

Analisi e progettazione del software

Parlare di software si intende un insieme di programmi per computer e la documentazione che li accompagna, come la specifica dei requisiti, i modelli di progetto, il manuale utente, e così via. Il software può essere di due tipi:

- **Generico**, sviluppato per un ampio insieme di clienti
- **Personalizzato**, sviluppato per un singolo cliente in base alle sue esigenze.

Per creare nuovo software, si possono usare diverse strategie, come sviluppare nuovi programmi da zero, personalizzare quelli già esistenti, o riusare software già sviluppato.

L'obiettivo di chi crea software è di produrre software di **buona qualità**, cioè software che abbia le seguenti caratteristiche:

- **Mantenibilità**, ovvero che sia scritto in modo tale che possa evolvere per soddisfare le esigenze che cambiano nel tempo
- **Fidatezza**; ovvero che gli utenti possano avere fiducia nel suo funzionamento e nella sua sicurezza
- **Efficienza**, ovvero che deve essere progettato in modo tale che usi un basso numero di risorse
- **Accettabile**, ovvero che deve essere accettato dagli utenti per cui è stato progettato, e deve essere facile da usare, da capire e da apprendere.

L'**ingegneria del software** è la disciplina che si occupa della costruzione del software di qualità, e si occupa di tutti gli aspetti della produzione del software, dalle prime fasi in cui si definiscono i requisiti e si progetta il software, fino alle fasi finali in cui si verifica, si valida e si mantiene il software. Usa teoria e metodi appropriati per risolvere i problemi tenendo conto dei vincoli organizzativi e finanziari.

Sviluppo del software

Per sviluppare il software, si segue un processo software, che è una serie di passi che vanno dall'idea al prodotto. Esistono diversi processi software, che si differenziano per il modo in cui organizzano le fasi e le attività dello sviluppo; alcuni esempi sono il *processo a cascata*, il *processo unificato* e il *processo a spirale*.

Le attività più comuni a tutti i processi software sono:

- **Analisi dei requisiti** consiste nel capire cosa il software deve fare, e quali sono le esigenze e le aspettative degli utenti

- **Progettazione** consiste nel definire come il software deve essere organizzato e funzionare
- **Implementazione** consiste nel scrivere il codice del software
- **Evoluzione** consiste nel modificare e aggiornare il software per adattarlo ai cambiamenti delle esigenze e del contesto
- **Validazione** consiste nel verificare e testare il software per assicurarsi che funzioni correttamente e soddisfi i requisiti.

Inoltre, il processo software ci definisce anche quali *artefatti* bisogna produrre per ogni attività, quali sono i *ruoli* e le *responsabilità* delle persone coinvolte, e quali sono le *condizioni* che devono essere vere prima e dopo ogni attività.

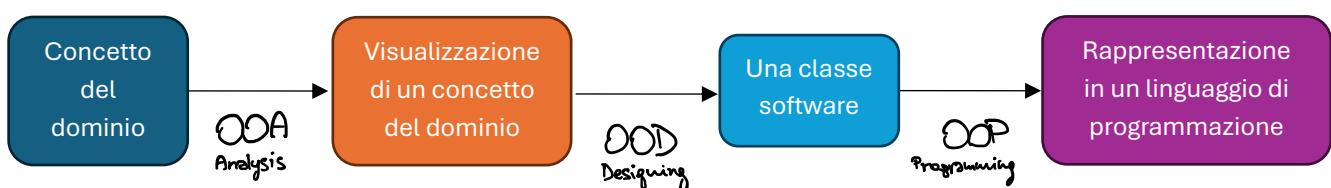
Suddivisione

Il processo software viene diviso in due parti:

- L'**analisi** enfatizza un'investigazione di un problema e dei suoi requisiti, anziché di una soluzione – si concentra quindi sul *dominio del problema*. Se è orientata agli oggetti, si enfatizza sull'*identificazione dei concetti o degli oggetti*.
- La **progettazione** enfatizza una soluzione concettuale che soddisfa i requisiti del problema. Se è orientata agli oggetti, enfatizza sulla *definizione di oggetti software* che collaborano per soddisfare i requisiti.

Per questo procedimento, possiamo definire quattro fasi:

- **Definizione dei casi d'uso**, che sono storie scritte relative al mondo in cui il sistema viene utilizzato, e che descrivono le funzionalità che il sistema deve offrire, e le interazioni tra il sistema e gli utenti.
- **Definizione di un modello di dominio**, che è un diagramma che mostra i concetti o gli oggetti significativi del dominio, con i loro attributi e le associazioni tra di essi
- **Definizione di diagrammi di interazione**, che sono diagrammi che mostrano le collaborazioni tra gli oggetti software, e le sequenze di messaggi che si scambiano per realizzare un caso d'uso
- **Definizione dei diagrammi delle classi di progetto**, che sono diagrammi che mostrano la vista statica delle definizioni delle classi software, con i loro attributi e metodi, e le relazioni tra di esse.



Unified Modelling Language

Per rappresentare i diagrammi di analisi e progettazione, si può usare il linguaggio **UML**, che è un linguaggio visuale di modellazione dei sistemi software e non solo, che rappresenta una collezione di best practices di ingegneria, dimostra testi vincenti nella modellazione di sistemi vasti e complessi.

È uno standard, ed è unificato per diversi aspetti:

- **Storico**, perché prende le parti migliori dei linguaggi precedenti e le unifica
- **Ciclo di sviluppo**, perché usa la stessa sintassi visuale per tutte le fasi dello sviluppo
- **Domini applicativi**, perché si adatta a diversi tipi di sistemi, da quelli embedded a quelli gestionali
- **Linguaggi e piattaforme di sviluppo**, perché non dipende da esse
- **Processi di sviluppo**, perché si integra con diversi processi software.

Modella quindi i sistemi come insiemi di oggetti che collaborano tra loro, distinguendo tra di loro due strutture:

- **Struttura statica**: riguarda quali tipi di oggetti sono necessari, e come sono tra loro correlati
- **Struttura dinamica**: riguarda il ciclo di vita di questi oggetti, e come collaborano per fornire le funzionalità richieste

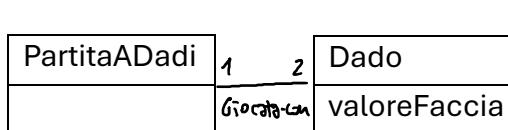
Può essere usato in tre modi:

- **Come abbozzo**, per creare diagrammi di progetto informali e incompleti, che servono a comunicare le idee principali
- **Come progetto**, per creare diagrammi di progetto relativamente dettagliati, che servono a documentare il sistema
- **Come linguaggio di programmazione**, per creare diagrammi completamente eseguibili di un sistema software con UML, e per generare il codice a partire da uno di essi.

Le classi

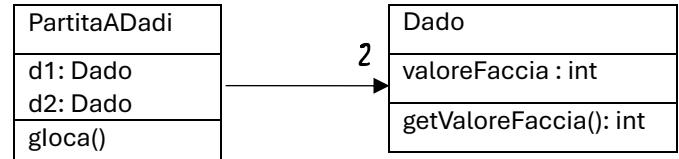
All'interno del diagramma UML, possiamo delineare le classi in due diversi sottogruppi:

- **Concettuale:** è un oggetto o concetto del mondo reale, da un punto di vista concettuale ed essenziale. Fa parte del *modello di dominio*
- **Software:** rappresenta un componente software. Fa parte del *modello di progetto*



Punto di vista concettuale (modello di dominio)

La notazione dei diagrammi delle classi UML è utilizzata per visualizzare concetti del mondo reale



Punto di vista software (diagramma delle classi di progetto)

La notazione dei diagrammi delle classi di UML è utilizzata per visualizzare elementi software

Processo software

Un processo software è essenzialmente un **insieme di attività correlate** che definiscono chi fa cosa, quando e come. Questo approccio stabilisce le attività da svolgere, i ruoli coinvolti, le metodologie da applicare e la tempistica di ogni fase.

Processi agili e basati sul piano

I processi basati sul piano sono quelli in cui tutte le attività sono pianificate in anticipo, e i progressi vengono misurati rispetto a questo piano. Al contrario, i processi agili adottano una **pianificazione incrementale**, il che li rende più flessibili e adatti a modifiche in risposta alle esigenze mutevoli dei clienti. Nella pratica, la maggior parte dei processi **combina elementi** dei due tipi di modelli.

Modelli di sviluppo software

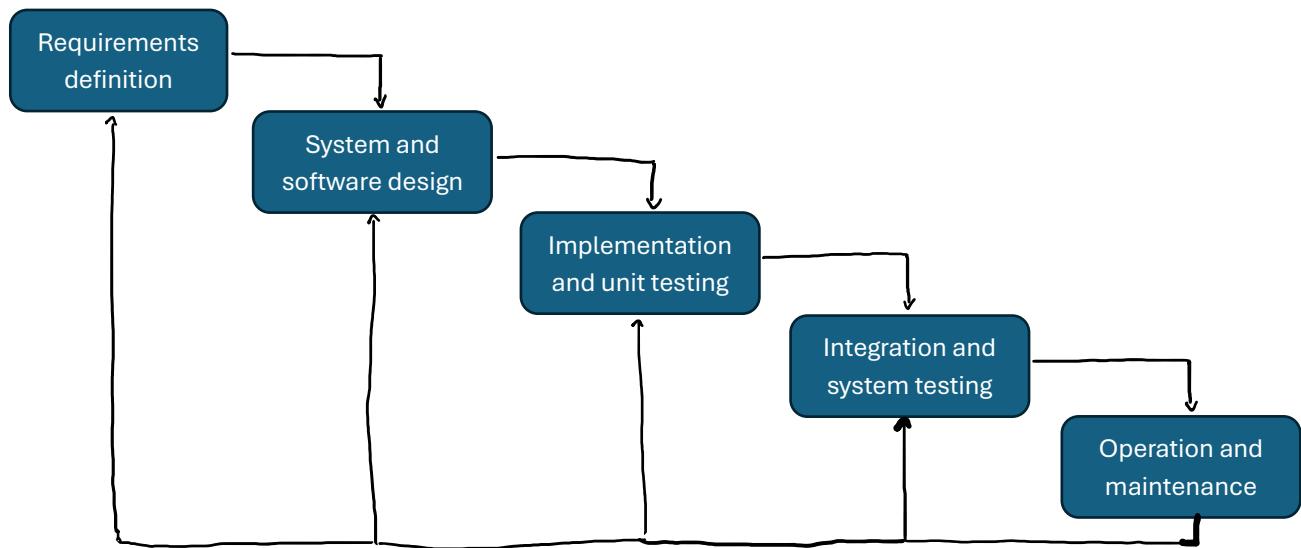
Abbiamo diversi modelli di sviluppo:

- **Modello a cascata:** questo modello è fortemente guidato dal piano, con fasi separate e distinte di specifica e sviluppo
- **Sviluppo incrementale:** caratterizzato da una successione di specifica, sviluppo e validazione, questo modello può essere sia guidato dal piano, sia agile.

- **Integrazione e configurazione:** i sistemi sono assemblati a partire da componenti configurabili esistenti. Questo approccio può essere adottato sia in contesti pianificati sia agili.

Modello a cascata

Il modello a cascata è un approccio sequenziale allo sviluppo del software, dove ogni fase deve essere completata prima di passare alla successiva. È noto per la sua struttura rigida e ben definita, che lo rende adatto a progetti con requisiti chiari e stabili.

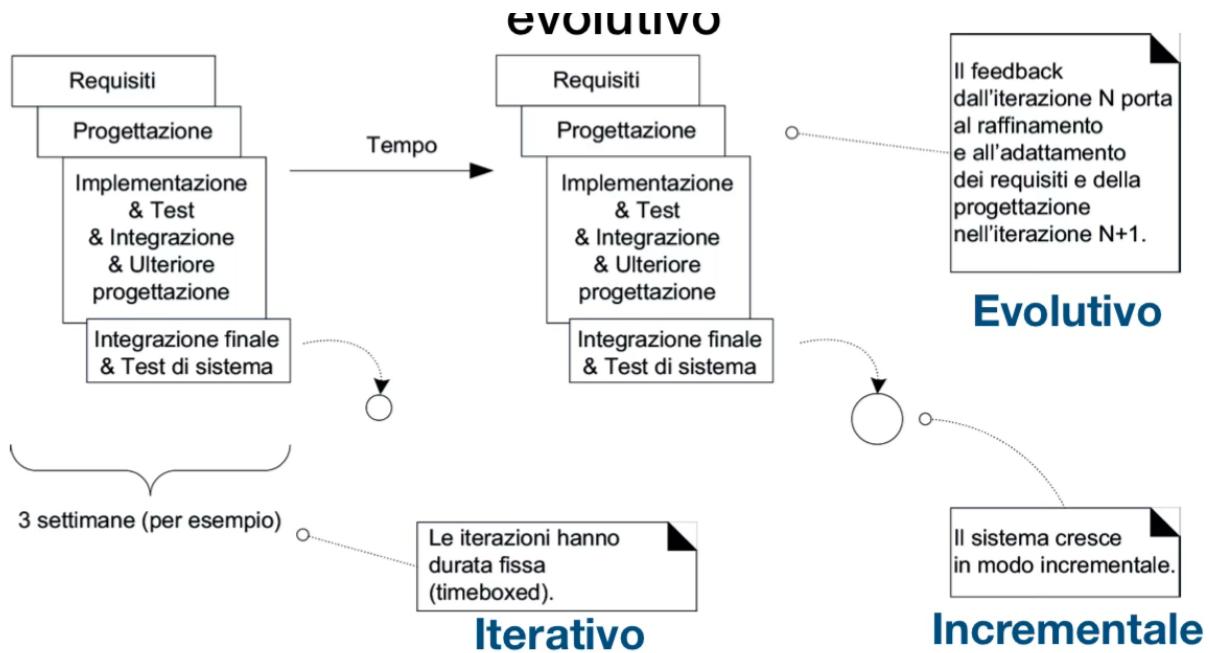


Il problema principale di questo modello è la sua **inflessibilità**: non è adattabile a cambiamenti, e richiede il completamento di una fase prima di passare alla successiva. È appropriato solo quando i requisiti sono ben compresi e le modifiche sono limitate. Viene utilizzato soprattutto in progetti di ingegneria di sistemi di grandi dimensioni sviluppati in diversi siti, in quanto facilita il coordinamento del lavoro.

Sviluppo iterativo

L'idea alla base dello sviluppo iterativo è quella di realizzare un'implementazione iniziale del sistema, per permettere agli utenti di fornire feedback immediati. Ogni iterazione seleziona un piccolo sottosinsieme di requisiti per una rapida progettazione, implementazione e test.

Il processo procede attraverso cicli strutturati di **costruzione-feedback-adattamento**, permettendo al sistema di evolvere e convergere verso i requisiti corretti e il progetto più appropriato. Nelle iterazioni finali, è improbabile che si verifichino cambiamenti significativi dei requisiti.



I vantaggi dello sviluppo iterativo includono una **minore probabilità di fallimento del progetto**, una **maggior produttività**, **minori percentuali di difetti**, una **riduzione precoce dei rischi maggiori**, **progresso visibile fin dall'inizio**, **feedback e adattamento tempestivi**, e una **gestione efficace della complessità**.

Gli svantaggi sono che il processo può essere **meno visibile**, e la struttura del sistema può **degradarsi** con l'aggiunta di nuovi incrementi, a meno che non si investa tempo e denaro nel refactoring per migliorare il software.

Il **feedback** è un elemento cruciale in questo processo, proveniente sia dai test e dagli sviluppatori per raffinare il progetto, sia dal team per affinare le stime di tempi e costi, sia dal cliente e dal mercato per assegnare o modificare le priorità ai requisiti da affrontare nell'interazione successiva.

È importante che un pensiero a cascata non debba invadere un progetto iterativo. Questo succede quando, per esempio, *si scrive la maggior parte dei requisiti o i casi d'uso prima dello sviluppo, o si crea in modo dettagliato e completo delle specifiche, dei modelli o il progetto prima di iniziare l'implementazione*.

La **prima iterazione** si concentra sugli elementi più critici e rischiosi del progetto. Questo è essenziale per identificare e risolvere i problemi principali il prima possibile, garantendo che il progetto sia una traiettoria di successo.

Timeboxing

Nel contesto dello sviluppo iterativo, il timeboxing è una tecnica che stabilisce una durata fissa per ogni iterazione. Questo approccio enfatizza l'importanza di **passi misurati, feedback tempestivi e adattamenti rapidi**. Iterazioni troppo lunghe possono allontanarsi dallo spirito dell'agilità e aumentare i rischi, mentre quelle troppo brevi potrebbero non consentire di realizzare abbastanza lavoro per ottenere feedback significativi.

Una volta stabilita, la durata di un'iterazione rimane invariata per mantenere la coerenza e l'efficacia del feedback.

Flessibilità e modificabilità del codice

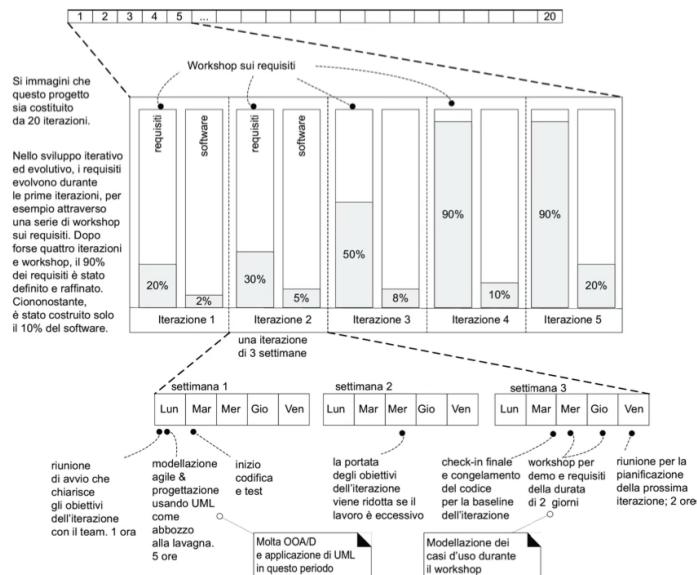
Per adottare con successo lo sviluppo iterativo, è fondamentale che il software sia progettato con **flessibilità**, permettendo che l'impatto dei cambiamenti sia gestibile. Questo richiede che il codice sorgente sia non solo facilmente modificabile ma anche comprensibile, per facilitare eventuali adattamenti.

Pianificazione iterativa

La pianificazione delle iterazioni si distingue dal modello a cascata; non è necessario pianificare **l'intero progetto in dettaglio fin dall'inizio**. I processi iterativi favoriscono una pianificazione girata dal rischio e dai requisiti del cliente. In questo processo, **avere un'architettura di base solida** è cruciale per mitigare i rischi associati allo sviluppo.

Una volta definiti gli obiettivi, i **requisiti di un'iterazione vengono fissati** e non dovrebbero cambiare durante l'iterazione stessa. Questo permette al team di lavorare senza interruzione,

Come eseguire l'analisi e la progettazione in modo iterativo ed evolutivo



mentre i committenti **possono interagire con il team solo al termine di ogni iterazione**. Il team di sviluppo ha la *flessibilità di modificare il piano se necessario*.

Sviluppo agile

Lo sviluppo agile è una forma di sviluppo iterativo che valorizza l'**agilità**, ovvero una risposta rapida e flessibile ai cambiamenti. I metodi agili si basano su iterazioni brevi e **timeboxed**, con una pianificazione iterativa, e promuovono consegne incrementale. Essi sostengono valori agili come la **semplicità**, la **leggerezza**, il **valore delle persone** e la **comunicazione**, oltre a pratiche agili come la *programmazione a coppie*, il *Test-Driven Development*, il *refactoring* e altri.

Modellazione agile

La modellazione agile non esclude la modellazione, ma la utilizza per facilitare la comprensione e la comunicazione. Si applica UML solo alle parti del progetto che presentano difficoltà, sono insolite o particolarmente complesse, utilizzando gli strumenti più semplici possibili. La modellazione dovrebbe essere un'**attività collaborativa**, fatta a coppie o in piccoli gruppi, e si deve accettare che tutti i modelli saranno inevitabilmente incompleti e imprecisi.

Riutilizzo del software

Possiamo integrare dei sistemi da componenti o sistemi applicativi esterni. Gli elementi riutilizzati vengono configurati per adattare il comportamento e la funzionalità secondo uno standard. A seconda dell'implementazione che viene effettuata, abbiamo diverse tipologie di software riutilizzabile:

- **Sistemi applicativi stand-alone** configurati per l'uso in un ambiente specifico
- **Collezioni di oggetti** sviluppati come pacchetti da integrare con un framework
- **Servizi web** sviluppati secondo standard di servizio, e disponibili per invocazione remota

I vantaggi di questo approccio includono la **riduzione dei costi e dei rischi, meno sviluppo da zero**, e una **consegna più rapida**. Gli svantaggi, invece, comprendono possibili compromessi sui requisiti, e una perdita di controllo sull'evoluzione degli elementi di sistema riutilizzati.

Processo unificato

Il processo unificato è un approccio iterativo ampiamente adottato per lo sviluppo di software orientato agli oggetti. Ha qualche caratteristica particolare:

- **Promuove** l'integrazione di pratiche da altri metodi iterativi
- **È guidato** dai casi d'uso e dai fattori di rischio
- **Incentrato** sull'architettura
- **Approccio** iterativo, incrementale ed evolutivo

Segue anche una lista di principi:

- **Affrontare** le problematiche di rischio maggiore e di alto valore nelle prime iterazioni
- **Coinvolgere** continuamente gli utenti per valutazioni, feedback e definizione dei requisiti
- **Sviluppare** un'architettura coesa nelle prime iterazioni
- **Assicurare** una verifica continua della *qualità*, testando presto e frequentemente in modo realistico
- **Utilizzare** la modellazione visuale con UML
- **Gestire** con attenzione i *requisiti* e le *richieste di cambiamento*.

All'interno del processo unificato, abbiamo diversi step in cui un prodotto viene a formarsi, chiamate **iterazioni**. Nel cuore del Processo Unificato, troviamo la flessibilità come principio cardine. Sebbene alcune pratiche, come lo sviluppo iterativa e la gestione del rischio, siano fondamentali, la maggior parte delle attività e degli elaborati sono opzionali. La personalizzazione dell'UP per un progetto specifico può essere delineata in un documento sintetico noto come “*scenario di sviluppo*”.

