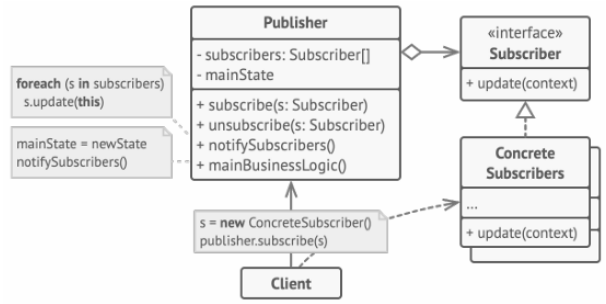


Figure 7.7: Struttura dell'Observer

7.7 Observer

Problema: Come notificare più oggetti quando cambia lo stato di un oggetto?
Soluzione: Usare Publisher (Subject) e Subscriber (Observer). Quando lo stato cambia, il publisher notifica tutti gli osservatori.
Vantaggi:

- Accoppiamento debole
- Comunicazione dinamica tra componenti

Pattern correlati: Eventi GUI, Model-View-Controller (MVC)

Chapter 8

Sviluppo guidato dai test e refactoring

8.1 Progettare la visibilità

Cos'è la visibilità?
In un sistema orientato agli oggetti, visibilità è la capacità di un oggetto di "vedere" o avere un riferimento a un altro oggetto.
In pratica, se un oggetto A vuole inviare un messaggio a un oggetto B, A deve avere visibilità su B. Questo significa che A deve disporre di un riferimento a B per poterlo invocare.

8.2 Modalità di visibilità

La visibilità può essere ottenuta in quattro modi comuni. Ognuno offre un diverso grado di permanenza del riferimento:

Visibilità per Attributo

Definizione: Un oggetto B è un attributo dell'oggetto A.

Caratteristiche:

- Visibilità permanente, persiste fintanto che gli oggetti A e B esistono.
- Ideale quando A ha bisogno frequente di accedere a B.

Visibilità per Parametro

Definizione: B viene passato come parametro a un metodo di A.

Caratteristiche:

- Visibilità temporanea, limitata all'interno del metodo.
- È utile quando A ha bisogno di usare B solo per l'esecuzione di un metodo.

Visibilità Locale

Definizione: B è una variabile locale dichiarata all'interno di un metodo di A.

Caratteristiche:

- Anche questa è temporanea, ma più limitata della visibilità per parametro.
- Può derivare da una nuova istanza o da un metodo invocato.

Visibilità Globale

Definizione: B è globale rispetto ad A.

Caratteristiche:

- Permanente, ma meno usata nei linguaggi moderni per motivi di buona progettazione.
- In C++ si può avere una variabile globale. In Java, si simula attraverso Singleton.

È possibile trasformare un tipo di visibilità in un altro per migliorare la struttura del progetto:

- Locale → Parametro: si passa l'oggetto come argomento a un altro metodo.
- Parametro → Attributo: si memorizza l'oggetto ricevuto in un campo della classe.

8.3 Sviluppo guidato dai test

Nel contesto di Unified Process (UP), gli elaborati della progettazione (diagrammi delle classi, di interazione, contratti delle operazioni, ecc.) sono input fondamentali per la generazione del codice.
La progettazione orientata agli oggetti non è fine a sé stessa: è un mezzo per guidare la codifica, riducendo ambiguità e migliorando la qualità del software prodotto.
Anche con una buona progettazione, durante la programmazione:

- Avviene ulteriore progettazione e prototipazione
- Si introducono modifiche e deviazioni dal progetto iniziale

La progettazione non termina prima della codifica, ma evolve con essa. La generazione del codice non è parte del modello OOA/D, ma ne rappresenta un obiettivo finale.
L'implementazione in un linguaggio orientato agli oggetti implica:

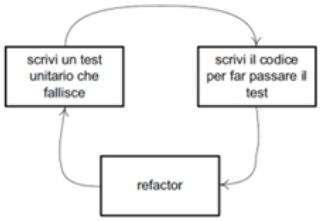
- Definizione di classi e interfacce
- Dichiarazione delle variabili di istanza
- Definizione di metodi, costruttori e comportamenti

8.3.1 Test-Driven Development (TDD)

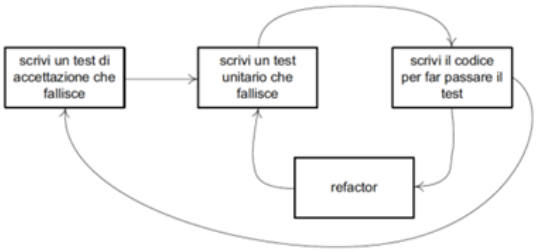
Il Test-Driven Development (TDD) è una tecnica agile, promossa da XP e compatibile con UP e Scrum.

