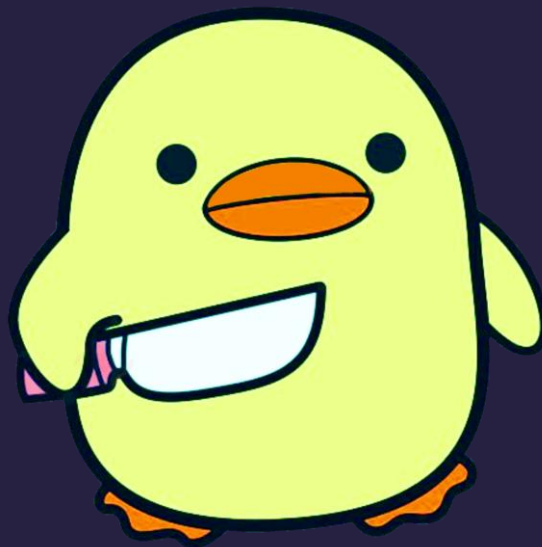


# Analisi e Progettazione del Software

Appunti per la preparazione all'orale



Blue3141  
Gatto

# Sommario

SOMMARIO.....	2
<b>INTRODUZIONE.....</b>	<b>4</b>
IL SOFTWARE .....	4
<i>Caratteristiche</i> .....	4
INGEGNERIA DEL SOFTWARE.....	4
<i>Processo Software</i> .....	4
<i>Attività fondamentali di processo</i> .....	4
<i>Nello specifico: Analisi e Progettazione</i> .....	5
STRUMENTI USATI .....	5
UML .....	5
<b>PROCESSI SOFTWARE .....</b>	<b>7</b>
PROCESSO SOFTWARE A CASCATA.....	7
SVILUPPO ITERATIVO, INCREMENTALE, EVOLUTIVO .....	7
<i>Iterazione</i> .....	8
UNIFIED PROCESS (UP) .....	8
<i>Fasi</i> .....	8
<i>Iterazioni</i> .....	9
<i>Cattive pratiche di UP</i> .....	9
METODO AGILE .....	9
<i>UP agile</i> .....	10
SCRUM.....	10
<b>UNIFIED PROCESS .....</b>	<b>11</b>
OVERVIEW .....	11
DIAGRAMMA DEGLI ELABORATI .....	12
<b>PRIMA FASE: L'IDEAZIONE.....</b>	<b>13</b>
REQUISITI .....	14
<i>I tipi di requisiti</i> .....	14
Gli obiettivi .....	15
<i>Acquisizione e redazione dei requisiti</i> .....	15
<i>[D] I Casi d'Uso</i> .....	15
Perché li usiamo?.....	16
Definizioni utili .....	16
Tipologie di Casi d'Uso .....	16
Template per un Caso d'Uso dettagliato .....	17
Stesura di un Caso d'Uso.....	17
Verificare l'utilità di un Caso d'Uso .....	18
<i>[D] Diagramma dei Casi d'Uso</i> .....	18
<b>FASE 2.1: ELABORAZIONE - 1° ITERAZIONE .....</b>	<b>19</b>
MODELLAZIONE DEL BUSINESS.....	20
<i>Analisi a oggetti</i> .....	20
<i>[D] Modello di Dominio</i> .....	20
Perché lo usiamo? .....	20
Le classi concettuali .....	21
Gli attributi.....	21
Le classi descrizione .....	21
Le associazioni tra classi concettuali .....	21
Stesura del Modello di Dominio.....	22
(Extra) <i>[D] Diagramma degli Oggetti</i> .....	23
REQUISITI .....	23
<i>[D] Diagrammi di Sequenza di Sistema (SSD)</i> .....	23
Linee guida .....	24
<i>[D] Contratti</i> .....	24
Struttura di un Contratto .....	24
Post-condizioni.....	24
DAI REQUISITI ALLA PROGETTAZIONE .....	25
<i>Architettura Software</i> .....	25