UP non è stato concepito per essere un approccio pesante o rigido; molti dei suoi elementi sono opzionali, permettendo una personalizzazione agile. In un'applicazione agile dell'UP, si seleziona un insieme ristretto di attività ed elaborati, e si procede **senza completare i requisiti e la progettazione prima dell'implementazione**, favorendo una **pianificazione interattiva e adattiva**.

Ideazione

L'ideazione è la fase iniziale di un progetto, in cui si studia il problema da risolvere e si valuta la fattibilità e la convenienza della soluzione proposta. Si tratta di una fase di analisi e di decisione, in cui si pongono domande come: *qual è la visione e lo studio economico del progetto? Il progetto è realizzabile con le risorse e i tempi disponibili? Conviene acquistare un sistema già esistente o svilupparlo da zero? Quanto costa e quanto dura il progetto?*

Durante la fase di ideazione, si svolge un **workshop sui requisiti**, si definiscono gli attori e i casi d'uso, e si identificano i requisiti di qualità e i rischi. Si sviluppano anche *prototipi*, e si

abbozza un'architettura di alto livello. Questa fase è cruciale per stabilire una visione chiara del progetto, e per pianificare le iterazioni successive. Inoltre, si fa la visione di progetto, lo studio economico e dei tempi di sviluppo, l'analisi di circa il 10% dei Casi d'Uso, dei requisiti non funzionali più critici, e la preparazione dell'ambiente di sviluppo.

Si ha **una sola iterazione**. All'interno della fase di ideazione, si producono i seguenti documenti:

- **Visione e studio economico:** descrive obiettivi e vincoli di alto livello
- **Modello dei casi d'uso:** descrive i requisiti funzionali
- **Specifiche supplmentari:** altri requisiti non funzionali
- **Glossario:** dizionario dei dati e terminologia chiave del dominio
- **Lista dei rischi e piano di gestione dei rischi**
- **Prototipi e prof of concept**
- **Piano dell'iterazione**
- **Piano delle fasi e piano di sviluppo del software**
- **Scenario di sviluppo:** descrizione della personalizzazione dei passi e degli elaborati

Requisiti

Per rispondere a queste domande, l'ideazione richiede una certa **esplorazione dei requisiti**, ovvero delle capacità e delle condizioni che il sistema e il progetto devono soddisfare. I requisiti possono essere di due tipi:

- **Funzionali:** descrivono il comportamento del sistema in termini di funzionalità fornite ai suoi utenti, sono informazioni che il sistema deve gestire
 - Funzionalità o servizi che l'OS deve fornire, risposte che l'utente si aspetta, o risultati che il software deve produrre dati determinati input
- **Non funzionali:** sono relativi a proprietà del sistema nel suo complesso, come per esempio *sicurezza, prestazioni, scalabilità, usabilità*, ecc....
 - Definiscono proprietà e vincoli del sistema, quali sono i vincoli nel processo usato per sviluppare il software, etc.
 - Influisce sull'architettura complessiva del sistema, piuttosto che su singoli componenti. Per garantire il rispetto dei requisiti prestazionali, può essere necessario organizzare il sistema per ridurre al minimo le comunicazioni tra i componenti
 - Un unico requisito può generare serie di requisiti funzionali correlati che definiscono servizi di sistema necessari
 - Sono difficili da stabilire con precisione, e dei requisiti imprecisi possono essere difficili da verificare.

I problemi sorgono quando i **requisiti funzionali non sono indicati con precisione**. Se sono ambigui possono essere interpretati in modo diversi da sviluppatori e utenti. Dobbiamo avere quindi un **approccio sistematico** per trovare, documentare, organizzare e tracciare i requisiti che cambiano di sistema. Per validare una serie di requisiti, abbiamo una serie di tecniche: *interviste con clienti, scrivere casi d'uso con clienti, workshop dei requisiti con sviluppatori e clienti, gruppi di lavoro con rappresentanti clienti, dimostrazione a clienti di ciascuna iterazione per feedback.*

Per documentare i requisiti in modo chiaro e completo, si utilizzano diversi strumenti e tecniche, tra cui:

- Il **modello di casi d'uso**, che rappresenta graficamente e testualmente gli scenari tipici di interazione tra gli utenti e il sistema, evidenziando gli obiettivi e i vincoli di ciascuno
- Le **specifiche supplementari**, che raccolgono tutti i requisiti che non sono espressi dai casi d'uso, come i requisiti *non funzionali* e i requisiti di *dominio*
- Il **glossario**, che definisce i termini significativi e ambigui del dominio del problema, in modo da evitare confusioni e fraintendimenti tra le parti coinvolte nel progetto
- La **visione**, che riassume ad alto livello i requisiti del sistema, illustrando la sua finalità, il suo valore aggiunto, il suo contesto di utilizzo, e il suo pubblico di riferimento. Questo include anche uno studio economico per valutare la fattibilità e la redditività del progetto
- Le **regole di business**, che descrivono i requisiti e le politiche che trascendono il singolo progetto, come le regole di mercato, ecc....

Per scrivere questi requisiti, si devono seguire alcune buone pratiche, come:

- Adottare un **formato standard e riutilizzabile** che faciliti la *lettura, la comprensione e la manutenzione dei requisiti*
- Usare un **linguaggio consistente**, che sia coerente con il glossario, e che usi parole chiave come “deve” per i requisiti obbligatori, e “dovrebbe” per i requisiti desiderabili
- Usare l'**evidenziazione del testo** per mettere in risalto le parti più importanti e rilevanti dei requisiti
- Evitare il **gergo informatico**, che potrebbe essere incomprensibile o fuorviante per i clienti o gli utenti finali, e preferire un *linguaggio naturale e comprensibile*.

Flusso di lavoro dei requisiti

Il flusso di lavoro dei requisiti gioca un ruolo cruciale nel processo di sviluppo del software. La sua finalità è quella di **delineare una specifica di alto livello** che definisca con precisione le funzionalità e le caratteristiche che il software dovrà implementare. Questo processo inizia

con l'*intervista delle parti interessate*, o stakeholder, che rappresenta il metodo più efficace per raccogliere i requisiti essenziali.

Abbiamo vari ruoli di esempio:

- **Analisti di sistema:** incaricati di identificare gli attori e i casi d'uso, oltre a strutturare il modello dei casi d'uso
- **Architetti:** compito di assegnare le priorità ai casi d'uso
- **Analisti dei casi d'uso:** descrizione dettagliata di ciascun caso
- **Progettisti di interfaccia utente:** prototipazione dell'interfaccia, assicurando che sia intuitiva e risponda alle esigenze degli utenti

Per garantire una gestione completa dei requisiti, è prevista un'**estensione** del flusso di lavoro dei requisiti dell'Unified Process attraverso l'introduzione di nuove attività.

Organizzazione e attributi

Con l'aumentare del numero dei requisiti, diventa fondamentale organizzarli in una **tassonomia**, ovvero una gerarchia che permette di raggruppare i requisiti in categoria. Generalmente, due o tre livelli di profondità sono sufficienti per sistemi di complessità non elevata.

Ogni requisito può essere associato a diversi attributi che forniscono informazioni aggiuntive. Ad esempio, la **priorità** è un attributo comune, spesso definito attraverso il sistema

MoSCoW, che include:

- **Must have:** requisiti fondamentali per il sistema
- **Should have:** requisiti importanti ma non essenziali
- **Could have:** requisiti opzionali da realizzare se vi è tempo disponibile
- **Want to have:** requisiti che potrebbero essere implementati in versioni future del sistema

Elaborazione

L'elaborazione è il periodo in cui si **realizza il nucleo dell'architettura software**, e si affrontano i rischi più significativi. È un processo iterativo che include la programmazione, la verifica e il testing. L'obiettivo è stabilizzare i requisiti, e definire un piano di lavoro dettagliato.

È spesso costituita da due o più iterazioni che durano ognuna dalle 2 alle 6 settimane. È importante anche *iniziare presto la programmazione, effettuare test presto, adattare in base al feedback, e scrivere la maggior parte dei casi d'uso e degli altri requisiti nel dettaglio*. Vengono iniziati anche alcuni elaborati, spiegati qua di fianco:

Elaborato	Commento
Modello di Dominio	È una visualizzazione dei concetti del dominio, simile a un modello statico delle informazioni delle entità del dominio.
Modello di Progetto	È l'insieme dei diagrammi che descrivono la progettazione logica. Comprende diagrammi delle classi software, diagrammi di interazione degli oggetti, diagrammi dei package e così via.
Documento dell'Architettura Software	Un aiuto per l'apprendimento che riassume gli aspetti principali dell'architettura e la loro risoluzione nel progetto. È un riepilogo delle idee di progettazione più significative all'interno del sistema e delle loro motivazioni.
Modello dei Dati	Comprende gli schemi della base di dati e le strategie di mapping tra la rappresentazione a oggetti e la base di dati.
Storyboard dei casi d'uso, Prototipi UI	Una descrizione dell'interfaccia utente, della navigazione, dei modelli di usabilità e così via.

In questa fase, vengono raffinati il modello dei casi d'uso, cominciati gli SSD e i contratti, la visione, le specifiche supplementari, e il glossario.

Iterazioni

Ogni iterazione è un mini-progetto che comprende *pianificazione, analisi, progettazione, costruzione, integrazione, test e rilascio*. Le iterazioni possono sovrapporsi, facilitando lo sviluppo parallelo e il lavoro flessibile in grandi squadre.

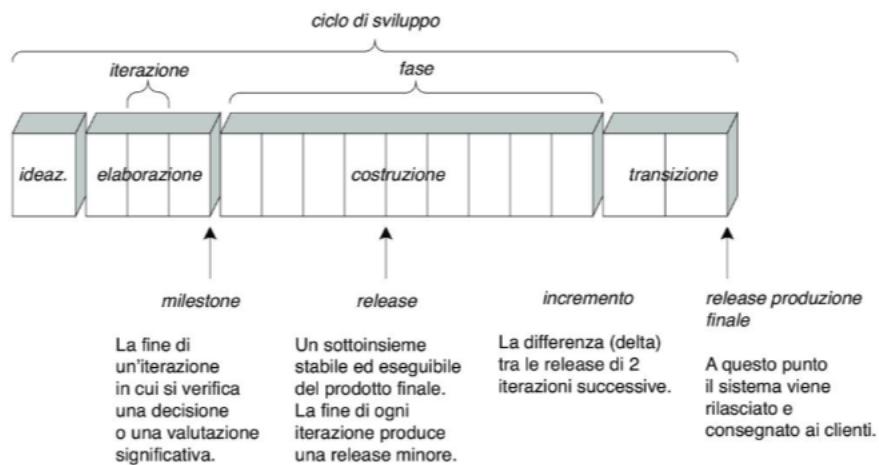
Anche se ogni iterazione può prevedere tutti i flussi di lavoro, la collocazione dell'iterazione all'interno del ciclo di vita del progetto determina una maggiore enfasi su uno dei flussi di lavoro.

Si noti che, nonostante l'iterazione includa **lavoro** nella maggior parte delle discipline, l'impegno relativo e l'enfasi cambiano nel tempo.



Ogni iterazione produce una **release**, che è un insieme di manufatti approvati e fornisce una base solida per le attività successive. Un incremento rappresenta il progresso tra una release e la successiva, avvicinandosi al rilascio finale del sistema. Le fasi del processo unificato includono:

- **Ideazione**: avvio del progetto con una visione approssimativa e stime iniziali
- **Elaborazione**: sviluppo del nucleo dell'architettura e risoluzione dei rischi maggiori
 - **Milestone**: la fine di un'iterazione in cui si verifica una decisione o una valutazione significativa
- **Costruzione**: realizzazione delle capacità operative e preparazione al rilascio
 - **Release**: un sottoinsieme stabile ed eseguibile del prodotto finale. La fine di ogni iterazione produce una release minore
 - **Incremento**: la differenza (Delta) tra le release di 2 iterazioni successive
- **Transizione**: completamento e rilascio del prodotto
 - **Release produzione finale**: a questo punto, il sistema viene rilasciato e consegnato ai clienti



In ogni iterazione si produce codice eseguibile, non solo durante la fase di ideazione, garantendo così un avanzamento tangibile e continuo.

Casi d'uso

I casi d'uso sono essenzialmente storie che ci aiutano a scoprire e documentare i requisiti di un sistema. Si tratta di **dialoghi scritti** che descrivono l'interazione tra un **attore**, che può essere una persona o un sistema, e il **sistema** stesso per compiere una determinata azione.

I casi d'uso sono un metodo diretto e semplice per descrivere i requisiti funzionali di un sistema. Sono *facilmente comprensibili dai clienti*, e permettono di coinvolgerli attivamente nella definizione e revisione dei requisiti. Evidenziano gli obiettivi e il **punto di vista degli utenti**, risultando utili nella creazione di guide per l'utente e nei test di sistema.

È importante notare che i casi d'uso non sono documenti orientati agli oggetti. Tuttavia, hanno un impatto significativo su diversi aspetti di un progetto, influenzando l'analisi e la progettazione orientata agli oggetti.

Il **modello dei casi d'uso** comprende tutti i casi d'uso scritti, e si concentra sulla narrazione piuttosto che sulla creazione di diagrammi; questo non toglie che può includere un diagramma UML, se necessario.

Definizione

La creazione di un caso d'uso inizia con la definizione di un obiettivo specifico dell'utente. Il nome assegnato al caso d'uso dovrebbe riflettere questo obiettivo e iniziare con un verbo per indicare un'azione.

Tuttavia, esiste un'eccezione per gli obiettivi **CRUD** (Create, Retrieve, Update, Delete), che vengono spesso raggruppati in un unico caso d'uso denominato “Gestisci X”.

Validità e utilità

Per valutare la validità di un caso d'uso, è essenziale considerare il *contesto*, i *confini del sistema*, gli *attori coinvolti* e gli *obiettivi da raggiungere*.

Ad esempio, “Negociare un o fornitore” potrebbe essere valido a un certo livello, mentre “Effettuare il login” potrebbe non essere considerato utile se non contribuisce significativamente agli obiettivi aziendali

I componenti

I casi d'uso si compongono dei seguenti componenti:

- **Attori:** sono entità con comportamento, come un cassiere o un *sistema di pagamento*
 - Il sistema in discussione stesso è considerato un attore quando interagisce con altri sistemi
 - **Attore primario:** utilizza il SuD per raggiungere obiettivi specifici, come un cassiere. Sono fondamentali per identificare gli obiettivi degli utenti
 - **Attore finale:** desidera l'uso del SuD per i propri fini, ad esempio un cliente. Aiuta a scoprire obiettivi che possono rivelare nuovi casi d'uso
 - **Attore di supporto:** fornisce servizi al SuD, come un sistema di autorizzazione dei pagamenti. Questo chiarisce le interfacce e i protocolli dei sistemi esterni
 - **Attore fuori scena:** ha interesse nel comportamento del SuD ma non interagisce direttamente con esso.
- **Scenario:** sequenza specifica di azioni e interazioni tra il sistema e gli attori, che può descrivere sia storie di successo che di fallimento
- **Caso d'uso:** insieme di scenari correlati che illustrano come un attore utilizza il sistema per raggiungere un obiettivo

I formati

I casi d'uso possono essere documentati in vari modi, a seconda del livello di dettaglio desiderato:

- **Formato breve:** un riepilogo conciso, solitamente limitato allo scenario principale di successo
- **Formato informale:** descrizioni informali che coprono vari scenari
- **Formato dettagliato:** descrizione completa di tutti i passaggi e le variazioni, inclusi dettagli come pre-condizioni e garanzie di successo. È opportuno usare una tabella di questo formato:

Selezione del caso d'uso	Descrizione
Nome del caso d'uso	Inizia con un verbo
Portata	Il sistema che si sta progettando
Livello	“Obiettivo utente” (l’utente finale desidera raggiungere attraverso il sistema”), “sotto funzione” (più dettagliato)
Attore primario	Nome dell’attore primario
Parti interessate e interessi	A chi interessa questo caso d’uso e che cosa desidera
Pre-condizioni	Che cosa deve essere vero all’inizio del caso d’uso
Garanzia di successo	Che cosa deve essere vero se il caso d’uso viene completato con successo
Scenario principale di successo	Uno scenario comune di attraversamento del caso d’uso, di successo e incondizionato
Estensioni	Scenari alternativi, di successo e di fallimento
Requisiti speciali	Requisiti non funzionali correlati
Elenco delle variabili tecnologiche e dei dati	Varianti nei metodi di I/O e nel formato dei dati
Frequenza di ripetizione	Frequenza prevista di esecuzione del caso d’uso
Varie	Altri aspetti, ad esempio i problemi aperti

Stilistica

Quando si scrivono casi d'uso, è preferibile adottare uno stile essenziale, focalizzandoci sugli obiettivi dell'attore piuttosto che sull'interfaccia utente. Forniamo due esempi:

- **Stile essenziale:** *l'amministratore si identifica, e il sistema verifica l'autenticità*
- **Stile concreto:** *l'amministratore inserisce ID e password, e il sistema fornisce l'accesso all'interfaccia di gestione utenti*

È importante scrivere in modo **conciso ma completo**, con una chiara indicazione di soggetto e verbo. È importante anche avere un **approccio a scatola nera**, dove bisogna definire i requisiti funzionali del sistema senza entrare nei dettagli di come questi saranno implementati. Bisogna, inoltre, **concentrarsi sugli utenti o attori**, comprendendo i loro obiettivi e le situazioni tipiche, e definire ciò che considerano un risultato di valore.

Identificare i casi d'uso

Per trovare i casi d'uso, possiamo seguire questi passaggi:

1. **Definire i confini del sistema:** comprendere ciò che è esterno ai confini del sistema è cruciale. *Nel caso di un sistema POS, il cassiere e il sistema di autorizzazione dei pagamenti sono fuori dai confini del sistema*
2. **Identificare gli attori primari:** un'alternativa per questo (+ identificazione di tutto il resto) è l'*analisi degli eventi* che interessano il sistema, determinando chi li genera e perché.

Evento esterno	Dall'attore	Obiettivo/Caso d'uso
inserire la riga di vendita per un articolo	Cassiere	elaborare una vendita
inserire il pagamento	Cassiere o Cliente	elaborare una vendita
...		

3. Determinare gli obiettivi per ogni attore primario
4. Sviluppare i casi d'uso che soddisfano questi obiettivi

Requisiti funzionali

I casi d'uso sono principalmente requisiti funzionali, o comportamentali, che specificano le azioni che il sistema deve essere in grado di eseguire. Possono essere anche associati ad altri tipi di requisiti, specialmente se questi sono strettamente legati a un caso d'uso specifico. Può essere per esempio *risposte che l'utente aspetta dal software in determinate condizioni, o risultati che il software deve produrre in risposta a specifici input.*

Requisiti non funzionali

Questi definiscono le **proprietà e i vincoli del sistema**, ad esempio *affidabilità e tempi di risposta*, ecc..... I requisiti non funzionali possono essere più critici dei requisiti funzionali; in caso contrario, il sistema potrebbe risultare inutilizzabile. I requisiti non funzionali possono influire sull'**architettura complessiva** di un sistema, piuttosto che sui singoli componenti.