Idea di base:

1. Scrivi un test che fallisce per una funzione non ancora esistente
2. Scrivi il codice minimo per farlo passare
3. Rifattorizza il codice mantenendo il test verde



Il ciclo di base del TDD per i test unitari.



Il doppio ciclo del TDD.

Vantaggi del TDD

- I test vengono scritti per davvero
- Il codice è più chiaro e documentato dal comportamento atteso
- Le interfacce risultano più stabili e pensate
- Migliora la fiducia nel cambiare il codice
- Tutto è automaticamente verificabile

Tipi di test

- Test di unità (su singole classi o metodi)
- Test di integrazione (collaborazioni tra componenti)
- Test di sistema (completo end-to-end)
- Test di accettazione (validazione dei requisiti con il cliente)

Schema di un test unitario

1. Preparazione: crea la fixture (es. new Sale())
2. Esecuzione: invoca i metodi da testare
3. Verifica: confronta il risultato con il valore atteso
4. Rilascio (opzionale): pulisci risorse, se necessario

Linee Guida

- Scrivi solo il codice minimo per far passare il test
- Poi rifattorizza il codice in modo pulito
- La semplicità delle regole è ingannevole: richiedono disciplina, ma portano a software migliore.

8.4 Refactoring

Il Refactoring è un metodo strutturato e disciplinato per riscrivere o riorganizzare codice esistente senza modificarne il comportamento esterno. Si realizza applicando piccoli passi di trasformazione, accompagnati da test continui, eseguiti dopo ogni modifica.

Obiettivi del Refactoring

- Migliorare continuamente la qualità del codice
- Preparare il codice al cambiamento
- Rendere il codice più leggibile, conciso e riutilizzabile
- Eliminare ridondanze, duplicazioni e responsabilità mal distribuite

Un codice ben rifattorizzato è breve, chiaro e privo di duplicazioni. Appare come scritto da un "maestro programmatore".

Quando fare Refactoring?

- Quando si aggiunge una nuova funzionalità
- Quando si corregge un bug
- Durante la revisione del codice
- Quando si identificano dei code smell

8.4.1 Code Smell

Un code smell è un segnale di debolezza strutturale nel codice. Non è necessariamente un errore, ma suggerisce che il design può essere migliorato.

Tipologie principali:

- *Bloater*: Codice troppo grande e complesso
- *Change Preventer*: Impedimenti ai cambiamenti
- *Object Orientation Abuser*: Violazioni ai principi di OOP (es. switch)
- *Coupler*: Eccessivo accoppiamento tra classi
- *Dispensable*: Codice o commenti inutili, duplicazioni

Long Method
Sintomo: Metodo troppo lungo (oltre 10 righe)
Problema: Difficile da leggere, modificare, riutilizzare
Soluzione: Utilizzare Extract Method per isolare sezioni del metodo in sottometodi descrittivi.

Duplicated Code
Sintomo: Stesso codice in punti diversi del programma
Problema: Difficile da mantenere; errori facilmente replicabili
Soluzione: Extract Method e chiamata condivisa nei punti duplicati.

Feature Envy
Sintomo: Un metodo accede più ai dati di un altro oggetto che ai propri
Problema: Indica che il metodo è collocato nella classe sbagliata
Soluzione: Move Method nella classe di cui usa maggiormente i dati.

Large Class
Sintomo: Troppi metodi e attributi in una sola classe
Problema: Responsabilità eccessive; difficile da gestire
Soluzione: Extract Class, Move Method o Extract Subclass per dividere le responsabilità.

Data Class
Sintomo: Classe con soli attributi e getter/setter
Problema: Comportamenti esterni concentrati in altre classi (Feature Envy)
Soluzione: Usare Move Method per spostare comportamenti dentro la data class.

Long Parameter List
Sintomo: Metodo con molti parametri
Problema: Difficile da leggere, fragile nel tempo
Soluzione: Replace Parameter with Method Call, Preserve Whole Object, Introduce Parameter Object.

Shotgun Surgery Sintomo: Per una modifica bisogna toccare molte classi
Problema: Logica dispersa; alta fragilità
Soluzione: Move Method, Move Field o Inline Class per accorpare la responsabilità.

Comment Sintomo: Presenza eccessiva di commenti esplicativi
Problema: Il codice dovrebbe essere autoesplicativo
Soluzione: Rename Method, Extract Variable, Extract Method, Introduce Assertion.

Refused Bequest
Sintomo: Una sottoclasse eredita metodi che non utilizza
Problema: La gerarchia è sbagliata; comportamenti non richiesti
Soluzione: Replace Inheritance with Delegation oppure Extract Superclass per una gerarchia più corretta.