(Extra) Architetture a N+1 viste .....	25
<i>[D] Architettura logica</i> .....	25
Architettura logica a strati .....	26
Perché gli strati? .....	26
Strato: la logica applicativa .....	27
Principio di separazione Modello-Vista .....	27
PROGETTAZIONE .....	27
<i>Progettare a oggetti</i> .....	27
Modelli statici e dinamici .....	27
(Extra) Schede CRC .....	28
<i>[D] Diagrammi di Interazione</i> .....	28
[D] Diagrammi di Sequenza (SD) .....	28
[D] Diagrammi di Comunicazione (CD) .....	29
(Extra) [D] Diagrammi di Interazione Generale .....	29
<i>[D] Diagramma delle Classi Software</i> .....	30
Definizioni varie (nuove o rilevanti) .....	30
Ripasso generale UML .....	31
Interfacce .....	32
<i>[D] Macchine a Stati</i> .....	33
Rappresentazione di una Macchina a Stati .....	33
Gli stati .....	33
Stati composti .....	33
Pseudostati .....	34
Eventi .....	34
<i>[D] Diagramma delle Attività</i> .....	35
Token game .....	36
Nodo azione .....	36
Nodo controllo .....	36
Nodo Oggetto .....	36
Pin .....	36
Partizioni .....	36
<i>Progettazione guidata dalla responsabilità (RDD)</i> .....	37
<i>Pattern GRASP</i> .....	37
Tabella dei Pattern GRASP .....	38
<i>Design Pattern (GoF)</i> .....	41
Schemi dei Pattern GoF .....	42
<i>Progettare la visibilità</i> .....	46
DALLA PROGETTAZIONE ALL'IMPLEMENTAZIONE .....	46
TESTING .....	47
<i>Sviluppo guidato dai test (TDD)</i> .....	47
Tipi di test .....	47
Test di unità .....	47
Tipi di cicli per i test unitari .....	47
<i>Refactoring</i> .....	48
Quando fare refactoring .....	48
Processo di trasformazione .....	49
Code Smell .....	49
Tabella dei Refactoring .....	52
<b>DOMANDE DI TEORIA</b> .....	<b>58</b>

Gli autori si augurano che questo documento possa essere utile a chi studia per l'orale :3

Si tratta di una rielaborazione delle slide, integrata con il libro e con siti esterni (se c'è un approfondimento è chiaramente indicato con "Extra"), che dovrebbe contenere tutto il necessario per superare brillantemente il colloquio.

Osservazioni e correzioni sono ben accette!

# INTRODUZIONE

## Il software

Il software è un programma per computer con la sua documentazione (modelli di progetto, manuali...)

Può essere:

- **Generico:** sviluppato per un ampio insieme di clienti;
- **Personalizzato:** sviluppato per uno specifico cliente.

Un software può essere prodotto:

- **Partendo da zero**, sviluppando un nuovo programma;
- Sfruttando software integrandolo in un nuovo programma o utilizzandolo come base;
- Personalizzando programmi pre-esistenti.

## Caratteristiche

Un software deve avere le seguenti caratteristiche:

- Mantenibilità;
- Fidatezza;
- Efficienza;
- Accettabilità.

## Ingegneria del Software

Disciplina ingegneristica che si occupa di produrre software di buona qualità, dalle prime fasi fino alla messa in uso e oltre (manutenzione). È un insieme di teorie e metodi per sviluppare al meglio il progetto, ma è anche la gestione dello stesso e lo sviluppo di ciò che può supportarlo (nuovi strumenti/metodi...).

## Processo Software

Approccio disciplinato per la costruzione, il rilascio e la manutenzione del codice. Definisce chi (ruoli) fa cosa (attività), quando (organizzazione temporale) e come (metodologie) per raggiungere un certo obiettivo. Vi sono vari tipi di processi per lo sviluppo, ma la maggior parte hanno (o rielaborano) le attività fondamentali.

## Attività fondamentali di processo

Le seguenti attività sono solitamente incluse in ogni processo software (che tuttavia può cambiarle e gestirle come vuole).

- Requisiti;
- Analisi;
- Progettazione;
- Implementazione;
- Validazione;
- Rilascio e installazione;
- Manutenzione ed evoluzione;
- Gestione del progetto.

## Nello specifico: Analisi e Progettazione

Sono due attività di processo fondamentali; nel corso ci occuperemo principalmente di queste due attività (oggetto anche dei laboratori) per poi vedere più superficialmente le altre.