Definiamo delle metriche esempio:

Proprietà	Misura
Velocità	Transazioni elaborate al secondo/tempi di risposta a utenti/tempo di refresh schermo
Dimensione	Mbytes/Numero di chip RIM
Facilità d'uso	Tempo di addestramento/numero di maschere di aiuto

Affidabilità	Tempo medio di malfunzionamento, probabilità di indisponibilità, tasso di malfunzionamento, disponibilità
Robustezza	Tempo per il riavvio dopo malfunzionamento, percentuali di eventi causanti malfunzionamento, probabilità di corruzione dati dopo malfunzionamento
Portabilità	Percentuali di dichiarazioni dipendenti dall'architettura di destinazione, numero di architetture di destinazione

Livelli

I casi d'uso possono essere scritti a diversi livelli di dettaglio:

- **Livello di obiettivo utente:** questo è il livello più rilevante nell'analisi dei requisiti
- **Livello di sotto-funzione:** utile per descrivere parti comuni tra diversi casi d'uso, o per dettagliare interazioni specifiche
- **Livello di sommario:** un caso d'uso a questo livello può includere più casi d'uso a livello di obiettivo utente, facilitando la loro identificazione e comprensione nel contesto più ampio

I test per la verifica

Esistono tre test principali per valutare l'utilità di un caso d'uso:

- **Test del capo:** modo per valutare l'impatto di un caso d'uso. Se un caso d'uso descrive un'attività che il tuo capo riconoscerebbe come preziosa, e degna di essere svolta per un'intera giornata lavorativa, allora **supera il test**. Aiuta a evitare di concentrarsi su dettagli troppo granulari che non contribuiscono significativamente agli obiettivi aziendali
- **Test EBP:** questo test si concentra sull'aggiunta di valore misurabile all'azienda. Un caso d'uso che supera il test EBP descrive un processo che è sia completo in sé stesso, sia capace di trasformare un input in un output di valore, senza necessitare di ulteriori passaggi per raggiungere uno stato stabile
- **Test della dimensione:** un caso d'uso deve essere sufficientemente ampio da coprire un processo completo, che spesso si traduce in una descrizione dettagliata di diverse pagine. Non si tratta solo di quantità, ma di assicurarsi che il caso d'uso descriva un flusso di lavoro completo che porta valore all'azienda

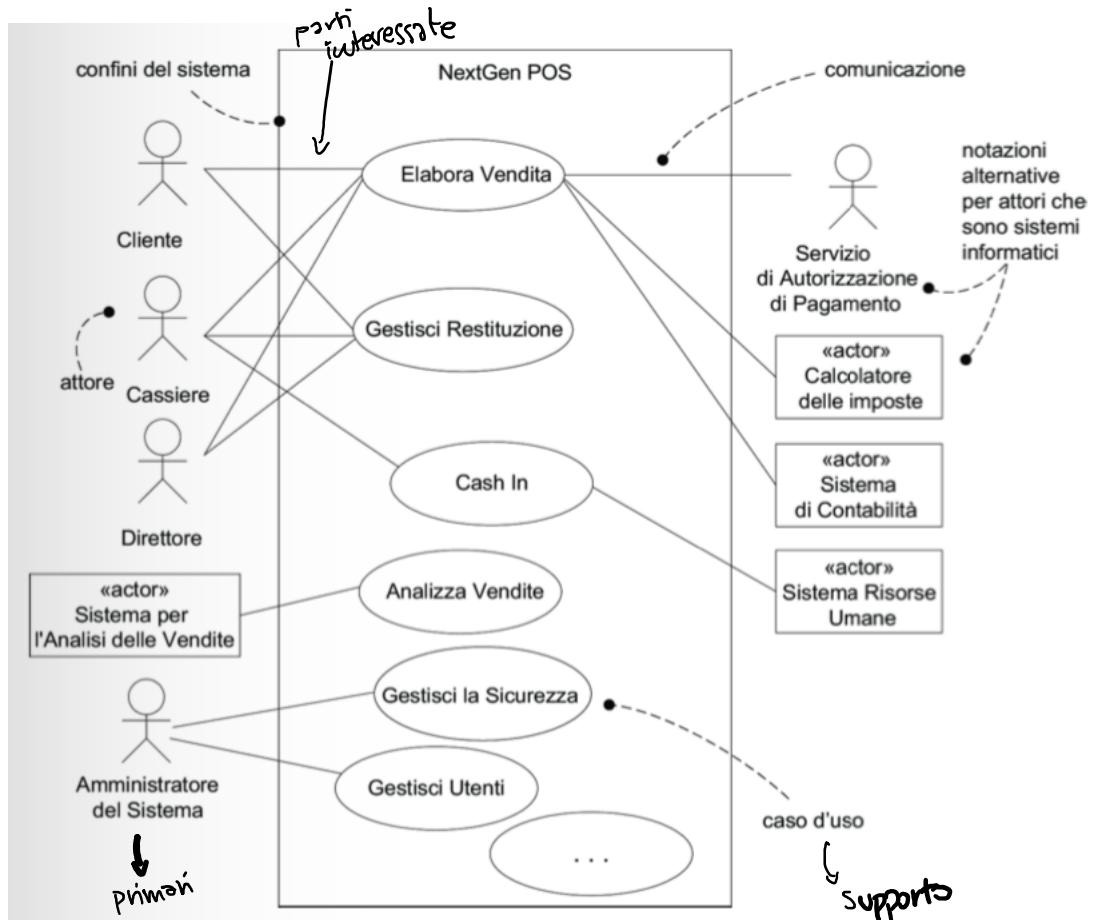
Violazioni ragionevoli dei test

In alcuni contesti, è accettabile deviare dai test standard per i casi d'uso, ad esempio:

- **Attività secondarie:** piccoli passaggi o attività di supporto possono essere elevati a casi d'uso per evitare ripetizioni eccessive nel testo

- **Login utente:** anche se il login potrebbe non superare il “test del capo”, per la sua apparente semplicità, può essere abbastanza complesso da richiedere un’analisi dettagliata, specialmente nel caso di funzionalità avanzate come il “single sign-on”

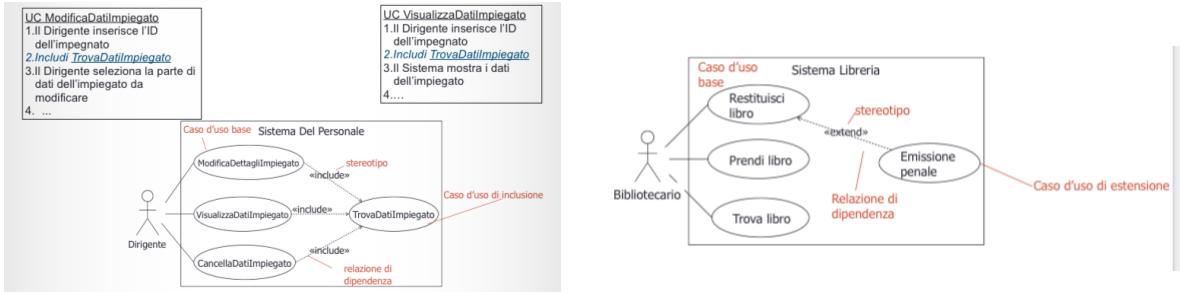
Esempio di diagramma



Associazioni e relazioni

Le associazioni definiscono i canali di comunicazione tra gli attori e i casi d’uso. Sono rappresentate con linee continue, e possono indicare la direzione dell’interazione, o bidirezionali. Le relazioni possono essere di tre tipi:

- **Include:** un caso d’uso base può includere un altro caso d’uso. L’inclusione viene eseguita completamente quando si raggiunge il punto specifico nel caso d’uso base
- **Extend:** questa relazione permette di aggiungere varianti a un caso d’uso base. L’estensione viene eseguita solo se la condizione di estensione è vera e solo in corrispondenza degli extension points
- **Generalization:** una relazione di generalizzazione permette di creare una gerarchia tra i casi d’uso, dove un caso d’uso più generale può essere specializzato in casi d’uso più specifici.

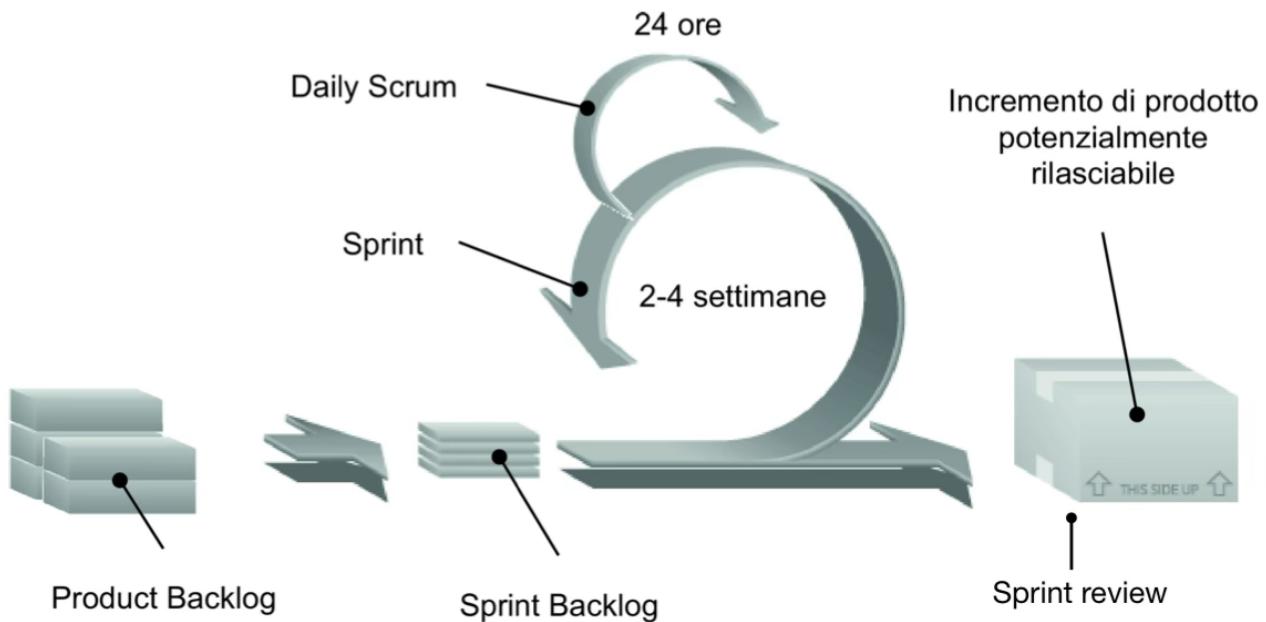


Misconcezioni comuni

(Non so perché le mette, ma bisogna perché se no il prof si arrabbia) Non si ha una piena comprensione dello sviluppo iterativo o dell'UP se si **cade in queste trappole**:

- Tentare di definire la maggior parte dei requisiti prima di iniziare la progettazione o l'implementazione
- Cercare di completare gran parte del progetto prima di iniziare l'implementazione effettiva
- Definire un'architettura completa e affidarsi ad essa prima di iniziare la programmazione e i test interattivi
- Pensare che i diagrammi UML e la progettazione debbano dettagliare completamente il progetto, riducendo la programmazione a una mera traduzione meccanica
- Equivalere l'ideazione ai requisiti, l'elaborazione alla progettazione, e la costruzione all'implementazione
- Ritenere che tre mesi sia una durata adeguata per un'iterazione, piuttosto che tre settimane
- Credere che adottare UP significhi eseguire il maggior numero possibile di attività e produrre numerosi elaborati
- Tentare di pianificare il progetto in modo speculativo e dettagliato dall'inizio alla fine

SCRUM



SCRUM è uno dei metodi più diffusi nell’ambito dello sviluppo agile di prodotti software. Si basa sul principio di fornire ai clienti il **massimo valore possibile in tempi brevi** adattandosi ai cambiamenti e alle esigenze emergenti. Non è un metodo completo, ma si focalizza sull’organizzazione del lavoro e sulla gestione dei progetti, lasciando spazio alla combinazione con altre pratiche e strumenti.

Abbiamo una serie di termini specifici importanti per SCRUM:

- **Scrum**: incontro giornaliero del team che esamina i progressi, le priorità del lavoro in quel giorno. Deve essere un breve incontro faccia a faccia per la squadra
- **ScrumMaster**: è il responsabile di assicurare che il processo sia seguito, e guida il team nell’uso efficace di questo metodo. È il responsabile dell’interfacciamento con il resto dell’azienda, e di assicurare che il team SCRUM non venga deviato da interferenze esterne.
- **Sprint**: iterazione di sviluppo, di solito lunga 2-4 settimane
- **Velocity**: stima di quanto lavoro rimanente un team può fare in un singolo sprint. Capire la velocità della squadra aiuta a stimare ciò che può essere coperto in uno sprint, e fornisce una base per misurare il miglioramento delle prestazioni.
- **Team di sviluppo**: è un gruppo auto-organizzatore di sviluppatori software, di solito meno di 7 persone. Sono i responsabili di sviluppo del software, e di altri documenti essenziali
- **Incremento potenzialmente rilasciabile**: è un incremento software fornito da altro software. Ciò dovrebbe essere potenzialmente trasportabile, e ciò significa che si trova

in uno stato finito, e non è necessario alcun lavoro ulteriore come il test per incorporarlo nel prodotto finale

- **Product backlog:** elenco di elementi “da fare”, quali definizioni di caratteristiche, requisiti software, storie degli utenti, e descrizioni di compiti supplementari necessari
- **Product owner:** individuo/piccolo gruppo il cui compito è di identificare caratteristiche o requisiti del prodotto, dare la priorità a questi per lo sviluppo, rivedere continuamente il product backlog per garantire che il progetto continui a soddisfare le esigenze di business critiche.

Analisi orientata agli oggetti

Nell’ambito dello sviluppo software, l’analisi orientata agli oggetti è un processo che si concentra su tre aree principali:

- **Dominio informativo:** rappresentato attraverso un modello di dominio, che illustra le classi concettuali e le relazioni tra di loro. Questo modello serve come fonte di ispirazioni per le classi di progetto e di implementazione, e viene sviluppato iterativamente, basandosi sui requisiti specifici di ogni iterazione.
- **Funzioni del sistema:** descritte dalle operazioni che il sistema deve eseguire, e l’ordine in cui queste operazioni possono essere richieste è rappresentato tramite diagrammi di sequenza di sistema
- **Comportamento del sistema:** definito dall’effetto che l’esecuzione di ogni operazione ha sulle informazioni contenute nel modello di dominio. Questi effetti sono dettagliati nei contratti delle operazioni di sistema, che descrivono come le operazioni modifichino lo stato del sistema.

Riguarda l’investigazione e la comprensione del problema che il sistema deve risolvere; è basata sull’**identificazione dei concetti nel dominio del problema**. L’analisi modella 3 aspetti di un sistema: *le informazioni da gestire* (modello di dominio), *le funzioni* (diagrammi di sequenza di sistema), *il comportamento* (contratti).

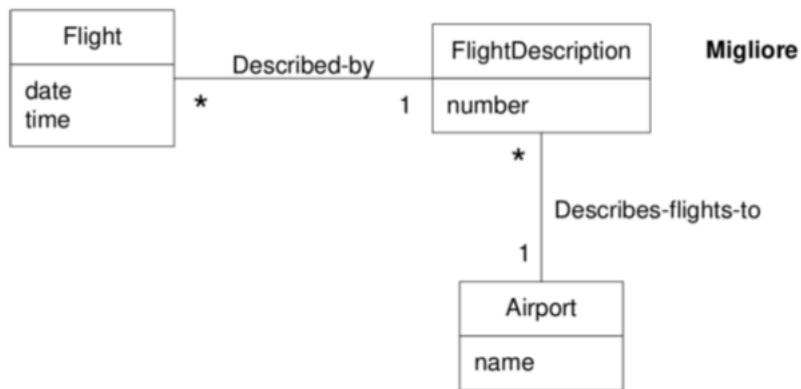
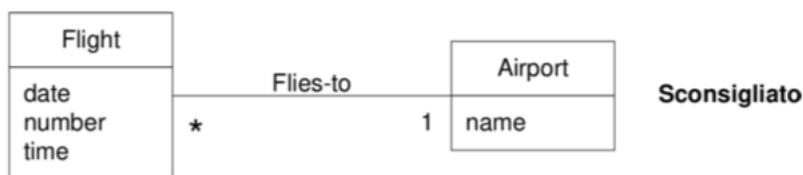
Modello di dominio

È essenzialmente un dizionario visuale che mostra le **astrazioni significative**, la terminologia e il contenuto informativo del dominio di interesse. Utilizzando **UML**, un modello di dominio è realizzato come uno o più diagrammi delle classi, che includono classi concettuali, associazioni e attributi, ma non operazioni o responsabilità software.

Si differenzia dal modello dei dati in quanto quest’ultimo si concentra sui dati che devono essere memorizzati in modo persistente, mentre il modello di dominio descrive le informazioni che devono essere gestite dal sistema, inclusi elementi che possono avere un ruolo comportamentale piuttosto che informativo.

Creare un modello di dominio è un passo essenziale nell'analisi orientata agli oggetti, perché fornisce una **comprendizione approfondita del sistema** da sviluppare e del suo vocabolario. Serve quindi come base per lo strato del dominio, e aiuta a visualizzare le informazioni che il sistema deve gestire. Si seguono questi passaggi:

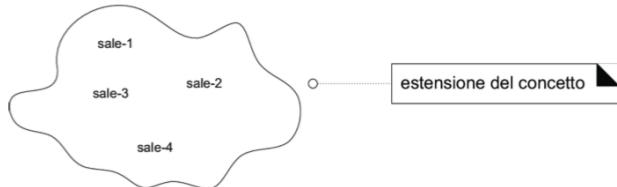
1. Identificare le classi concettuali pertinenti
 - a. Si possono riutilizzare modelli esistenti, usare categorie comuni, o individuare nomi e frasi nominali nei requisiti.
 - b. Importante utilizzare **termini specifici** del dominio, e includere solo elementi rilevanti per l'interazione corrente, evitando di aggiungere dettagli non necessari
2. Rappresentarle come classi in un diagramma delle classi UML
 - a. Le **classi descrizione** sono utili per rappresentare informazioni che descrivono altri oggetti, come le specifiche di un prodotto. Queste classi aiutano a *ridurre la ridondanza*, e a prevenire errori dovuti a informazioni duplicati
3. Aggiungere associazioni e attributi rilevanti



Classi concettuali

Le classi concettuali sono idee, cose o oggetti che possono essere rappresentati simbolicamente, definiti in termini di intenzione e estensione. Può essere usata in termini di:

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



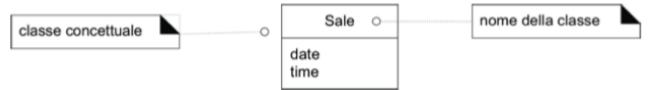
Nelle classi concettuali non bisogna considerare oggetti software. Per crearne, è utile usare una tabella del genere:

- Transazioni commerciali
- Elementi/righe di transazioni
- Prodotto o servizio legato a transazione
- Dove viene registrata la transazione?
- Ruoli di persone o organizzazioni correlati alle transazioni
- Luogo della transazione, luogo del servizio
- Eventi significativi spesso con un ora o un luogo che è necessario ricordare
- Oggetti fisici
- Descrizioni di oggetti
- Cataloghi
- Descrizioni di oggetti
- Cataloghi
- Contenitori di oggetti fisici e informazioni
- Oggetti in un contenitore
- Altri sistemi che collaborano
- Registrazioni di questioni finanziarie, di lavoro, contrattuali e legali
- Strumenti finanziari
- Piani, manuali, documenti cui si fa regolarmente riferimento per eseguire il lavoro

Vengono generate durante l'analisi, e poi inserite all'interno del modello di dominio

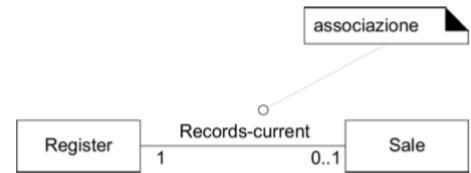
Classi

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 simili.



Associazioni

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 (istanze di queste classi) che indica una connessione significativa e interessante.

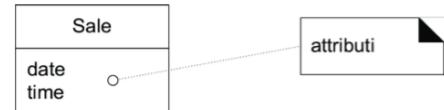


Un elenco di **associazioni comuni** è:

- A è una transazione correlata a un'altra transazione B
- A è un elemento/riga di una transazione B
- A è un prodotto/serivzio per una transazione B
- A è un ruolo relativo a una transazione B
- A è una parte fisica o logica di B
- A è contenuto fisicamente o logicamente in B
- A è una descrizione per B
- A è un membro di B
- A è una sottounità organizzativa di B
- A utilizza/gestisce/possiede B
- A è vicino/prossimo a B

Attributi

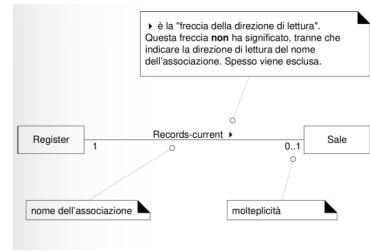
In UML, un attributo è la descrizione di una proprietà di una classe. Un attributo è un valore logico (ovvero una proprietà elementare) degli oggetti di una classe.



Associazione

Indicano relazioni significative tra classi, e dovrebbero essere incluse quando la conoscenza della relazione è importante per il sistema. È importante però non sovraccaricare il modello con troppe associazioni.

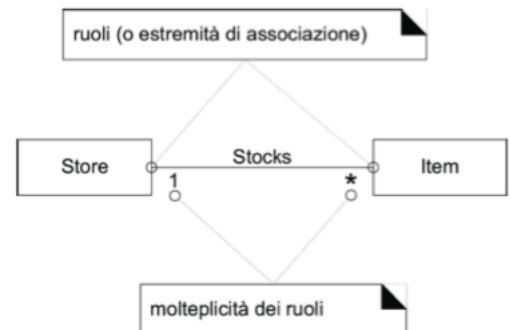
Bisogna mostrare associazioni solitamente quelle che implicano la conoscenza di una relazione che deve essere memorizzata per una certa durata di tempo, o associazioni identificate mediante l'elenco di associazioni comuni.



Ruoli

Ogni estremità di un'associazione è definita come un ruolo. I ruoli possono avere diverse caratteristiche:

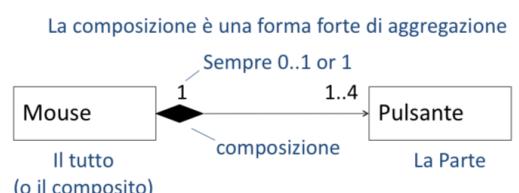
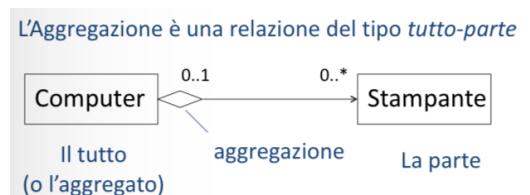
- **Moltività:** indica il numero di istanze di una classe che possono essere collegate a un'istanza di un'altra classe. Dipende dall'interesse che si ha nell'utilizzo del modello.
 - ***:** zero o più
 - **1..***: uno o più
 - **1..n**: da uno a n
 - **N**: esattamente n
 - **X,y,z**: esattamente x, y o z
- **Nome:** fornisce un'identificazione semantica al ruolo all'interno dell'associazione
- **Navigabilità:** determina se la relazione tra le classi può essere percorsa in una direzione specifica



Aggregazione e composizione

Queste due tipi di associazioni rappresentano relazioni “intero-parte” tra oggetti.

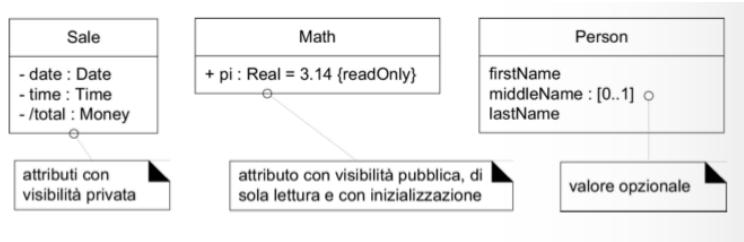
- **Aggregazione:** una relazione in cui le parti possono esistere indipendentemente dall'intero. Ad esempio, un *computer* può essere collegato a zero o più *stampanti* (le parti), e le stampanti possono esistere senza essere collegate a un computer.
- **Composizione:** una forma più forte di aggregazione in cui le parti sono dipendenti dall'intero per la loro esistenza. Se l'*intero* viene distrutto, anche le *parti* devono essere distrutte o trasferite. Ad esempio, un *mouse* (intero) e i suoi pulsanti (parti).



Attributi

Gli attributi sono una proprietà elementari degli oggetti di una classe, a cui **ogni oggetto associa un valore**.

Nel modello di dominio, gli attributi sono inclusi quando i requisiti indicano la necessità di memorizzare informazioni specifiche.



- **Notazione +:** attributo con visibilità pubblica, di solita lettura e con inizializzazione
- **[0..1]:** valore opzionale
- **Notazione -::** attributi con visibilità privata
- **Notazione /:** attributo derivato dal valore della molteplicità

Devono essere tipi di dato, che sono insiemi di valori dove l'identità unica non è significativa. Se si ha un dubbio su un particolare elemento, è consigliabile *creare una nuova classe concettuale* e stabilire un'associazione con essa. Questo approccio aiuta a mantenere il modello chiaro, e a evitare ambiguità.

Classi tipo di dati

Sono utilizzate nel modello di dominio per rappresentare elementi che potrebbero sembrare semplici valori, ma che in realtà hanno una struttura più complessa, o richiedono operazioni specifiche, come il parsing o la convalida.

Quando si modella con classi tipo di dati, è importante non includere attributi che *non sono necessari*, o che *non aggiungono valore al modello di dominio*. L'obiettivo è di rappresentare le informazioni in modo che riflettano accuratamente il dominio di interesse, senza sovraccaricare il modello con dettagli superflui.

Diagrammi di sequenza di sistema

Nel contesto dell'analisi dei sistemi, i casi d'uso svolgono un ruolo fondamentale descrivendo l'interazione tra gli *attori esterni* e *il sistema stesso*. Gli attori, attraverso la generazione di **eventi di sistema**, innescano l'esecuzione di determinate **operazioni di sistema**. Queste operazioni sono essenziali per il sistema, che deve essere in grado di gestirle efficacemente.

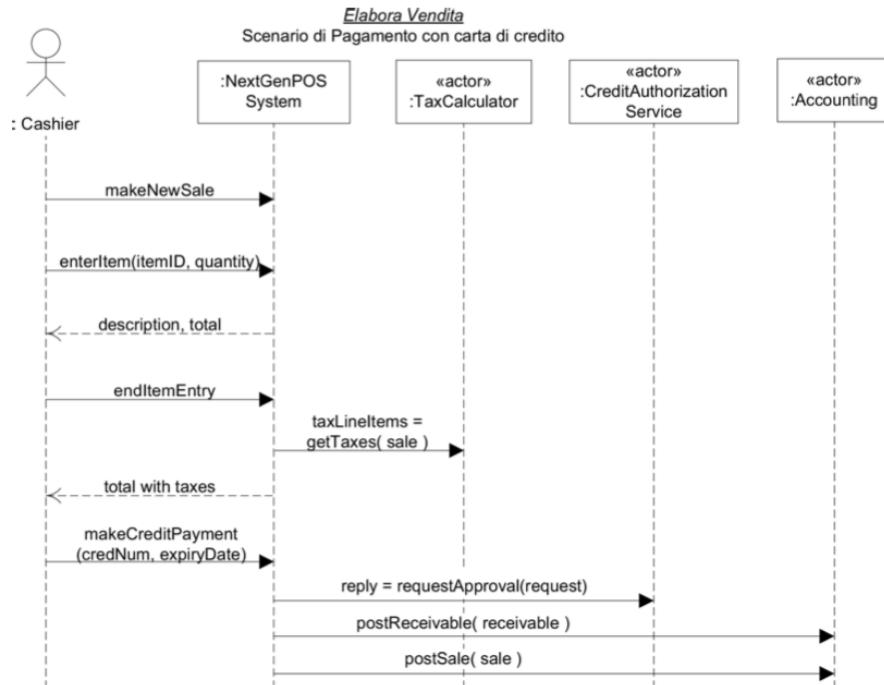
Per illustrare le interazioni tra attori e operazioni, UML utilizza i diagrammi di sequenza (SSD), che **mostra gli eventi generati dagli attori esterni**, e l'ordine in cui si verificano, nonché gli eventi che avvengono tra diversi sistemi.

È consigliabile disegnare un SSD per lo scenario principale di successo di ogni caso d'uso, così come per gli scenari alternativi più frequenti o complessi. Questo approccio enfatizza l'applicazione degli elementi UML ai sistemi, considerati come "scatole nere".

Sono strumenti cruciali perché *il software deve essere progettato per gestire gli eventi di sistema*. Un sistema software tipicamente reagisce a eventi esterni da attori, eventi temporali e guasti, o eccezioni, che spesso provengono da fonti esterne.

Gli SSD sono strumenti visivi che **illustrano gli eventi di input e output** che si verificano durante l'interazione tra attori esterni e il sistema in esame. In particolare, per un dato flusso di eventi all'interno di un caso d'uso, gli SSD mostrano:

- **Attori esterni**: individui o sistemi che interagiscono direttamente con il sistema
- **Sistema**: rappresentato come una scatola nera, ovvero senza dettagli interni
- **Eventi di sistema**: azioni generate dagli attori che influenzano il sistema



Gli elementi all'interno di un SSD sono **concisi**, e bisogna usare invece il *Glossario* per descrivere questi termini in maggior dettaglio.

Evento

Un evento è un'occorrenza significativa che si verifica durante l'esecuzione di un sistema.

Un **evento di sistema** è un evento esterno, spesso scatenato da un attore per interagire con il sistema.

Gli attori generano eventi di sistema per sollecitare l'esecuzione di operazioni specifiche.

Operazione

Un'operazione in UML rappresenta una trasformazione o una query che può essere eseguita da un oggetto, o componente del sistema.

Le operazioni generate da eventi di sistema sono **operazioni di sistema**, e sono funzioni che il sistema deve eseguire per rispondere agli eventi. È una *trasformazione, un'interrogazione che un soggetto o componente può essere chiamato ad eseguire.*

Un'operazione di sistema è un'azione pubblica che **il sistema è tenuto a eseguire in risposta a un evento di sistema**: queste operazioni hanno l'effetto di *modificare lo stato del sistema*, che è rappresentato dagli oggetti nel modello di dominio.

L'**interfaccia del sistema** è l'insieme di tutte le operazioni di sistema disponibili.

In UML, un'operazione è definita come la specifica di una trasformazione, o interrogazione, che un oggetto può essere chiamato ad eseguire.

Un metodo è l'**implementazione concreta di un'operazione**; ogni operazione ha una *firma* e un *insieme di vincoli*, classificati come pre-condizioni e post-condizioni, che ne specificano la semantica.

L'**Object Constraint Language (OCL)** è un linguaggio formale utilizzato in UML per esprimere vincoli. In teoria, le pre-condizioni e le post-condizioni delle operazioni possono essere espresse in OCL, fornendo una descrizione precisa e formale dei requisiti.

Contratti e post-condizioni

Un contratto è specificatamente legato a un'operazione di sistema, e **descrive come l'esecuzione di tale operazione influenzi lo stato** degli oggetti nel modello di dominio.

Le post-condizioni dettagliano i **cambiamenti nello stato degli oggetti**, senza però descrivere le azioni eseguite durante l'operazione. Questi cambiamenti possono includere la *creazione o cancellazione di un oggetto, la modifica di un valore di attributo, o la formazione o rottura di un collegamento*.

Le post-condizioni sono fondamentali perché delineano i cambiamenti necessari a seguito dell'esecuzione dell'operazione di sistema, *senza specificare il processo per realizzare tali cambiamenti*.

Il cambiamento dello stato del sistema dovuto all'esecuzione di un'operazione di sistema deve essere espresso in **termini di oggetti**, collegamenti e attributi nel dominio di interesse.

Durante la definizione dei contratti, può emergere la necessità di *aggiungere nuove classi concettuali, attributi o associazioni al modello di dominio*. È un processo normale nei metodi iterativi ed evolutivi, dove tutti gli elementi dell'analisi e della progettazione sono considerati lavori in corso, soggetti a miglioramenti continui.

I contratti sono utili per **rappresentare situazioni dettagliate o complesse**, che non sono facilmente descritte nei casi d'uso, fungendo da complemento a questi ultimi.

I contratti delle operazioni possono essere utilizzati per descrivere le operazioni di sistema all'interno del Modello dei casi d'uso. Non sono richiesi durante la fase iniziale di ideazione, ma vengono redatti principalmente durante la fase di elaborazione, in particolare per le operazioni più complesse.

Creazione di contratti

La creazione di contratti inizia con l'identificazione delle operazioni di sistema dai **Diagrammi di Sequenza di Sistema (SSD)**. Questo processo è fondamentale per definire le azioni che il sistema deve eseguire. Una volta identificate le operazioni, si procede alla **creazione di contratti** per quelle operazioni che risultano più complesse, o non sono chiaramente definite nei casi d'uso.

Abbiamo vari sezioni in un contratto:

- **Operazione:** nome e parametri (firma) dell'operazione
- **Riferimenti:** casi d'uso in cui può verificarsi questa operazione
- **Pre-condizioni:** ipotesi significative sullo stato del sistema o degli oggetti nel Modello di Dominio prima dell'esecuzione dell'operazione. Si tratta di *ipotesi non banali*, che dovrebbero essere comunicate al lettore.
- **Post-condizioni:** descrive i cambiamenti di stato degli oggetti nel Modello di Dominio dopo il completamento dell'operazione

Post-condizioni

Le post-condizioni sono utilizzate per descrivere i cambiamenti di stato degli oggetti nel modello di dominio. Questi cambiamenti possono essere raggruppati in tre categorie principali:

- **Creazione o cancellazione di un oggetto:** indica l'inizio o la fine dell'esistenza di un oggetto nel sistema
- **Modifica di un attributo:** riflette un cambiamento nel valore di un attributo di un oggetto
- **Formazione o rottura di un collegamento:** si riferisce alla creazione o alla distruzione di una relazione tra oggetti

È importante formare i collegamenti necessari tra gli oggetti esistenti e quelli appena creati. Le pre-condizioni, di solito, sono implicite dall'esecuzione di operazioni di sistema precedenti, e sono utili per indicare quali oggetti sono noti al sistema.

Funzioni e comportamento

Per descrivere le funzioni e il comportamento di un sistema senza entrare nei dettagli tecnici, si utilizzano modelli come **casi d'uso**, **SSD**, e **contratti delle operazioni di sistema**. Gli eventi e le operazioni di sistema devono essere nominati a un livello astratto, iniziando con un verbo che esprime l'intenzione, piuttosto che l'azione.

Gli elementi presentati negli SSD sono concisi, e spesso richiedono ulteriori spiegazioni. Un glossario può fornire dettagli più approfonditi su questi termini.

Creazione e utilizzo

Gli SSD vengono creati con lo scopo di **comprendere meglio il sistema**, e non sono necessari per tutti gli scenari, ma solo per quelli rilevanti nell'interazione corrente. Essi fanno parte del modello dei casi d'uso, e visualizzano le interazioni che questi implicano, contribuendo significativamente alla fase di elaborazione del progetto.

Contratti di sistema

Il comportamento di un sistema informatico è delineato dai **casi d'uso** e dai **requisiti funzionali**. Per approfondire, i contratti delle operazioni di sistema offrono una descrizione dettagliata del comportamento del sistema, utilizzando *pre-condizioni* e *post-condizioni* per illustrare le modifiche agli oggetti all'interno di un modello di dominio.

Architettura logica

L'architettura logica di un sistema software è l'organizzazione su larga scala delle classi software in **package**, **sottoinsiemi** e **strati**.

Uno stile comune è l'architettura a strati, dove uno strato è un **gruppo a grana molto grossa** di classi, package o sottoinsiemi che ha delle **responsabilità coese** rispetto a un aspetto importante del sistema. *Gli strati più alti ricorrono ai servizi degli strati più bassi.*

Normalmente, gli strati comprendono: *presentazione/interfaccia utente, logica applicativa/strato del dominio, servizi tecnici*.

Questo tipo di architettura è **stretta** se uno strato può solo richiamare i servizi dello strato immediatamente sottostante; al contrario, è detta **rilassata** se uno strato più alto può richiamare i servizi di strati più bassi di diversi livelli.

Architettura software

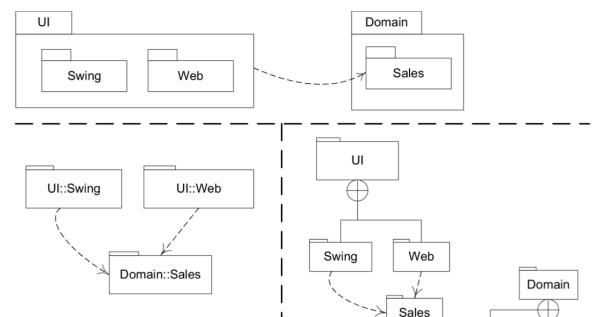
Una architettura software può essere diverse cose:

- L'insieme delle decisioni significative sull'organizzazione di un sistema software
- La scelta degli elementi strutturali da cui è composto il sistema e delle relative interfacce, insieme al loro comportamento specificato dalla collaborazione tra questi elementi
- La composizione di questi elementi strutturali e comportamentali in sottosistemi via via più ampi, e lo stile architettonico che guida questa organizzazione.

Diagrammi dei package

L'architettura logica può essere illustrata mediante un **diagramma dei package di UML**; uno strato può essere modellato come un package UML.

Un package UML può raggruppare qualunque cosa; è molto comune l'**annidamento di essi**; per mostrare la dipendenza tra essi viene utilizzata una **dipendenza** UML. Un package rappresenta un **namespace**, cosicché è possibile definire due classi con lo stesso nome su package diversi.



Progettazione degli strati

È importante organizzare la struttura logica di un sistema in strati separati con responsabilità distinte e correlate, con una **separazione netta** e coesa degli interessi:

- Gli **strati inferiori** sono servizi generali e di basso livello
- Gli **strati superiori** sono più specifici per l'applicazione

Collaborazioni e accoppiamenti vanno dagli strati più alti a quelli più bassi; l'obiettivo è la **suddivisione di un sistema in un insieme di elementi software** che, per quanto possibile, possano essere sviluppati e modificati ciascuno indipendentemente dagli altri.

Abbiamo tre definizioni:

- **Livello** (tier): solitamente indica un nodo fisico di elaborazione
- **Strato** (layer): una sezione verticale dell'architettura
- **Partizione** (partition): una divisione orizzontale di sotto sistemi di uno sotraso

Soltanente, gli strati possono essere divisi in questo modo:

Package	Specifico
UI (Presentation, View)	Finestre GUI, report, interfaccia vocale, WinForms, SwiftUI, etc.
Application (workflow, process, mediation, app controller)	Gestisce richieste dallo strato UI Workflow, stato sessioni, transizioni finestre/pagine, consolidamento/trasformazione di dati disparati per la presentazione
Domain (Business, Application Logic, Model)	Gestisce richieste dallo strato Application Implementazione regole di dominio, servizi di dominio (POS, Inventory) – possono essere usati da una sola applicazione ma sono possibili anche servizi di interesse per più applicazioni
Business Infrastructure	Servizi aziendali molto generali e di basso livello. Es. <i>CurrencyConverter</i>
Technical Services	Servizi tecnici e framework di livello alto, Persistence, Security
Foundation (Core Services, Base Services)	Servizi tecnici, utilità e framework di basso livello. Strutture dati, thread, matematica, file, database, I/O di rete

In uno strato, le responsabilità degli oggetti devono essere **fortemente coese** l'uno all'altro, e non devono essere mescolate con le responsabilità degli altri strati.

Logica applicativa ed oggetti

È importante creare oggetti software con nomi e informazioni simili al dominio del mondo reale, e assegnare a essi responsabilità della logica applicativa. Un oggetto software di questo tipo è chiamato **oggetto di dominio**: rappresenta una cosa nello spazio del dominio del problema, e ha una logica applicativa o di business correlata.

Progettando gli oggetti in questo modo si arriva a uno strato della logica applicativa che è chiamato **strato del dominio dell'architettura**. Gli oggetti di dominio (modello) non devono essere connessi o accoppiati direttamente agli oggetti UI (vista), e non mettere la logica applicativa nei metodi di un oggetto è l'interfaccia utente. Gli oggetti UI dovrebbero solo inizializzare gli elementi dell'interfaccia utente, ricevere eventi UI, e delegare le richieste di logica applicativi agli oggetti non UI.

Un rilassamento legittimo di questo principio è il **pattern observer**: gli oggetti del dominio inviano messaggi a oggetti della UI, visti però solo indirettamente, in termini di un'interfaccia come *PropertyListener*. L'oggetto di dominio non sa che quell'oggetto della UI è un oggetto UI. I messaggi inviati dallo strato UI allo strato del dominio sono i *messaggi mostrati negli SSD*.

Strato di dominio

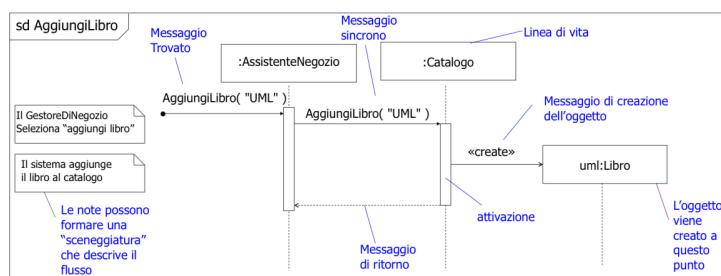
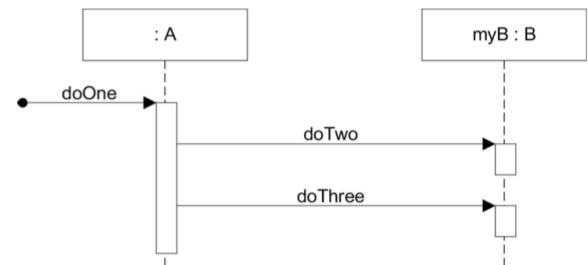
Lo strato di dominio contiene oggetti di dominio per gestire la logica applicativa.

Diagrammi di interazione

Permettono di modellare il comportamento dinamico di un sistema, visualizzando le interazioni tra diverse entità, come oggetti o componenti, e come queste collaborano per realizzare una funzionalità. I diagrammi includono principalmente:

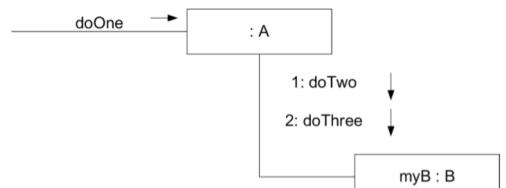
- **Diagrammi di sequenza:** mostrano l'ordine temporale degli eventi che coinvolgono le entità

- Enfatizzano la sequenza temporale degli scambi di messaggi
- Mostrano interazioni ordinate in una sequenza temporale
- Non mostrano le relazioni degli oggetti (possono essere dedotte dagli invii di messaggi)
- Mostra chiaramente la sequenza o l'ordinamento temporale dei messaggi. Inoltre, le opzioni di notazione sono numerose e dettagliate
- Costringe a estendersi verso destra quando si aggiungono nuovi oggetti; consuma quindi lo spazio orizzontale
- Abbiamo i seguenti **componenti**:



- **Diagrammi di comunicazione:** evidenziano le relazioni e gli scambi di messaggi tra entità

- Enfatizzano le relazioni strutturali tra gli oggetti
- Sono utilizzati per esplicitare le relazioni degli oggetti
- Economizza lo spazio, ed è più flessibile nell'aggiunta di nuovi oggetti nelle due dimensioni



- È più difficile vedere la sequenza dei messaggi, e ci sono meno opzioni di notazione
- **Diagrammi di interazione generale**
 - Illustrano come il comportamento complesso è realizzato da una serie di interazioni più semplici
- **Diagrammi di temporarizzazione**
 - Enfatizzano gli aspetti in tempo reale di una interazione

Questi diagrammi sono utili per capire il flusso di controllo e di dati all'interno del sistema, specificare i requisiti di dettaglio per la realizzazione di casi d'uso, e documentare e comunicare il comportamento previsto del sistema.

Interazioni

Un'interazione è una specifica di come alcuni oggetti si **scambiano messaggi** nel tempo per eseguire un compito nell'ambito di un certo contesto. I diagrammi di interazione come *linee di vita o messaggi*.

Linee di vita

Una linea di vita rappresenta un singolo partecipante a un'interazione, e raffigura come un'istanza del classificatore partecipa all'interazione.

Le linee di vita hanno:

- **Nome:** usato per far riferimento alle linee di vita nell'interazione
- **Tipo:** il nome del classificatore di cui rappresenta un'istanza
- **Selettore:** una condizione booleana che seleziona una specifica istanza

Abbiamo una certa notazione:

Notazione	Specifiche
:Sale	Linee di vita che rappresenta un'istanza senza nome
s : Sale	Linee di vita che rappresenta un'istanza con nome
<<metaclass>> Font	Linee di vita che rappresenta che Font è un'istanza della classe Class; un'istanza di una metaclasse
Linea tratteggiata	Sotto al rettangolo per un oggetto c'è una linea tratteggiata che indica proprio il tempo di vita dell'oggetto (sequenza)
No linea	La notazione per una linea di vita non comprende nessuna linea (comunicazione)
Sales: List<Sale>	Linea di vita che rappresenta un'istanza sales di una collezione di tipo List<Sale>, ovvero una lista
Sales[i] : Sale	Linea di vita che rappresenta una sola istanza della classe Sale, selezionata dalla collezione sales di tipo List<Sale>

x : MyInterface

Il tipo di un oggetto può essere anche un'interfaccia (o classe astratta)

Oggetti

(È tipo la terza volta che si ripete...) Un oggetto è un pacchetto coeso di dati e funzioni encapsulate in una unità riusabile. Ogni oggetto è un'**istanza di una classe** che definisce l'insieme comune di caratteristiche condivisi da tutte le sue istanze. Gli oggetti hanno *i valori degli attributi* (dati), e le *operazioni*. Tutti gli oggetti hanno *identità* (ogni oggetto ha il suo identificativo univoco), *stato* (viene stabilito dai valori effettivi dei dati memorizzati in un oggetto ad un dato istante), *comportamento* (l'insieme delle operazioni che l'oggetto può eseguire).

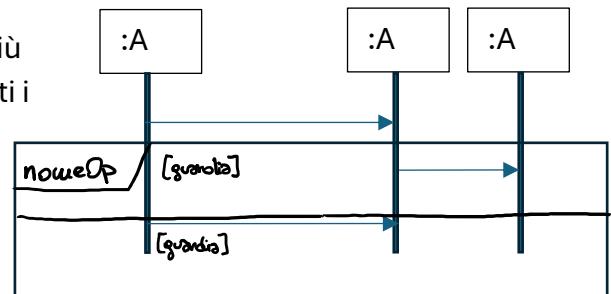
I dati sono nascosti all'interno dell'oggetto, e l'unico modo per accedere ai dati è attraverso le operazioni: questo è chiamato **incapsulamento**. Un oggetto **singleton** è indicato con un **1** in alto a destra.-

Messaggi

Gli oggetti collaborano per eseguire le funzioni del sistema stabilendo dei **collegamenti** tra loro e scambiandosi messaggi; i messaggi causano l'invocazione delle operazioni da parte dell'oggetto.