- **Analisi:** capire qual è il problema e i suoi requisiti;
- **Progettazione:** trovare una soluzione che soddisfa i requisiti.

Noi lavoriamo nell'ottica Object Oriented, dunque per noi l'analisi e la progettazione assumono significati più specifici:

- **Analisi Object Oriented:** definisce le classi concettuali (di dominio); servono a descrivere i concetti o gli oggetti (del mondo reale) relativi al problema;
- **Progettazione Object Oriented:** definisce le classi software, che servono a modellare i precedenti concetti del mondo reale in classi e oggetti software utilizzabili all'interno di codice scritto in un linguaggio Object Oriented.

Sono attività fortemente collegate: sebbene la prima operi a livello "astratto", a livello di dominio per formalizzare il problema, essa è la base per la seconda – a livello software – che andrà a definire classi e oggetti a partire da quelli di dominio per poter sviluppare il programma. Non sono da considerarsi tuttavia sequenziali: non è necessario completare l'analisi per procedere ad una prima progettazione.

Fare la cosa giusta (= analisi) e fare la cosa bene (= progettazione).

## Strumenti usati

Si possono usare vari programmi specifici per la modellazione e la gestione del software; un esempio è Rational Software Architect Designer, da noi usato nel laboratorio. Oltre ai programmi dedicati, si usano linguaggi visuali (UML) ed eventualmente strumenti creati appositamente per il progetto.

Un altro discorso è l'organizzazione del team di lavoro, sia a livello di tempo, che di componenti, che di materiali e ambienti di lavoro, tutti fattori subordinati al tipo di processo scelto; verranno discussi più avanti.

## UML

L'Unified modelling language è un (IL) linguaggio visuale di modellazione (per software e molto altro). Non ha uno standard ma è lo standard. **"Unified"** perché va bene con tutto: si adatta a qualsiasi piattaforma, processo di sviluppo, dominio applicativo (tipi di sistemi: embedded, app, server...) e a qualsiasi linguaggio. Non è una metodologia.

UML può modellare, all'interno del progetto, sia la struttura statica che quella dinamica. In particolare, nel paradigma a oggetti, si definiscono nel seguente modo:

- **Struttura statica:** classi e oggetti (sia di dominio che software) e le loro relazioni;
- **Struttura dinamica:** ciclo di vita degli oggetti e le loro interazioni che realizzano le funzionalità del programma.

Si può applicare UML a più livelli di dettaglio:

- **Abbozzo:** diagrammi informali, non completi, che descrivono i punti salienti, di interesse e/o complicati del progetto (*noi applichiamo questo*);
- **Progetto:** diagrammi relativamente dettagliati;

- **Linguaggio di programmazione:** diagrammi completi e completamente coerenti fra loro, che permettono di generare il codice del software (anche in maniera automatizzata se è stato usato un programma dedicato).

UML può essere usato sia durante l'analisi che durante la progettazione; più ampiamente, può descrivere e modellare classi e oggetti sia dal punto di vista concettuale che dal punto di vista software. Si possono quindi distinguere due punti di vista (applicabili a qualsiasi livello di dettaglio) che utilizzano la stessa notazione UML:

- **POV concettuale:** si modellano concetti o oggetti del mondo reale, scissi dalla loro trasposizione in codice. Le classi concettuali formano il **Modello di Dominio**;
- **POV software:** si modellano classi che rappresentano componenti software. Le classi software sono contenute nel **Modello di Progetto**.