Frame

I diagrammi di sequenza possono essere divisi in aree chiamate **frammenti combinati**, i quali hanno uno o più operandi. L'operatore determina come vengono eseguiti i suoi operandi; le condizioni di guardia, quindi, stabiliscono **se** i loro operandi devono essere eseguiti. L'esecuzione occorre se la condizione di guardia è valutata vera.

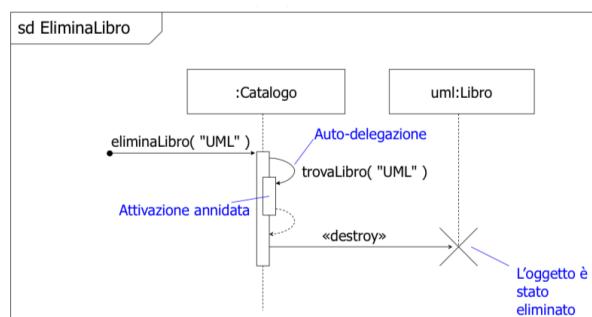


Abbiamo vari operatori che possono essere usati:

Operatore	Nome completo	Significato
Opt	Option	C'è un singolo operando che viene eseguito se la condizione è vera
Alt	Alternatives	Viene eseguito l'operando la cui condizione è vera Due o più operandi, ognuno protetto dalla propria condizione di guardia. Un operando viene eseguito se la sua condizione è true. L'operando opzionale con condizione [else] viene eseguito se nessuna delle altre condizioni è vera

Loop	Loop	<p>Sintassi speciale: loop min, max [condizione]</p> <p>Un ciclo senza max, min, o una condizione è un ciclo infinito. Se viene specificato solo min, allora si considera max=min</p> <p>La condizione può essere un'espressione booleana, o un testo purchè il suo significato sia chiaro.</p>																					
Break	Break	<p>Se la condizione di guardia è vera, viene eseguito l'operando, non il resto dell'interazione in cui è incluso</p> <p>Indica in quale condizione il loop viene interrotto. Logicamente è esterno al ciclo, e si disegna sovrapposto al loop</p> <table border="1"> <thead> <tr> <th>Tipo di ciclo</th> <th>Significato</th> <th>Espressione</th> </tr> </thead> <tbody> <tr> <td>While(true) {body}</td> <td>Continuare il ciclo per sempre</td> <td>Loop *</td> </tr> <tr> <td>For i = 1 to n {body}</td> <td>Ripetere n volte</td> <td>Loop n</td> </tr> <tr> <td>While(bool) {body}</td> <td>Ripetere mentre bool è vera</td> <td>Loop [bool]</td> </tr> <tr> <td>Repeat {body} while(bool)</td> <td>Eseguire una volta, poi ripetere mentre bool è vera</td> <td>Loop 1, * [bool]</td> </tr> <tr> <td>Foreach object in collection {body}</td> <td>Eseguire il corpo del ciclo una volta per ogni oggetto di una collezione</td> <td>Loop [for each object]</td> </tr> <tr> <td>Foreach object in class {body}</td> <td>Eseguire il corpo del ciclo una volta per ogni oggetto di una particolare classe</td> <td>Loop [for each object]</td> </tr> </tbody> </table>	Tipo di ciclo	Significato	Espressione	While(true) {body}	Continuare il ciclo per sempre	Loop *	For i = 1 to n {body}	Ripetere n volte	Loop n	While(bool) {body}	Ripetere mentre bool è vera	Loop [bool]	Repeat {body} while(bool)	Eseguire una volta, poi ripetere mentre bool è vera	Loop 1, * [bool]	Foreach object in collection {body}	Eseguire il corpo del ciclo una volta per ogni oggetto di una collezione	Loop [for each object]	Foreach object in class {body}	Eseguire il corpo del ciclo una volta per ogni oggetto di una particolare classe	Loop [for each object]
Tipo di ciclo	Significato	Espressione																					
While(true) {body}	Continuare il ciclo per sempre	Loop *																					
For i = 1 to n {body}	Ripetere n volte	Loop n																					
While(bool) {body}	Ripetere mentre bool è vera	Loop [bool]																					
Repeat {body} while(bool)	Eseguire una volta, poi ripetere mentre bool è vera	Loop 1, * [bool]																					
Foreach object in collection {body}	Eseguire il corpo del ciclo una volta per ogni oggetto di una collezione	Loop [for each object]																					
Foreach object in class {body}	Eseguire il corpo del ciclo una volta per ogni oggetto di una particolare classe	Loop [for each object]																					
Ref	Reference	Il frammento combinato fa riferimento a un'altra interazione (diagramma).																					
Par	Parallel	Tutti gli operandi sono eseguiti in parallelo																					
Seq	Weak sequencing	Tutti gli operandi vengono eseguiti in parallelo con il vincolo che gli eventi che arrivano sulla stessa linea di vita da operandi differenti avvengano nella stessa sequenza degli operandi																					
Strict	Strict sequencing	Gli operandi vengono eseguiti in sequenza stretta																					
Neg	Negative	L'operando mostra interazioni che non devono accadere																					
Critical	Critical region	L'operando viene eseguito atomicamente senza interruzione																					

La creazione di un'istanza viene applicata usando un messaggio «**create**». Possiamo avere, inoltre, anche un'**auto-delegazione**, ovvero quando una linea vita invia un messaggio a se stessa; questo genera una *attivazione annidata*. Per indicare la distruzione di un oggetto, si termina la linea di vita con una grossa croce e un messaggio «**destroy**».



Casella di azione

Una **casella di azione** può contenere istruzioni in un linguaggio arbitrario; è posizionata sopra la linea di vita che applica l'azione. È un rettangolo rotondo.

I++

Chiamate asincrone

Una freccia non piena indica una **chiamata asincrona**, diversamente da una freccia piena, che invece rappresenta una più comune chiamata sincrona.

Diagrammi di comunicazione

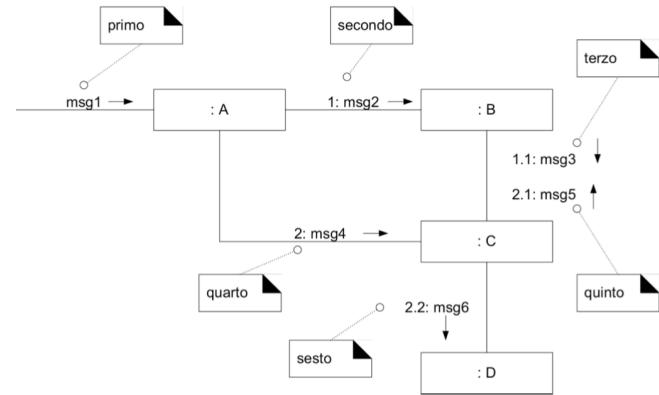
I diagrammi di comunicazione enfatizzano gli aspetti strutturali di un'interazione, mostrando **come si collegano le linee di vita**.

Un collegamento è un percorso di connessione tra due oggetti: mettiamo un *numero di sequenza*, in modo da ordinare quale azione viene fatta prima e quale dopo; successivamente, il messaggio e possibilmente anche il **verso**. Non si numera il primo messaggio, per semplificare la numerazione.

Se abbiamo una sequenza complessa, allora scriviamo **number.number** usando la dot notation.

Se abbiamo un **messaggio condizionale**, allora scriviamo la condizione tra [e].

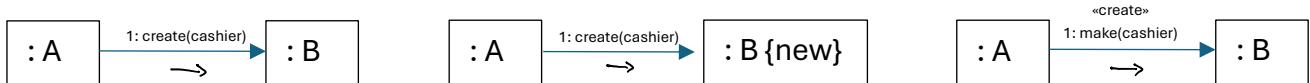
L'**iterazione** è mostrata dall'utilizzo dell'indicatore di iterazione (*), e un'espressione di iterazione. Se si vuole indicare che i messaggi sono tutti eseguiti in parallelo, si deve usare l'**indicatore parallelo di iterazione */**



Creazione di istanze

Abbiamo tre modi per mostrare la creazione in un diagramma di comunicazione; si usa, comunque, sempre un messaggio **create** con dei parametri opzionali di inizializzazione; questo verrà normalmente interpretato come una chiamata del costruttore.

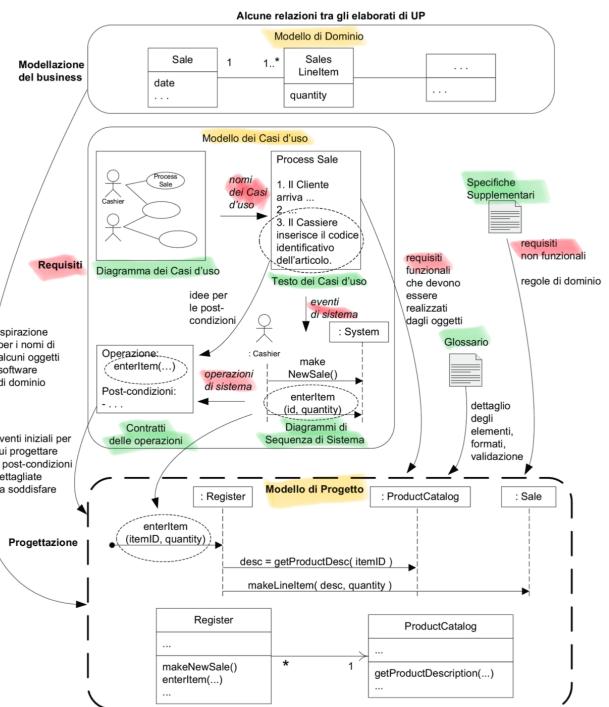
Può essere (senza la freccia sulla riga):



Progettazione ad oggetti

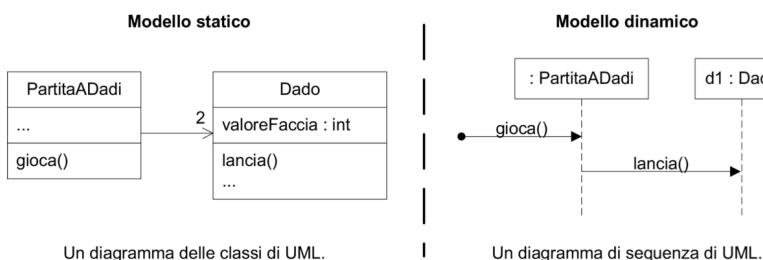
Gli sviluppatori usano tre tecniche per progettare ad oggetti:

- **Codifica:** progettare mentre si codifica, possibilmente insieme a TDD (Test-Driven Development) e refactoring
- **Disegno poi codifica:** disegnare alcuni diagrammi UML su una lavagna o con uno strumento CASE per UML, per poi passare alla codifica usando un IDE testuale
 - Prima della codifica, si dedicano, per esempio, per 3 settimane, *alcune ore o al massimo un giorno al disegno di UML*. Si passa alla codifica per il resto dell'iterazione; nel corso dell'iterazione possono esserci altre sessioni di disegno più brevi.
 - Prima di ogni sessione di modellazione successiva, si esegua il **reverse engineering**, e si faccia riferimento a questi diagrammi nel corso della sessione successiva.
- **Solo disegno:** in qualche modo, lo strumento genera ogni cosa dai diagrammi



Modellazione statica e dinamica

Ci sono due tipi di modelli per gli oggetti: *statici* e *dinamici*; è utile creare questi modelli in parallelo.



I modelli statici sono fatti nei *diagrammi delle classi*, e sono utili come sintesi e base per la struttura del codice. I modelli dinamici, invece, sono fatti nei *diagrammi di sequenza* e

comunicazione; sono i più importanti e difficili da creare, e si applicano la **progettazione guidata dalle responsabilità**, e i principi **GRASP**. Altra tecnica di progettazione oggetti sono le **schede CRC**, dove abbiamo tre parti: *Class*, *Responsibility* e *Collaboration*.

Input, output ed attività

Tra questi input troviamo i testi dei casi d'uso, le specifiche supplementari, il glossario, i diagrammi di sequenza di sistema, il modello di dominio e i contratti delle operazioni.

Possiamo anche adottare diversi approcci:

- **Progettare mentre si codifica**: prevede un ciclo di sviluppo agile, dove la progettazione avviene in tandem con la codifica, spesso preceduta da test
- **Modellazione UML**: aiuta a visualizzare e organizzare il sistema in maniera strutturata
- **Altre tecniche di modellazione**: possiamo anche iniziare con tecniche alternative, come le schede CRC (*Class-Responsibility-Collaborator*).

Gli output di questo processo includono:

- **Diagrammi UML**: diagrammi di interazione, classi e package che rappresentano la struttura e le interazioni all'interno del sistema
- **Interfacce utente**: prototipi e bozze che anticipano come gli utenti interagiranno con il sistema
- **Modelli di base di dati**: strutture che definiscono come i dati sono organizzati e gestiti nel sistema.

L'elemento più importante è l'**applicazione dei principi** di progettazione orientata agli oggetti, come i pattern **GRASP**, i design pattern del **Gang-Of-Four**, e la progettazione guidata dalle **responsabilità**.

Responsabilità e RDD

Con **RDD** intendiamo *Responsibility-Driven Design*, ed è un approccio che si concentra sulle responsabilità, i ruoli e le collaborazioni tra oggetti. Le responsabilità possono essere di *fare*, come eseguire calcoli o coordinare altri oggetti, o di *conoscere*, come gestire dati privati o relazioni con altri oggetti.

La traduzione delle responsabilità in classi e metodi è influenzata dalla granularità della responsabilità; quindi, **non è la stessa cosa di un metodo**, ma i metodi sono implementati per adempiere alle responsabilità. Inoltre, i metodi agiscono da soli, o collaborano con altri metodi e oggetti.

Il flusso di lavoro di **RDD** viene fatta iterativamente con:

- Identifica le responsabilità, e considerale una alla volta
- Chiediti a quale oggetto software assegnare questa responsabilità

- Chiediti come fa l'oggetto scelto a soddisfare questa responsabilità.

I pattern GRASP

Forniscono una guida per assegnare responsabilità durante la progettazione ad oggetti. L'acronimo significa **General Responsibility Assignment Software Patterns**. Tra i pattern GRASP di base troviamo:

- **Creator:** determina chi dovrebbe creare nuove istanze di una classe
 - **Problema:** Chi dovrebbe essere responsabile della creazione di una nuova istanza di una classe? È utile per avere un principio generale per assegnare questo tipo di responsabilità. Se vengono assegnate bene, il progetto può sostenere un accoppiamento basso, maggiore chiarezza, encapsulamento e riusabilità
 - **Soluzione:** Assegna alla classe B la responsabilità di creare un'istanza della classe A se una di queste condizioni è vera:
 - B contiene/aggrega con una composizione oggetti di tipo A
 - B registra A
 - B utilizza strettamente A
 - B possiede i dati per l'inizializzazione di A
 - **Discussione:** lo scopo è trovare un creatore che abbia bisogno di essere connesso all'oggetto creato (low coupling). Suggerisce che la classe per il "contenitore che racchiude" o "registratore" siano buoni candidati per la responsabilità di creare la cosa contenuta o registrata. Utilizzato il concetto di **composizione** tra le considerazioni per applicare il pattern creator
 - **Controindicazioni:** se la creazione dell'oggetto è complessa, meglio usare altre soluzioni come l'*Abstract Factory*
 - **Vantaggi:** favorisce un accoppiamento basso, poiché gli oggetti creati sono probabilmente già visibili all'oggetto creatore
 - **Pattern/principi correlati:** low coupling, design pattern creazionali, Whole-Part
- **Information Expert:** assegna responsabilità all'oggetto che ha le informazioni necessarie per compiere un'azione
 - **Problema:** qual è un principio di base con cui assegnare responsabilità agli oggetti?
 - **Soluzione:** assegna una responsabilità alla classe che ha le informazioni necessarie per soddisfarla
 - Guida verso una scelta che sostiene *Low Coupling*: Expert chiede di trovare l'oggetto che possiede la maggior parte delle informazioni richieste per la

responsabilità. Se ignoriamo i consigli, ci sono più oggetti o informazioni che devono essere condivisi

- **Discussione:** principio di base usato frequentemente nell'assegnazione di responsabilità, poiché per soddisfarla è spesso necessario far collaborare molti esperti parziali delle informazioni. Porta spesso a progetti in cui gli *oggetti software* fanno cose che normalmente *vengono fatte sugli oggetti inanimati del mondo reale* che essi rappresentano.
 - **Controindicazioni:** in alcuni casi, le soluzioni suggerite non sono opportune a causa di problemi di accoppiamento e di coesione.
 - **Vantaggi:** l'incapsulamento dell'informazione viene mantenuto, favorisce di solito un low coupling. Il comportamento è distribuito tra tutte le classi che possiedono le informazioni richieste, incoraggiando la definizione di classi più coese e leggere (*high cohesion*)
 - **Pattern/principi correlati:** low coupling, high cohesion
- **Low Coupling:** promuovere la minimizzazione delle dipendenze tra classi
- **Accoppiamento:** misura di quanto un elemento è **collegato**, conosce o si basa su altri elementi. Una classe con alto accoppiamento comportano che *i cambiamenti in classi correlate obbligano a cambiamenti locali, sono difficili da comprendere in isolamento, sono difficili da riusare perché il loro riuso richiede anche il riuso delle classi da cui dipendono*
 - **Problema:** come ridurre l'impatto dei cambiamenti?
 - **Soluzione:** assegna le responsabilità in modo tale che l'accoppiamento rimanga basso
 - **Discussione:** principio di valutazione. Incoraggia l'assegnazione di responsabilità in modo tale che il coupling raggiunto non sia troppo alto. Sostiene la **progettazione di classi** indipendenti, con un basso impatto dei cambiamenti; valuta meccanismi alternativi all'estensione di classi. Le forme di coupling da un tipo X a Y comprendono:
 - *La classe X ha attributo di tipo Y/referenzia una collezione di oggetti di tipo Y*
 - *Un oggetto di tipo X richiama servizi o operazioni di un oggetto di tipo Y*
 - *Un oggetto di tipo X crea un oggetto di tipo Y*
 - *Il tipo X ha un metodo che contiene un elemento di tipo Y*
 - *X è una sottoclasse di Y*
 - *Y è una interfaccia, e X implementa Y*
 - **Controindicazioni:** l'accoppiamento con elementi stabili non è un problema, il problema è l' high coupling con elementi instabili
 - **Vantaggi:** il cambiamento di altre classi ha un impatto basso, consente di definire classi semplici da comprendere in isolamento, e convenienti da riusare
 - **Pattern correlati:** protected variations
- **High Cohesion:** assicura che le operazioni all'interno di una classe siano strettamente correlate tra loro

- **Problema:** come mantenere gli oggetti focalizzati, comprensibili e gestibili e, come effetto collaterale, sostenere Low Coupling?
 - **Soluzione:** assegna le responsabilità in modo tale che la coesione rimanga alta. Usa principio per valutare le alternative.
 - **Coesione:** misura di quanto sono correlate le operazioni di un elemento software da un punto di vista funzionale.
 - **Discussione:** è un principio di valutazione, con vari livelli di coesione:
 - **Coesione molto bassa:** una classe è la sola responsabile di molte cose in aree funzionali molto diverse (*RDB-RPC-Interface*)
 - **Coesione bassa:** classe ha da sola responsabilità leggere in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe, ma non l'una con l'altra
 - **Coesione moderata:** una classe ha da sola responsabilità leggere in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe, ma non l'una con l'altra
 - **Coesione alta:** una classe ha responsabilità moderate in un'unica area funzionale e collabora con le altre per svolgere i suoi compiti. Pochi metodi, con funzionalità altamente correlate e focalizzate
 - **Controindicazioni:** i casi in cui la coesione bassa è giustificabile è quando c'è *raggruppamento di responsabilità/codice in una classe o componente gestito e mantenuto da una singola persona*. Inoltre, per motivi di efficienza, può essere necessario definire pochi *oggetti server* distribuiti che forniscono servizi ampi e poco coesi
 - **Vantaggi:** chiarezza e facilità di comprensione, manutenzione ed evoluzione semplificata, spesso favorisce un accoppiamento basso, funzionalità altamente correlate a grana fine favoriscono il riuso
 - **Pattern correlati:** Single-Responsibility Principle
- **Controller:** gestisce le richieste dell'interfaccia utente, e le delega al livello di dominio
- **Problema:** qual è il primo oggetto oltre lo strato UI a ricevere e coordinare un'operazione di sistema?
 - **Soluzione:** assegna la responsabilità a un oggetto che rappresenta:
 - Il **sistema complessivo**, un oggetto radice, un dispositivo all'interno del quale viene eseguito il software, un punto d'accesso al software, un sottosistema principale (*facade controller*)
 - Uno **scenario di un caso d'uso** all'interno del quale si verifica l'operazione di sistema (*controller di caso d'uso*)
 - **Discussione:** è semplicemente un pattern di delega, *da UI a dominio*. Un controller deve delegare ad altri oggetti il lavoro da fare durante l'operazione di sistema, limitandosi a controllare o coordinare le attività. Un **difetto** comune nella progettazione dei controlli consiste nel dare troppe responsabilità a un controller.

- Un **controller gonfio** è un controller con *coesione bassa* che gestisce *responsabilità in troppi contesti*: esiste una unica classe controller che riceve tutti i numerosi eventi di sistema, il controller svolge parte del lavoro prima di delegarlo, il controller ha numerosi attributi e conserva informazioni sul sistema e sul dominio.
 - **Vantaggi:** aumenta la possibilità di riuso dello strato di dominio, e consente di definire interfacce utente “inseribili”.
- **Pure fabrication**
 - **Problema:** quale oggetto deve avere la responsabilità quando non si vogliono violare High Cohesion e Low Coupling, o altri obiettivi, ma le soluzioni da Expert non sono appropriate?
 - **Soluzione:** assegnare un insieme di responsabilità altamente coeso a una classe artificiale o di convenienza, che non rappresenta un concetto del dominio del problema, ma piuttosto è una **classe inventata** per sostenere coesione alta, accoppiamento basso e riuso
 - **Svantaggi:** un utilizzo eccessivo può portare ad una decomposizione del codice: ovvero, si potrebbe avere un eccesso di oggetti comportamentali che non possiedono le informazioni richieste per soddisfarli
 - **Vantaggi:** sostiene high cohesion, con una possibilità di riuso aumentata
- **Polymorphism**
 - **Problema:** come gestire alternative basate sul tipo? Come creare componenti software inseribili?
 - **Soluzione:** quando le alternative e comportamenti correlati variano con il tipo, allora assegna la *responsabilità del comportamento* ai tipi per i quali il comportamento varia, utilizzando operazioni polimorfe
 - **Svantaggi:** progettare interfacce e polimorfismi per una adattibilità speculativa rispetto a delle possibili variazioni sconosciute può portare alla **progettazione di variazioni improbabili che in realtà non si verificheranno mai**
 - **Vantaggi:** le estensioni sono facili da aggiungere, si possono introdurre nuove implementazioni senza influire sui client
- **Indirection**
 - **Problema:** dove assegnare una responsabilità, per evitare l'accoppiamento diretto tra più elementi? Come disaccoppiare gli oggetti in modo da sostenere un alto riuso e un accoppiamento basso?
 - **Soluzione:** si assegna la responsabilità ad un oggetto intermediario che media tra altri o componenti o servizi, in modo che **non ci sia un accoppiamento diretto** tra di essi
- **Protected Variations**
 - **Problema:** come progettare oggetti, sottosistemi e sistemi in modo tale che *le variazioni o l'instabilità di questi elementi* non si riversino su altri elementi?

- **Soluzione:** si identificano i punti in cui sono previste delle variazioni, poi si assegnano le responsabilità per creare una “interfaccia” stabile attorno a questi punti
- **Svantaggi:** è inutile e costoso usarlo per delle variazioni che non si verificheranno mai. In alcuni momenti, conviene di più riprogettare che realizzare la protezione da tutte le possibili variazioni
- **Vantaggi:** le estensioni richieste per nuove variazioni sono facili da aggiungere, è possibile introdurre una nuova implementazione senza influire sui client, favorisce il low coupling, è possibile ridurre l'impatto o il costo dei cambiamenti

Nell'ambito di UML, il disegnare i diagrammi di interazione diventa *l'occasione per realizzare tali responsabilità*, realizzate come metodi; quindi, mentre si disegna un diagramma di interazione vanno prese delle decisioni riguardo all'assegnazione di responsabilità.

Cosa sono i pattern

Un buon pattern è una coppia problema/soluzione ben conosciuta, e con un nome. Può essere applicato in nuovi contesti, contiene consigli su come applicarla in situazioni nuove, e discute compromessi, implementazioni e variazioni.

I pattern **codificano soluzioni, idiomì e principi già applicati**, e che si sono dimostrati corrette in varie situazioni. Quindi, **non affermano nuove idee**. Hanno un nome, per facilitare la comprensione, memorizzazione e comunicazione.

Diagrammi di classi

Dipendenza

Una dipendenza è una **relazione di dipendenza** dove un elemento è **accoppiato** con un altro elemento, o dipende da un altro elemento. Alcune tipologie possono essere avere *un attributo del tipo del fornitore, inviare un messaggio a un fornitore, ricevere un parametro del tipo del fornitore*. Le dipendenze vanno usate per descrivere *le dipendenze tra oggetti per visibilità globale, per variabile parametro, per variabile locale, per metodo statico*.

Vincoli

Un vincolo indica una restrizione o una condizione su un elemento di un diagramma; il testo può essere in linguaggio naturale o altro, per esempio nel linguaggio di specifica formale per UML ovvero OCL.

Nel rettangolo di una classe, un attributo o un metodo sottolineato indicano un membro statico, anziché un membro d'istanza. Indicando un “1”, si può indicare che sia un **oggetto singleton**.

Interfaccia

Un'interfaccia è un **insieme di funzionalità pubbliche** identificate da un nome, e separa le specifiche di una funzionalità dall'implementazione. Un'interfaccia **definisce un contratto** e tutti i classificatori che la devono rispettare. Ecco cosa specifica l'interfaccia:

Interfaccia specifica	Classificatore di realizzazione
Operazione	Deve avere un'operazione con la stessa segnatura e semantica
Attributo	Deve avere operazioni pubbliche per leggere o cambiare il valore dell'attributo. Al classificatore non è richiesto di possedere l'attributo, ma deve comportarsi come se lo avesse
Associazione	Deve avere un'associazione al classificatore destinazione – se interfaccia specifica associazione a un'altra interfaccia, i classificatori di queste interfacce devono avere un'associazione tra di loro
Vincolo	Deve supportare il vincolo
Stereotipo	Deve avere lo stereotipo
Valore etichettato	Deve avere il valore etichettato
Protocollo	Deve realizzare il protocollo

Un'interfaccia fornita indica che un classificatore implementa il servizio definito in un'interfaccia; può essere fatta con una notazione a **pallina**. Un'interfaccia richiesta indica che un classificatore utilizza i servizi definiti dall'interfaccia.

Progettando con le interfacce si:

- Può stabilire **protocolli comuni** che potrebbero essere realizzati da molte classi o componenti
- Forniscono la capacità di inserire facilmente **nuove classi** nel sistema, poiché associazioni e invio di messaggi non sono più associati a specifici oggetti di classi particolari
- Può fare il **pattern di progetto facade**: nasconde i dettagli implementativi di sottoinsiemi complessi dietro un'interfaccia semplice e ben definita

Definendo un'interfaccia, definiamo un insieme di operazioni pubbliche, attributi e relazioni non implementati.



Connettore di assemblaggio

Si possono connettere interfacce richieste e fornite utilizzando il **connettore di assemblaggio**, il che è un cerchio con sopra un arco.

Architettura

L'insieme dei sottoinsiemi e delle interfacce di progetto costituisce l'**architettura di alto livello** di un sistema. L'insieme dovrebbe essere organizzato in modo coerente, applicando un'**architettura a strati**. Le dipendenze sono *unidirezionali* e tutte nello stesso senso, e tutte le dipendenze sono mediate da interfacce.

Vantaggi e svantaggi

Vantaggi:

- Quando progettiamo con le classi, stiamo progettando *specifiche implementazioni*
- Quando progettiamo con le interfacce, stiamo progettando dei **contratti** che possono essere realizzati da molte implementazioni diverse
- La progettazione per interfacce svincola il modello da dipendenze dell'implementazione e ne aumenta la *flessibilità* e la *estendibilità*

Svantaggi:

- Maggiore flessibilità significa anche **più complessità**
- Maggior costo a livello di prestazioni

Oggetti

Gli oggetti sono pacchetti come set di dati e funzioni **incapsulate in una unità riusabile**; quindi, incapsulano i dati.

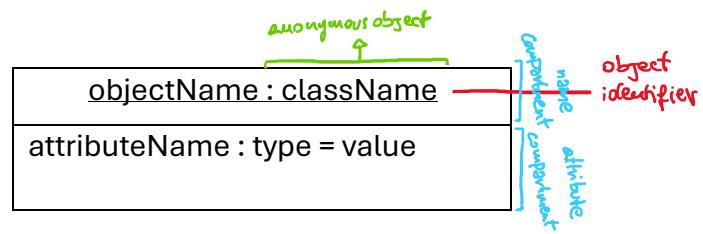
Ogni oggetto è un'**istanza di una classe** che definisce l'insieme comune di caratteristiche, attributi e operazioni, condivisi da tutte le sue istanze. Gli oggetti possono avere *i valori degli attributi*, e *le operazioni*; tutti gli oggetti hanno *l'identità* (identificativo univoco), lo *stato* (stabilito dai valori effettivi dei dati memorizzati), e il *comportamento* (l'insieme di operazioni che l'oggetto può eseguire).

Grazie all'**incapsulamento**, i dati sono nascosti all'interno dell'oggetto, e quindi l'unico modo per accedere ai dati è attraverso le operazioni.

Gli oggetti collaborano per eseguire le funzioni del sistema stabilendo dei **collegamenti** tra loro, e scambiandosi **messaggi**; quindi, questi causano l'invocazione delle operazioni da parte dell'oggetto.

Sintassi degli oggetti

Tutti gli oggetti di una particolare classe hanno lo **stesso insieme di proprietà**. I nomi degli oggetti sono scritti con le lettere maiuscole e minuscole, e iniziano con una lettera minuscola. Il nome della classe inizia invece con la *maiuscola*.



Classi

Una classe **modella** un insieme di oggetti omogenei (le istanze della classe) ai quali sono **associate** proprietà statiche (attributi) e dinamiche (operazioni); la classificazione è uno dei più importanti modi che noi abbiamo per organizzare la nostra vista del mondo.

ClassName
Attribute : type
Operazioni

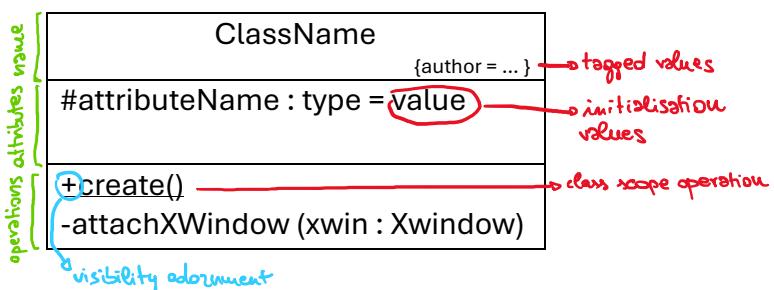
Tra un oggetto che è istanza di una classe C e la classe C si traccia un arco **Instantiate**; gli oggetti formano il *livello estensionale*, mentre le classi il *livello intensionale*.

"instantiate"
ObjectName

Attribute = value

Sintassi delle classi

È importante nella sintassi delle classi usare nomi descrittivi, ed evitare abbreviazioni.



Generalizzazione

Può accadere che **tra due classi sussista una relazione is-a**, tale che ogni istanza di una sia anche istanza dell'altra. La generalizzazione coinvolge una superclasse, ed una o più sottoclassi (classi derivate). Quindi, **ogni istanza di ciascuna sottoclasse è anche istanza della superclasse**. La si modella con una freccia vuota.

Vale ovviamente anche l'**ereditarietà**, dove ogni proprietà della superclasse è anche una proprietà della sottoclasse, e non si riporta esplicitamente nel diagramma. Ovviamente,

valgono sia per classi, oggetti, molteplicità e associazioni. Poiché un oggetto è istanza di una sola classe più specifica, due sottoclassi non disgiunte possono avere istanze comuni solo se hanno una sottoclasse comune. L'**ereditarietà multipla** in UML è supportata, e molte volte offre la soluzione più elegante; tuttavia, *non tutti i linguaggi la supportano*, e può essere sempre sostituita con ereditarietà singola e delegazione.

Con l'ereditarietà, quindi, definiamo l'**interfaccia** (le operazioni pubbliche della classe base) e l'**implementazione** (gli attributi, le relazioni e i metodi della classe base).

La superclasse, però, può anche generalizzare **diverse sottoclassi rispetto ad un unico criterio**, che si può indicare con un nome, ovvero il nome della generalizzazione.

Abbiamo, come tipi di generalizzazioni, **{overlapping}**, **{disjoint}** (i classificatori specifici non hanno istanze comuni), **{complete}**, **{incomplete}**.

Diagrammi delle classi

I diagrammi delle classi illustrano le classi, le interfacce e le relative associazioni; sono usati per la **modellazione statica degli oggetti**. Abbiamo un paio di note.

- Ufficialmente, in UML il formato in alto è utilizzato per **distinguere il nome del package dal nome della classe** (es. Java.awt::Font)
- La dipendenza è definita attraverso **una linea tratteggiata con curva**
- L'interfaccia è una **linea retta tratteggiata**, ed è mostrata con la parola chiave «interface» nel nome della classe
- La definizione di sottoclasse è una **linea retta con freccia vuota**
- I puntini di sospensione indicano che *ci possono essere degli elementi, ma non sono mostrati*
- Una sezione vuota significa ufficialmente “*sconosciuto*”, o anche per indicare che non c’è nessun membro.

I **classificatori** sono una generalizzazione di molti degli elementi di UML, come *classi*, *interfacce*, *casi d’uso*, e *attori*. Nei diagrammi delle classi, i due classificatori più comuni sono le **classi** e le **interface**.

Aggregazione e composizione

Una **aggregazione** è una associazione che rappresenta una relazione *intero-parte*, mentre una **composizione** è una forma forte di aggregazione in cui *una parte appartiene a un composto alla volta, ciascuna parte appartiene sempre a un composto*, e *il composto è responsabile della creazione e cancellazione delle sue parti*. L’ aggregazione si effettua con un **rombo vuoto** sulla freccia.

Template di classe

Fin ora, per definire una classe di progettazione si è detto che è **necessario specificare esplicitamente i tipi** dei suoi attributi, e i tipo di parametri ed il valore restituito dalle sue operazioni. I parametri del template sono sostituiti con valori effettivi per creare delle nuove classi: il valore predefinito verrà utilizzato se il tipo di parametro non è specificato quando il template è istanziato. Il binding esplicito è consigliato in quanto consente alle classi istanziate da template di avere un loro nome.

Proprietà strutturali

Le proprietà strutturali di un classificatore comprendono due tipi.

Inoltre, abbiamo le **parole chiavi**, un decoratore testuale per classificare un elemento di un modello. Alcune di queste sono:

Parola chiave	Significato	Esempio di uso
«actor»	Il classificatore è un attore	Nei diagrammi delle classi sopra un classificatore
«interface»	Il classificatore è un'interfaccia	Nei diagrammi delle classi, sopra al nome di un classificatore
{abstract}	L'elemento è astratto, e quindi non può essere istanziato	Nei diagrammi delle classi, dopo il nome di un classificatore o il nome di un'operazione
{ordered}	Un insieme di oggetti ha un ordine preferito	Nei diagrammi delle classi a un'estremità di associazione

Meccanismi di estendibilità

Abbiamo i seguenti meccanismi di estendibilità:

- **Vincoli**: estendono la semantica di un elemento consentendo di aggiungere nuove regole
- **Stereotipi**: definiscono un nuovo elemento basandosi su un esistente
- **Tag**: permettono di estendere la definizione di un elemento tramite l'aggiunta di nuove informazioni specifiche

Profili

I profili si usano per personalizzare UML per un uso specifico; è un **insieme di stereotipi, tag e vincoli**. Ogni stereotipo in un profilo estende uno degli elementi del meta-modello UML.

Proprietà

Una proprietà è un valore con un nome, che **denota una caratteristica di un elemento**. Possono essere, per esempio, la visibilità, e possono esservi anche associati dei valori.

Attributi

Un attributo **modella una proprietà locale** della classe, ed è caratterizzato da un nome e dal tipo dei valori associati. Ogni attributo di una classe stabilisce una proprietà locale **valida per tutte le istanze** della classe; il fatto che la proprietà sia locale significa che è una *proprietà indipendente da altri oggetti*.

Un attributo A della classe C si può considerare una **funzione** che *associa un valore di tipo T ad ogni oggetto* che è istanza di C. Però, un oggetto X non può avere un valore per un attributo che non è definito nella classe di cui X è istanza.

Due oggetti con identificatori distinti sono comunque distinti, *anche se hanno i valori di tutti gli attributi uguali*.

Sono scritti come:

Visibilità nome : tipo [molteplicità] = valoreIniziale

Solo il nome è **obbligatorio**; se non viene indicata alcuna visibilità, solitamente si ipotizza che gli attributi siano *privati*.

Visibilità

Solitamente, nella progettazione, gli attributi hanno di solito visibilità privata, grazie all'*incapsulamento*. Abbiamo il seguente schema:

Simbolo	Nome	Semantica
+	Pubblica	Ogni elemento che può accedere alla classe può anche accedere a ogni suo membro con visibilità pubblica
-	Privata	Solo le operazioni della classe possono accedere ai membri con visibilità privata
#	Protetta	Solo operazioni appartenenti alla classe o ai suoi discendenti possono accedere ai membri con visibilità protetta
~	Package	Ogni elemento nello stesso package della classe può accedere ai membri della classe con visibilità package.

Molteplicità

Se prendiamo un attributo B di tipo T in una classe C, e **non specifichiamo la molteplicità**, stiamo implicitamente affermando che ogni istanza di C avrà **esattamente un valore** di tipo T per l'attributo B; ha quindi molteplicità **1..1**.

Ogni attributo con molteplicità diversa è un **multivaleore**; diventa quindi una relazione che può associare più valori di tipo T ad ogni istanza di C. Nelle istanze, il valore di un attributo multivaleore si indica mediante un insieme, per esempio **[0..*]**.

UML mette a disposizione delle proprietà standard che possono essere applicate alle molteplicità per **indicare la semantica richiesta dalla connessione**.

Proprietà	Semantica
{ordered}	Gli elementi nella collezione vengono tenuti in ordine rigoroso
{unordered}	Non esiste alcun ordine degli elementi della collezione
{unique}	Gli elementi dell'insieme sono tutti univoci
{nonunique}	La collezione ammette duplicati

Si possono usare comunque **parole chiavi definite dall'utente**; basta che vengano definite all'interno di {}.

Attributi di associazioni

Anche le associazioni possono avere attributi. Formalmente, un attributo di una associazione è **una funzione che associa ad ogni link** (che è istanza di associazione) **un valore di un determinato tipo**. Quindi, una proprietà dell'associazione non è di nessuna delle classi, ma dell'associazione e basta.

Specializzazione di un attributo

In una generalizzazione, la sottoclasse non solo può avere proprietà aggiuntive rispetto alla superclasse, ma può specializzare le proprietà ereditate dalla superclasse.

Associazioni

Una associazione tra una classe C1 ed una classe C2 **modella una relazione matematica** tra l'insieme delle istanze di C1, e l'insieme delle istanze di C2. Modellano proprietà locali di una classe, e proprietà che coinvolgono altre classi. Le istanze di associazioni si chiamano **collegamenti**.

I collegamenti, però, *non hanno identificatori esplicativi*, che è quindi implicitamente identificato dalla tupla di oggetti che esso rappresenta. Tra le stesse due classi, inoltre, possono essere definite più associazioni.

Sintassi

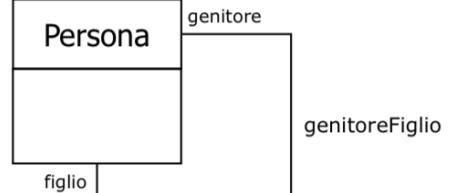
Le associazioni possono essere indicate o con il **nome**, oppure con il **nome dei ruoli**.

Potrebbe essere interessante un **verso per il nome** dell'associazione, anche se *non è una caratteristica del significato della associazione*, ma dice semplicemente che il nome scelto per la associazione evoca un verso.



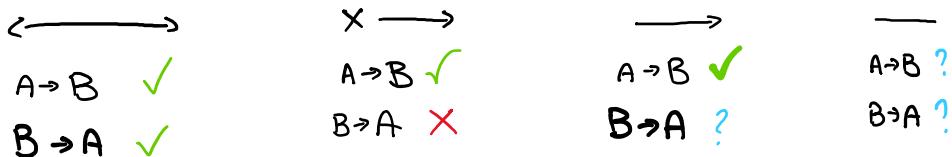
Ruoli

È possibile aggiungere una informazione che specifica il **ruolo** che una classe gioca nella associazione, il quale si indica con un nome posizionato lungo la linea che rappresenta l'associazione, vicino alla classe alla quale si riferisce. L'unico caso in cui il ruolo è obbligatorio è quello in cui **l'associazione insiste più volte sulla stessa classe**, e rappresenta una relazione non simmetrica, anche se non è necessario metterlo quando c'è una relazione simmetrica.



Navigabilità

Indica che è **possibile spostarsi da un qualsiasi oggetto della classe origine a uno o più di destinazione**; gli oggetti della classe di origine possono fare riferimento a oggetti della classe di destinazione usando il **nome del ruolo**.



Molteplicità

La molteplicità limita il numero di oggetti di una classe che possono partecipare in una relazione in un dato istante. Se la molteplicità non viene indicata esplicitamente, allora è **indefinita**. Abbiamo questa notazione:

- **x..y**: da x elementi a y elementi
- *****: più elementi (indefinito)
- **x..y, z..j**: da x elementi a y elementi, o da z elementi a j elementi

Associazioni n-arie

Una associazione può essere definita su tre o più classi; in questo caso, si dice **n-aria**, e modella una relazione matematica tra n insiemi; dunque, ogni istanza di una associazione n-aria è un **link n-ario**.

Specializzazione di un'associazione

Se una classe C1 partecipa ad un'associazione R con un'altra classe C3, e C2 è una sottoclasse di C1, specializzare R in C2 significa:

- **Definire** una nuova associazione, R2 tra la classe C2 e C4, che è sottoclasse di C3
- **Stabilire** una dipendenza di tipo {subsets} da R2 a R
- **Definire** eventualmente molteplicità più specifiche su R2 rispetto alle corrispondenti molteplicità definite su R.

Operazioni

Un'operazione è una **dichiarazione**, con un nome, parametri, tipo di ritorno, elenco di eccezioni, e magari un *insieme di vincoli di precondizioni e postcondizioni*. Una operazione indica che sugli oggetti della classe C si può eseguire una computazione *per calcolare le proprietà, o per effettuare cambiamenti di stato*.

Viene definito come:

```
visibility name(parameter-list) : return-type {property-string}  
visibility return-type name(parameter-list) property-string
```

Una operazione di una classe è pensata per essere invocata facendo riferimento ad un'istanza, chiamato **oggetto di invocazione**.

Le operazioni che non hanno un'implementazione sono **operazioni astratte**; quindi, una classe con una o più operazioni astratte non può essere istanziata (classe astratta). Abbiamo anche il **polimorfismo** nelle operazioni, dove un'operazione ha molte implementazioni.

Metodi

Un metodo è l'**implementazione di un'operazione**, e può essere illustrato in vari modi, tra cui:

- Nei **diagrammi di interazione**, dai dettagli che dalle sequenze dei messaggi
- Nei **diagrammi delle classi**, come una nota di UML stereotipata con «*method*»

Create

L'operazione **create** corrisponde a un *costruttore*, scritto con lo stereotipo «Create» o «Constructor». Operazioni di accesso come il **get** e **set** accedono e modificano gli attributi; di solito non vengono mostrate per l'elevato rapporto disturbo-valore che generano.

Specializzazione di operazioni

Una operazione si specializza **specializzando i parametri e/o tipo di ritorno**. Si noti che il metodo associato ad una operazione specializzata in una sottoclasse è in genere diverso dal metodo associato alla stessa operazione nella superclasse.

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 nel contesto di un particolare classificatore che *risponde a eventi esterni, ha un ciclo di vita definito che può essere modellato come una successione di stati/transizioni/eventi, e può avere un comportamento corrente che dipende dai comportamenti precedenti*. Le macchine a stati di solito vengono utilizzate per modellare il comportamento dinamico degli oggetti.

Abbiamo due tipi di oggetti all'interno:

- **Indipendente dallo stato:** risponde sempre nello stesso modo ad un determinato evento
- **Dipendente dallo stato:** risponde in modi diversi ad un determinato evento a seconda dello stato in cui si trova

Possiamo modellare il comportamento di un oggetto reattivo complesso in risposta agli eventi, oppure *modellare le sequenze valide delle operazioni, ovvero specifiche di protocollo o di linguaggio*. Può essere considerato una specializzazione del primo se l'oggetto è un protocollo o un processo.

Sintassi



Ogni macchina a stati dovrebbe avere uno **stato iniziale** che indica il primo stato della sequenza. A meno che esista un ciclo perpetuo di stati, dovrebbe avere uno stato finale che termina la sequenza di transizioni.

Stati

Uno stato è una condizione o una situazione della vita di un oggetto durante la quale tale oggetto soddisfa una condizione, esegue un'attività o aspetta un evento. Lo stato di un oggetto in qualsiasi momento è determinato da *i valori dei suoi attributi, le relazioni che ha con altri oggetti, e le attività che sta eseguendo*.

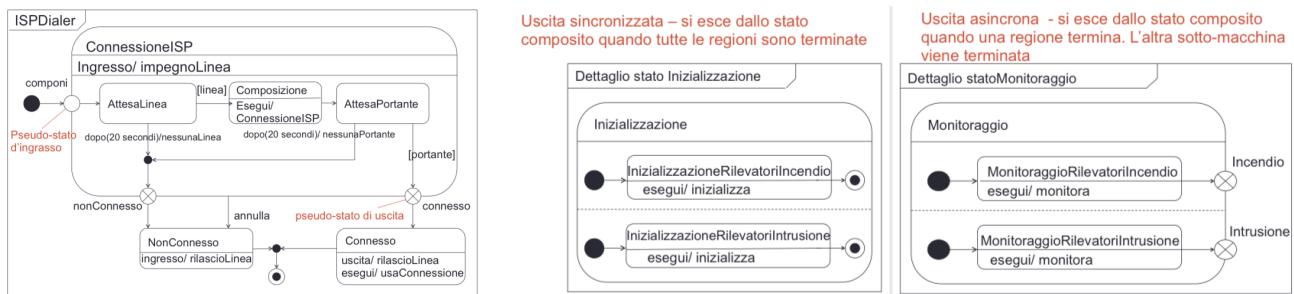
Le azioni sono istantanee e non interrompibili, le transizioni interne occorrono dentro lo stato. Non causano la transizione di un nuovo stato.

Le attività richiedono un intervallo di tempo finito, e sono **interrompibili**.

Nome stato
Azioni
ingresso/uscita
Transizioni interne

Stati compositi

Hanno una o più regioni ognuna delle quali contiene una **sotto-macchina annidata**. Possono essere **semplici**, con esattamente una regione, e **stati compositi ortogonali**, con due o più regioni. Lo stato finale viene applicato solo all'interno della regione in cui si trova, dove le altre regioni continuano la loro esecuzione. Lo pseudo-stato finale termina l'esecuzione dell'intero stato composto; si usa l'icona di composizione quando le sotto-macchine sono nascoste.



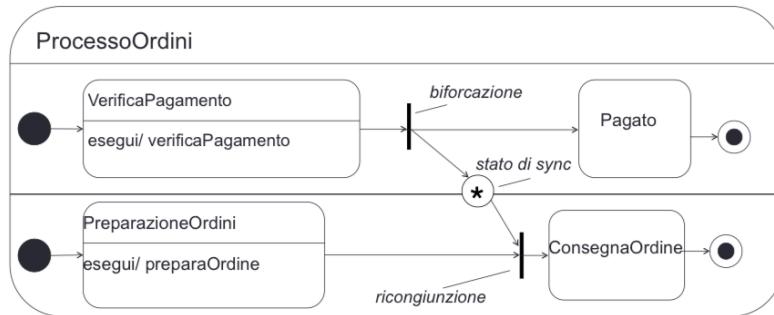
Stati della sotto-macchina

Se vogliamo fare riferimento a questa macchina a stati in altre macchine a stati, senza ingombrare i diagrammi, dobbiamo usare gli **stati della sotto-macchina**. Gli stati della sotto-macchina fanno riferimento a un'altra macchina a stati; gli stati della sotto-macchina sono semanticamente equivalenti agli stati compositi.

Uno stato della sotto-macchina è equivalente a **includere una copia della sotto-macchina** al posto dello stato della sotto-macchina.



Lo stato di **sync** è uno stato speciale il cui compito è quello di **tenere traccia di ogni singola attivazione** della sua unica transizione di input. Lo stato di sync è come una coda: *si aggiunge un elemento alla coda ogni volta che viene attivata la transizione di input*.



Stati con memoria semplice

Lo pseudo-stato con memoria semplice **ricorda in quale sotto strato si era quando si è lasciato il super-stato**. In seguito, quando si ritorna da uno stato esterno allo stato con memoria, l'indicatore ridireziona la transizione sull'ultimo sottostato memorizzato. Uno stato con memoria semplice non può ricordarsi dei sotto-stati di un suo sottostato.

Sottostato con memoria multilivello

Uno stato con memoria multilivello può non solo ricordarsi l'ultimo sottostato attivo allo stesso livello dello pseudo-stato, ma anche *l'eventuale ultimo sottostato attivo e così via...*

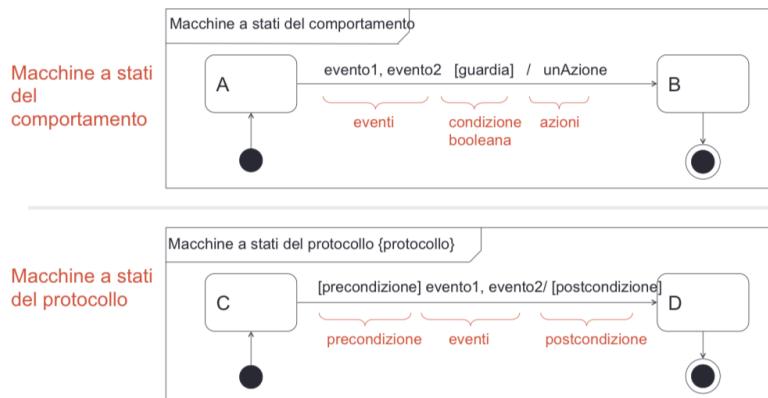
Comunicazione

Capita spesso di avere l'esigenza di far comunicare due sotto-macchine. Biforazioni e ricongiunzioni possono essere utilizzate per **creare sotto-macchine concorrenti** e per **risincronizzarle**.

La comunicazione **asincrona** è ottenuta da una sotto-macchina configurando un flag per un'altra sotto-macchina.

Transizioni

Sono rappresentate da frecce che collegano gli stati. Una transizione avviene quando un oggetto cambia stato in risposta a un evento, spesso sotto determinate condizioni.



Con una giunzione di ricongiungimento, si usa un **punto nero**.

Eventi

Gli eventi sono la **specificità di un'occorrenza di interesse** che ha una collocazione nel tempo e nello spazio. Gli eventi attivano le transizioni nelle macchine a stati.

Un evento di chiamata è una **chiamata per una specifica operazione**; l'evento dovrebbe avere la stessa segnatura di un'operazione della classe di contesto. Una sequenza di azioni può essere associata a un *evento di chiamata*.

Un **segnale** è un pacchetto di informazioni inviato in modo asincrono tra oggetti; gli attributi del segnale contengono le informazioni, e non ha operazioni perché il suo scopo è di trasportare informazioni. La ricezione di un segnale è indicata da un pentagono concavo.

Un **evento di variazione** è un'espressione booleana: l'azione viene eseguita quando il valore dell'espressione passa da falso a vero, e i valori dell'espressione booleana devono essere costanti, variabili globali, attributi o operazioni della classe di contesto. Da un punto di vista implementativo, un evento di variazione implica un *ciclo di test continuo* della condizione booleana per tutta la durata dello stato.

Gli **eventi temporali** occorrono quando un'espressione di tempo diventa vera. Le parole chiavi sono **dopo** e **quando**.

Diagramma dell'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, etc.

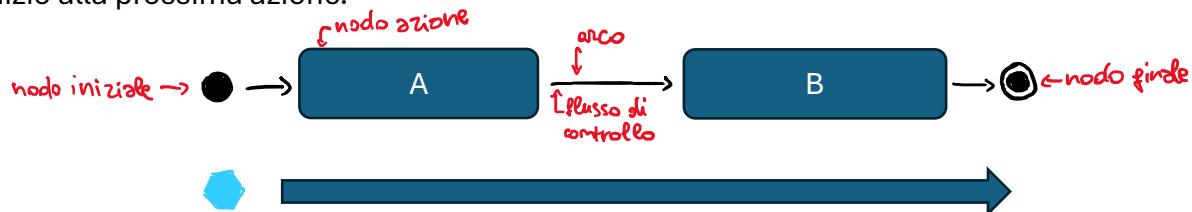
Le attività sono **reti di nodi connessi ad archi**, ed esistono tre categorie di nodi:

- **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.

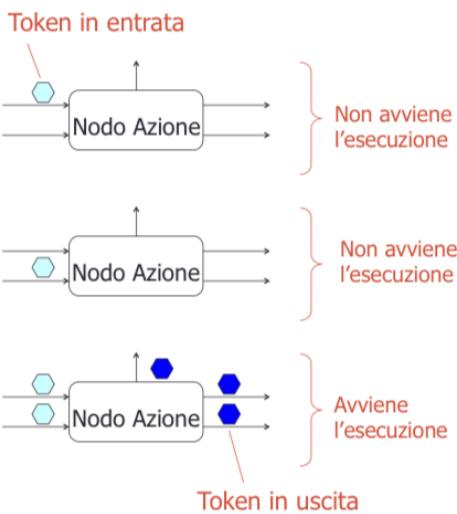
Gli archi rappresentano il flusso attraverso le attività, ed esistono due categorie: *flussi di controllo* (rappresentano il flusso di controllo attraverso le attività), e *flussi di oggetti* (rappresentano il flusso di oggetti attraverso l'attività). Questi diagrammi sono strettamente più legati alla fase di **progettazione**.

Sintassi e semantica

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



Il **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.



Nodi azione

L'esecuzione dei nodi azione avviene quando:

- Esiste un token simultaneamente su ciascun arco entrante
- I token in ingresso soddisfano tutte le precondizioni del nodo azione
- I nodi azione
 - o Eseguono un *AND logico* sui loro token di entrata
 - o Eseguono una *fork implicita* su tutti i suoi archi uscenti quando l'esecuzione è terminata

I tipi sono:

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

Nodo azione di chiamata

È il tipo più comune di nodo azione. Il nodo azione può invocare *un'attività*, *un comportamento* e *un'operazione*. Può contenere frammenti di codice in un specifico linguaggio di programmazione. La parola chiave **self** fa riferimento alle caratteristiche del contesto dell'attività.

Nodi controllo

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

Nodi decisione e fusione

Un **nodo decisione** è un nodo controllo che ha un arco entrante, e due o più archi alternativi uscenti. Ogni arco uscente è protetto da una condizione di guardia, e la condizione di guardia deve essere mutuamente esclusiva. L'arco uscente può essere preso se e solo se *la condizione di guardia è vera*. La parola chiave, altrimenti, specifica che il percorso viene preso se nessuna delle condizioni di guardia è vera.

Un **nodo fusione** accetta uno dei diversi flussi alternativi, dove ha due o più archi in ingresso, e esattamente un arco in uscita.

Nodi biforcazione e ricongiunzione

I **nodi biforcazione** modellano flussi concorrenti; i token che arrivano all'arco in ingresso vengono duplicati e offerti simultaneamente su tutti gli archi in uscita.

I **nodi ricongiunzione** sincronizzano due o più flussi concorrenti. Hanno più archi in ingresso e un solo arco in uscita, e offrono sul loro arco in uscita quando c'è un token su tutti i loro archi in ingresso.

Nodi oggetto

I nodi oggetto indicano che sono disponibili istanze di un particolare classificatore. Ogni nodo oggetto può tenere un numero infinito di token oggetto; il limite superiore definisce il numero massimo di token che è in grado di tenere.

I token sono offerti agli archi in uscita secondo un ordine **FIFO**, **LIFO**, o **comportamento di selezione**.

Inoltre, hanno una sintassi flessibile, dove è possibile indicare *limiti superiori*, *ordinamento*, *insiemi di oggetti*, *criteri di selezione*, e lo *stato dell'oggetto*.

Un **pin** è un nodo oggetto che rappresenta un input in un'azione o output da un'azione.

Design pattern

Un design pattern è un modo di **riutilizzare la conoscenza astratta** di un problema e la sua soluzione; è una descrizione del problema e l'essenza della sua soluzione. Dovrebbe essere sufficiente astratto per essere riutilizzato in diversi contesti, mentre le descrizioni dei pattern di solito fanno uso di caratteristiche orientate all'oggetto, come l'**ereditarietà** e il **polimorfismo**. Aiutano a:

- Costruire del software che sia riusabile
- Evitare scelte che compromettano il riutilizzo del software
- Migliorare la documentazione e la manutenzione di sistemi esistenti

Un design pattern *nomina*, *astrae* ed *identifica* gli aspetti chiave di una struttura di design comune, che la rendono utile per creare un design OO riusabile. È formato da quattro elementi essenziali:

- Il **nome** del pattern, è utile per descrivere la sua funzionalità in una o due parole
- Il **problema** nel quale il pattern è applicabile. Spiega il problema e il contesto, a volte descrive dei problemi specifici del design, mentre a volte può descrivere strutture di classi e oggetti. Può anche includere una lista di condizioni che devono essere soddisfatte perché il pattern possa essere applicato
- La **soluzione** descrive in modo astratto come il pattern risolve il problema; descrive gli elementi che compongono il design, le loro responsabilità e le collaborazioni
- Le **conseguenze** portate dall'applicazione del pattern spesso sono tralasciate, ma sono importanti per poter valutare i costi-benefici nell'utilizzo del pattern

Descrizione ed obiettivi

Il formato che si sceglie per la descrizione di un pattern non è rilevante, purchè la descrizione sia completa, coerente e accurata. La descrizione di un design pattern viene comunque data usando un formato preciso, articolato in diverse sezioni:

Nome	Descrizione
Nome	Classificazione del pattern
Sinonimi	Altri nomi del pattern
Scopo	Cosa fa il pattern? A cosa serve?
Motivazione	Scenario che illustra un design problem
Applicabilità	Situazioni in cui si applica il pattern
Struttura	Rappresentazione delle classi in stile Object Modeling Technique
Partecipanti	Classi e oggetti inclusi nel pattern
Collaborazioni	Come i partecipanti collaborano
Conseguenze	Come conseguono i suoi obiettivi il pattern?
Implementazione	Che tecniche di codifica sono necessarie?
Codice di esempio	Frammenti di codice
Usi noti	Esempi di applicazione del pattern in sistemi reali
Pattern correlati	Quali pattern sono correlati?

Questi aiutano gli sviluppatori a risolvere problemi già studiati, mentre mostra anche che la soluzione descritta è **utile** (ricorrenza), **utilizzabile** (contesto specifico) e **usata** (provata).

Catalogo e classificazione

Un primo criterio per la classificazione riguarda lo **scope**:

- **Classi**: pattern che definiscono le relazioni fra classi e sottoclassi. Le relazioni sono basate prevalentemente sul concetti di ereditarietà, e sono quindi **statiche**
- **Oggetti**: pattern che definiscono relazioni tra oggetti, che possono cambiare durante l'esecuzione e sono quindi più dinamiche.

Altro criterio invece riguarda il **purpose**:

- **Creazionali**: i pattern di questo tipo sono relativi alle operazioni di creazione di oggetti
- **Strutturali**: sono utilizzati per definire la struttura del sistema in termini della composizione di classi ed oggetti. Si basano su OO di ereditarietà e polimorfismo
- **Comportamentali**: permettono di modellare il comportamento del sistema definendo le responsabilità delle sue componenti, e definendo le modalità di interazione

Creational

Raccoglie i pattern che forniscono un'astrazione del processo di istanziazione degli oggetti; permettono di rendere un sistema **indipendente da come gli oggetti sono creati, rappresentati e composti al suo interno**.

Design patterns	Descrizione	Pattern collegati
Builder	Separa la costruzione di un oggetto complesso dalla sua rappresentazione in modo da poter usare lo stesso processo di costruzione per altre rappresentazioni	
Abstract Factory	Fornisce un'interfaccia per creare famiglie di oggetti in relazione, senza specificare le loro classi concrete	
Factory Method	<p>Definisce un'interfaccia per creare un oggetto, ma lascia decidere alle sottoclassi quale classe istanziare</p> <p>Problema: chi deve essere responsabile della creazione di oggetti quando ci sono delle considerazioni speciali, come una logica di creazione complessa, quando si desidera separare le responsabilità di creazione per una coesione migliore, e così via?</p> <p>Soluzione: crea un oggetto Pure Fabrication chiamato una Factory che gestisce la creazione</p>	<p>La logica della creazione è data-driven, sostiene protected variations rispetto a cambiamenti nella scelta dell'adattatore da utilizzare</p> <p>Spesso si accede agli oggetti factory mediante il pattern singleton</p>
Prototype	Specifica il tipo di oggetto da creare usando un'istanza prototipo, e crea nuovi oggetti copiando questo prototipo	
Singleton	<p>Assicura che la classe abbia una sola istanza, e fornisce un modo di accesso</p> <p>Problema: è consentita, o richiesta, esattamente una sola istanza di una classe; gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto</p> <p>Soluzione: definisci un metodo statico della classe che restituisce l'oggetto singleton</p> <p>Definisce un punto di accesso globale a un unico oggetto istanza di una classe —> inizializzazione pigra</p> <p>Definisce oggetti con visibilità globale —> seabusata ha aumento accoppiamento, può essere non efficace nella realizzazione di applicazioni distribuite</p> <p><i>Cosa fare:</i></p> <ul style="list-style-type: none"> - Attributo instance di tipo della classe, statico e singleton - Metodo getInstance di tipo della classe, statico e singleton 	Viene spesso usato per oggetti Factory e Facade

	<pre>public static synchronized Type getInstance() { if(instance == null) { instance = new Type(); } return instance; }</pre>	
--	---	--

Structural

I pattern di questa categoria sono dedicati alla **composizione di classi e oggetti** per formare strutture complesse. È possibile creare delle classi che ereditano da più classi, per consentire di utilizzare proprietà di più superclassi indipendenti. Particolarmenete utili per fare in modo che le **librerie di classi** sviluppate indipendentemente possano operare insieme.

Design Patterns	Descrizione	Pattern collegati
Adapter	<p>Converte l'interfaccia di una classe in un'altra, permettendo a due classi di lavorare assieme anche se hanno interfacce diverse</p> <p>Problema: come gestire interfacce incompatibili, o fornire un'interfaccia stabile a componenti simili ma con interfacce diverse?</p> <p>Soluzione: converti l'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio</p>	
Bridge	Disaccoppia un'astrazione dalla sua implementazione in modo che possano variare in modo indipendente	
Composite	<p>Compone oggetti in strutture ad albero per implementare delle composizioni ricorsive</p> <p>Problema: come trattare un gruppo o una struttura composta di oggetti dello stesso tipo nello stesso modo di un oggetto non composto?</p> <p>Soluzione: definisci le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia Interfaccia comune tra il padre e le foglie, ogni foglia deve avere metodi per aggiungere altri metodi composti</p>	
Decorator	Aggiunge nuove responsabilità ad un oggetto in modo dinamico, è un'alternativa alle sottoclassi come strumento per estendere le funzionalità	

Facade	<p>Fornisce un'interfaccia unificata per le interfacce di un sottosistema in modo da rendere più facile il loro utilizzo</p> <p>Problema: è richiesta un'interfaccia comune e unificata per un insieme disparato di implementazioni o interfacce, come per definire un sottosistema. Può verificarsi un accoppiamento indesiderato a molti oggetti nel sottosistema, oppure l'implementazione del sottosistema può cambiare; che cosa fare?</p> <p>Soluzione: definisci un punto di contatto singolo con il sottosistema, ovvero un oggetto facade che copre il sottosistema; presenta un'interfaccia singola e unificata, ed è responsabile della collaborazione con i componenti del sottosistema.</p>	<p>Si accede a Facade attraverso Singleton.</p> <p>Fornisce Protected Variations dall'implementazione di un sotto sistema, aggiungendo Indirection che aiuta a sostenere Low Coupling.</p> <p>Un oggetto Adapter può essere utilizzato per avvolgere l'accesso a sistemi esterni con interfacce variabili</p>
Proxy	Fornisce un surrogato di un oggetto per controllarne gli accessi	
Flyweight	Usa la condivisione per supportare in modo efficiente un elevato numero di oggetti con granularità fine	

Behavioural

Sono dedicati all'**assegnamento di responsabilità tra gli oggetti e alla creazione di algoritmi**; una caratteristica comune di questi pattern è data dal supporto fornito per seguire le comunicazioni che avvengono tra gli oggetti.

L'utilizzo di questi pattern permette di dedicarsi principalmente alle connessioni tra oggetti, tralasciando la gestione dei flussi di controllo.

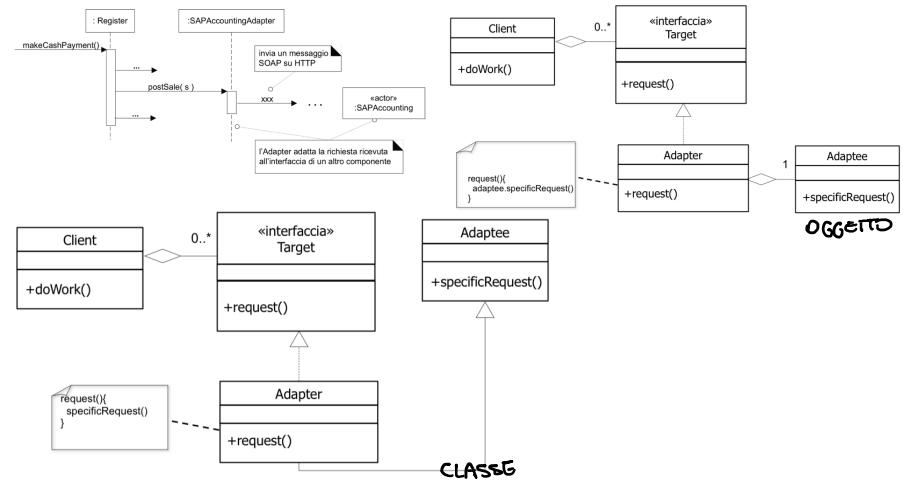
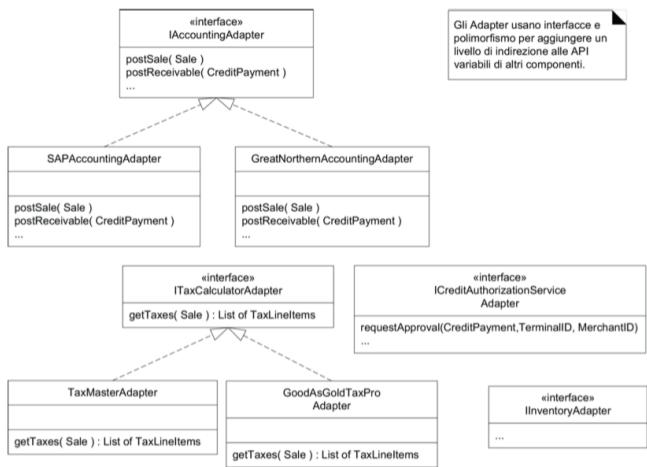
Design patterns	Descrizione	Pattern collegati
Chain of responsibility	Evita l'accoppiamento di chi manda una richiesta con chi la riceve, dando a più oggetti la possibilità di maneggiare la richiesta	
Command	Incapsula una richiesta in un oggetto in modo da poter eseguire operazioni che non si potrebbero eseguire	
Interpreter	Dato un linguaggio, definisce una rappresentazione per la sua grammatica, ed un interprete per il linguaggio	

Iterator	Fornisce un modo per accedere agli elementi di un oggetto aggregato in modo sequenziale	
Mediator	Definisce un oggetto che incapsula il modo in cui un insieme di oggetti interagiscono mantenendo la loro indipendenza	
Memento	Cattura e porta all'esterno lo stato interno di un oggetto senza violare l'incapsulamento in modo da poter ripristinare il suo stato	
Observer	<p>Fornisce un modo per accoppiare in modo debole gli oggetti che devono comunicare. I publisher conoscono i subscriber con un'<i>interfaccia</i>, i subscriber si registrano dinamicamente con il publisher</p> <p>Definisce una dipendenza 1:N tra oggetti in modo che se uno cambia stato, gli altri siano aggiornati automaticamente</p> <p>Problema: diversi tipi di oggetti <i>subscriber</i> sono interessati ai cambiamenti di stato o agli eventi di un oggetto <i>publisher</i>. Ciascun subscriber vuole reagire in un suo modo proprio quando il publisher genera un evento. Il publisher vuole mantenere un accoppiamento basso verso i subscriber, che cosa fare?</p> <p>Soluzione: definisci un'interfaccia <i>subscriber</i> o <i>listener</i>. Gli oggetti <i>subscriber</i> implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.</p>	<p>Observer è basato su Polymorphism. Fornisce Protected Variations in termini di protezione del publisher dalla conoscenza della classe specifica degli oggetti con cui comunica quando genera un evento.</p>
State	Permette ad un oggetto di cambiare il proprio comportamento a seconda del suo stato interno, come se cambiasse classe di appartenenza	
Strategy	<p>Definisce una famiglia di algoritmi, li incapsula e li rende intercambiabili indipendentemente dagli utilizzatori</p> <p>Problema: come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati? Come progettare per consentire di modificare questi algoritmi o politiche?</p> <p>Soluzione: definisci ciascun algoritmo/politica/strategia in una classe separata con un'interfaccia comune</p>	<p>L'applicazione di Strategy e Factory consente di ottenere Protected Variations rispetto alle politiche di determinazione del totale. Si basa su Polymorphism e sulle interfacce per consentire degli algoritmi inseribili in un progetto.</p>

		Strategie spesso create da una Factory
Template Methods	Definisce lo scheletro di un algoritmo in un'operazione dove alcuni passi sono definiti nelle sottoclassi	
Visitor	Rappresenta un'operazione sugli oggetti, permette di definire nuove operazioni senza cambiare classi	

Esempi

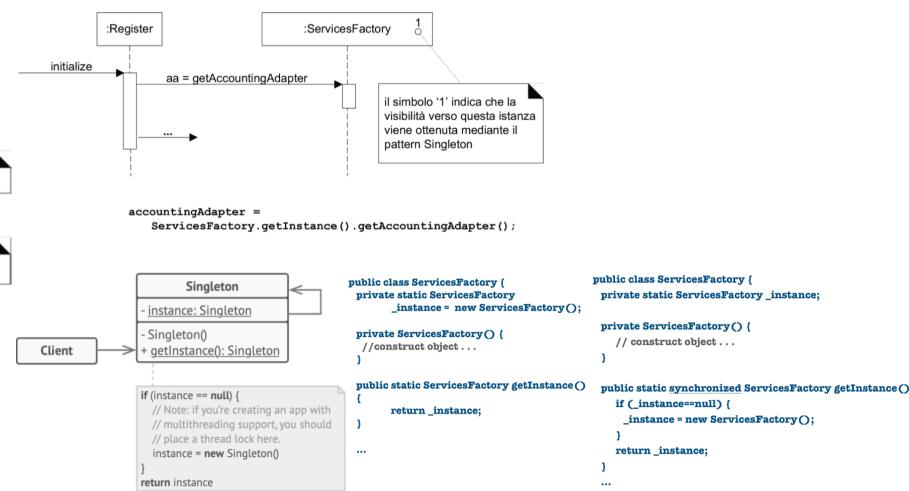
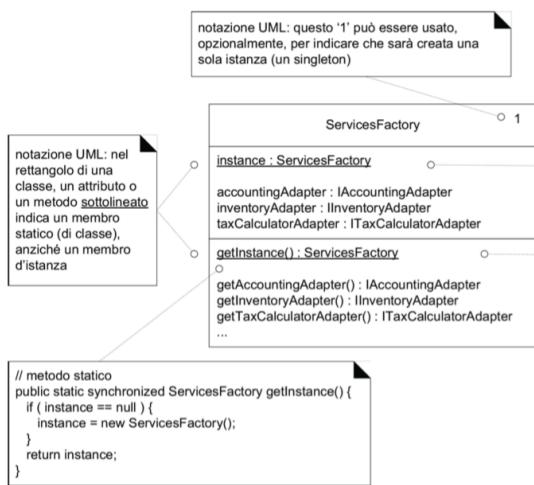
Adapter



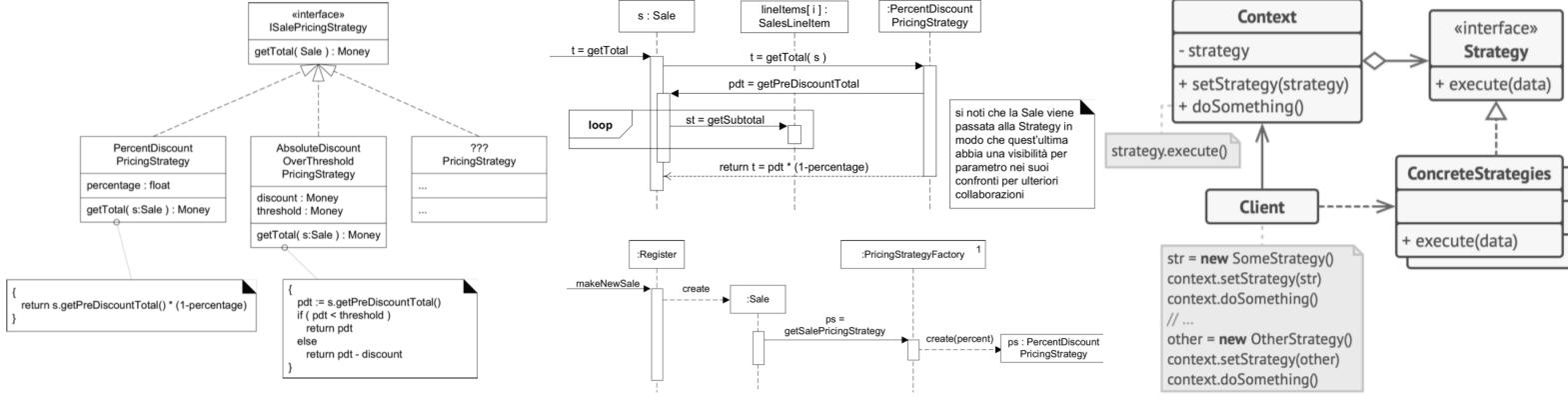
Factory

I metodi factory restituiscono degli oggetti il cui tipo è un'interfaccia, anziché una classe, in modo tale che la factory possa restituire una qualsiasi implementazione dell'interfaccia.

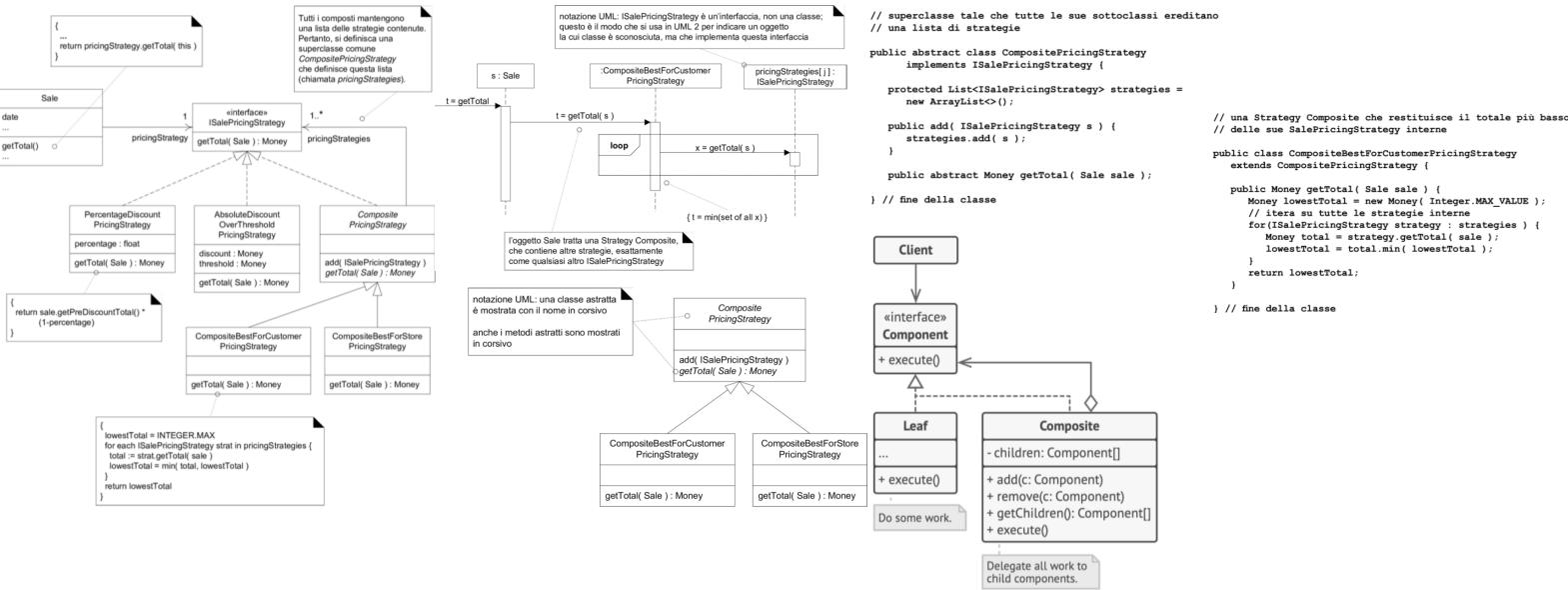
Singleton



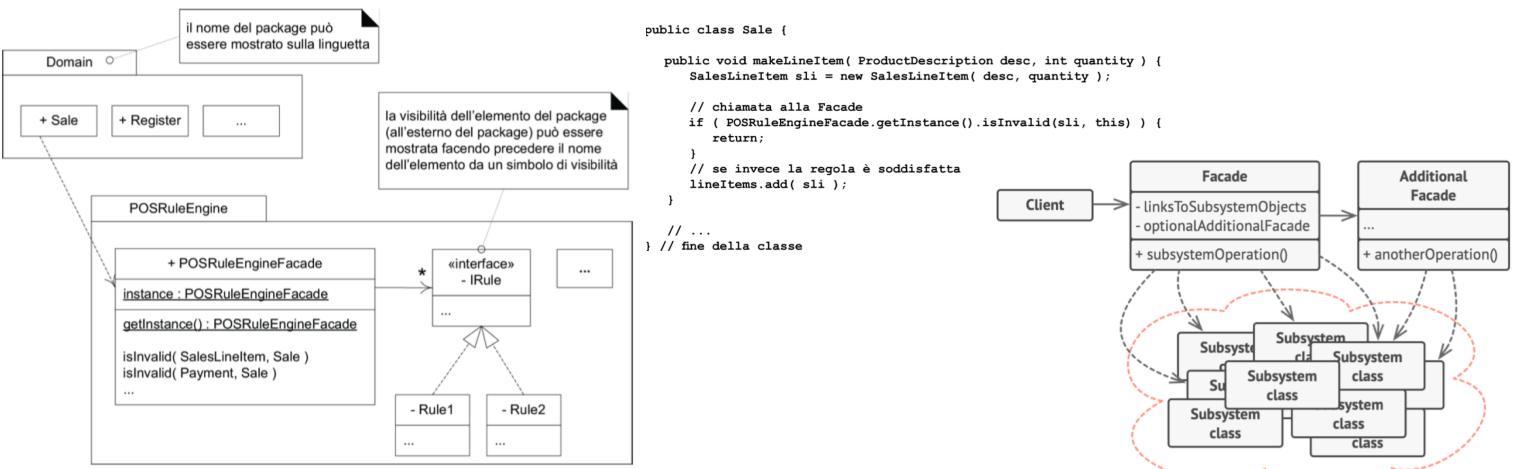
Strategy



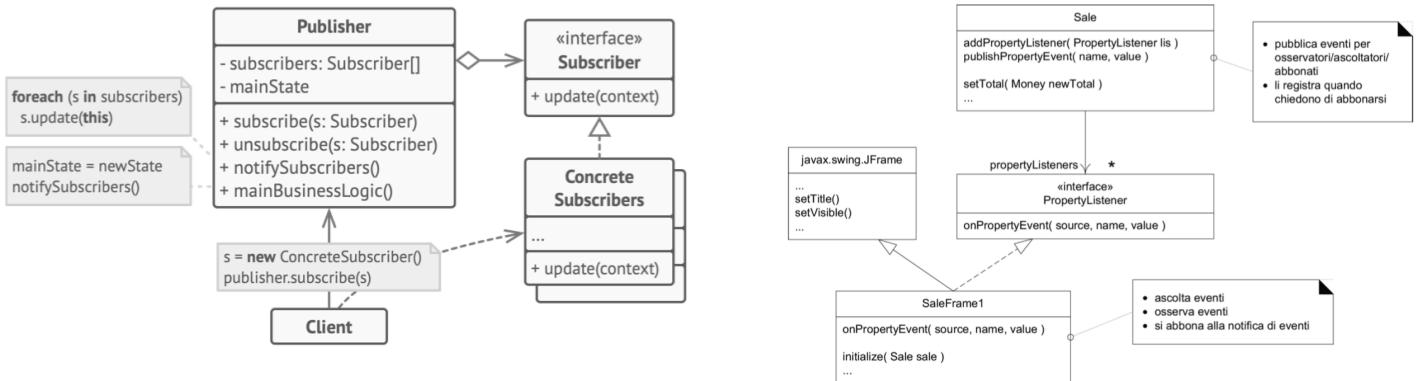
Composite



Facade



Observer



Framework

Un framework non è una semplice libreria, e rappresenta il **design riusabile di un sistema**, definito da un insieme di classi astratte. È lo **scheletro di un'applicazione che viene personalizzato** da uno sviluppatore; il programmatore di applicazioni implementa le interfacce e classi astratte, ed ottiene automaticamente la gestione delle funzionalità richieste.

I framework permettono quindi di definire lo scopo e la *struttura statica* di un sistema, e sono un buon esempio di progettazione orientata agli oggetti: raggiungono il riuso del design e del codice.

Una **classe astratta** è una classe che possiede almeno un metodo non implementato, definito astratto; è quindi un template per le sottoclassi da cui si derivano le specifiche.

applicazioni. È rappresentato da un'insieme di classi astratte e dalle loro interrelazioni, e sono spesso i mattoni per la costruzione di framework.

Pattern GRASP e altri pattern

La caratterizzazione, per esempio, del design pattern Adapter in termini di pattern GRASP:

- Sostiene **Protected Variations** rispetto al cambiamento di interfacce esterne o di package di terze parti
- Attraverso l'uso di un oggetto **Indirection**
- Applica interfacce e **polymorphism**

Inoltre, molti design pattern possono essere analizzati in termini dei pattern GRASP: i pattern GRASP possono semplificare la comprensione dei molti design pattern esistenti. Per un progettista, è utile saper vedere i principi di base dei numerosi design pattern esistenti.

Software Configuration Management

La SCM è una disciplina che si occupa di controllare l'evoluzione dei sistemi software, specialmente quelli sviluppati da team che lavorano in modo **parallelo** e **asincrono** su progetti complessi. L'obiettivo principale è quello di mantenere lo *stato del progetto e dei suoi artefatti* sempre ben definiti, e di garantire la possibilità di ripristinare uno stato precedente in un qualsiasi momento.

Repository

Un repository agisce come un file system virtuale gestito da un server. Ogni elemento all'interno del repository è soggetto a versionamento, con ogni versione identificata da un **numero di revisione progressivo**. Quando un elemento viene modificato, il numero di revisione corrispondente aumenta. È comune che un singolo server ospiti molti repository, generalmente uno per ogni nuovo progetto software.

Le operazioni principali all'interno di una repository sono:

- **Checkout:** creazione di una copia locale del repository
- **Commit:** invio al server delle modifiche. Il server verifica e applica cambiamenti al repository
- **Conflitti:** la gestione dei conflitti è cruciale quando più persone lavorano simultaneamente sullo stesso progetto. Il software di gestione facilita questo processo, identificando i cambiamenti fatti sulla stessa base di codice, segnalando eventuali conflitti e tentando di risolverli

- **Pull:** sincronizza la repository locale con la repository online, scaricando i cambiamenti localmente (online → locale)
- **Push:** carica la repository online alla repository locale (locale → online)

Possiamo categorizzare le directory in questo modo:

- **Trunk:** contiene la versione più recente del progetto
- **Branches:** rami per le diverse release, usati per lavorare su versioni diverse del progetto
- **Tags:** simili ai branches, ma usati per marcare specifiche release senza apportare modifiche.

Versioni e release

Il numero di revisione di un repository aumenta con ogni modifica apportata. Ogni repository ha numeri di versione indipendenti. Questo numero di revisione è **fondamentale** perché permette di recuperare lo stato del repository in corrispondenza di una determinata release.

I dati

Git considera i dati come una **serie di fotografie di un filesystem in miniatura**. Ogni volta che si fa una commit, ovvero salva lo stato del progetto, viene scattata una foto dei file e viene memorizzata. Se non c'è stata modifica di un file, viene creato un riferimento alla foto precedente.

I file possono avere tre tipi di stati:

- **Modified:** il file è stato cambiato ma non è stata effettuata commit
- **Staged:** file marcato come modificato, pronto per essere caricato nella prossima commit
- **Committed:** i dati sono stati caricati nel database

Inoltre, possono essere **tracked** se erano *file che erano già nella foto precedente, o che sono stati aggiunti*. Sono invece **untracked** se sono *file di cui git non è a conoscenza, e che quindi possono essere aggiunti*.

Git

Git è uno strumento essenziale per la gestione delle versioni, locale e online. Ecco le operazioni principali:

- **Inizializzazione:** `git init "nome_repo"` per creare una nuova repository locale
- **Aggiunta di file:** `git add "nome_file"` per aggiungere file alla repository; è fondamentale farlo ogni volta prima di fare un commit o qualsiasi altra operazione
- **Pull e push:** `git pull` scarica i file dal repository remoto, mentre `git push` invia i tuoi cambiamenti a GitHub dopo un commit

- `git pull "repoURL" "branch" -allow-unrelated-histories`
forza il pull da un certo repository e branch
- `git push -set-upstream "repoName" "branchName"` forza il push a un certo repository e branch
- **Commit:** `git commit -m "messaggio"` registra i tuoi cambiamenti nel repository
 - **Storia delle commit:** `git log`
- **Repository:** `git remote add "repoName" "repoURL"` – è necessario copiare il codice della repository da “<> Code”, e aggiungerlo per ogni sessione. Per richiamare la repo, sarà necessario *repoName*.
- **Differences:** `git diff` visualizza i file nuovi e modificati
- **Stato dei file:** `git status`
- **Branches**
 - `git branch "nomeBranch"` crea un nuovo branch
 - `git branch -list` visualizza la lista di tutti i branch
 - `git branch "nomeBranch" -d` elimina un branch
 - `git push "branch" -delete` cancella un branch remoto
 - `git branch "nomeBranch" -m` rinomina un branch
 - `git checkout "nomeBranch"` passa da un branch all'altro
 - `git merge "nomeBranch"` unisce un branch con quello attuale
 - `git fetch "url"` recupera un repository esterno e cambia la cronologia
 - `git fetch -all` scarica tutti i branch
 - `git fetch origin` sincronizza un repository originale
- **Stash**
 - `git stash` salva senza commit
 - `git stash save "comment"` salva le modifiche ma con un commento