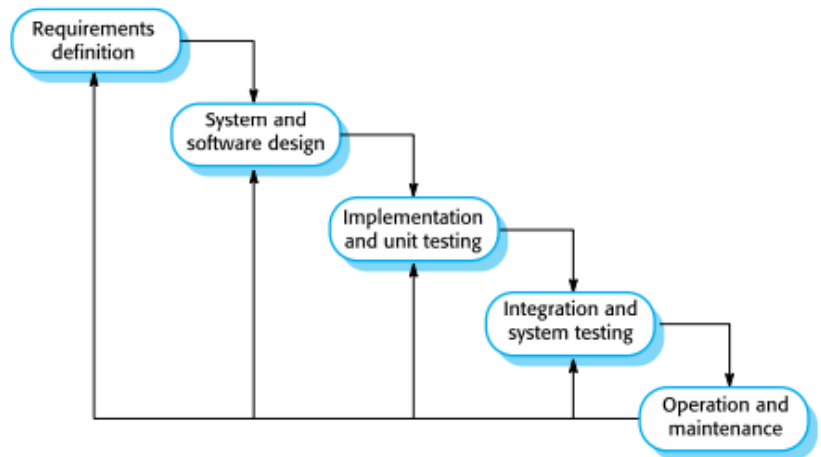
# PROCESSI SOFTWARE

Vediamo qui diverse tipologie di **processi software**, premettendo che nel resto del corso ci occuperemo dell'Unified Process.

## Processo Software a Cascata

Il **processo software a cascata** prevede l'**esecuzione sequenziale** (a cascata) delle seguenti attività:

- Definizione dei requisiti;
- Design del sistema e del software;
- Implementazione e testing unitario;
- Integrazione e testing di sistema;
- Messa in uso e manutenzione (che fa ripartire la cascata da uno degli step precedenti).



**Fallisce spesso** perché non è flessibile rispetto alle mutevoli esigenze dei clienti; richiede quindi **requisiti stabili**, non permette modifiche sostanziali. Più il progetto è complesso, più è facile che i requisiti iniziali vengano cambiati non banalmente e portino quindi al fallimento del progetto se viene applicato il processo software a cascata.

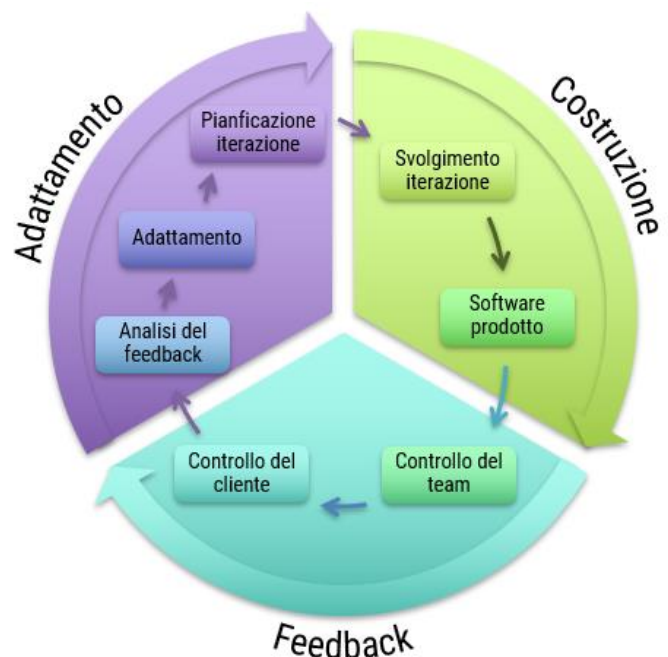
## Sviluppo iterativo, incrementale, evolutivo

Non è un processo software specifico ma un **modello di organizzazione del processo software**. Prevede multiple **iterazioni** di durata temporale fissa (time-boxed, solitamente circa 2-6 settimane) all'interno delle quali si svolgono tutte le attività di processo (il rapporto del tempo dedicato ad ogni attività cambia in base a quante iterazioni sono già state fatte) e si produce del **software eseguibile**; tutto il materiale prodotto servirà da base per l'iterazione successiva (incrementale) e l'iterazione corrente definirà gli obiettivi e raffinerà l'organizzazione dell'iterazione successiva, per meglio adattarsi al progetto (evolutivo). In questo modo i **requisiti vengono stabilizzati nel tempo**, lasciando spazio soprattutto all'inizio per modifiche anche non banali, portando a una maggior probabilità di successo.

Lo sviluppo iterativo, incrementale, evolutivo ha diversi vantaggi:

- Elevata **flessibilità** rispetto alle modifiche;
- **Riduzione** precoce dei **rischi maggiori** (subito gestiti);
- **Progresso visibile** (eseguibile fin dalle prime iterazioni);
- **Feedback precoce**, coinvolgimento dell'utente;
- Migliore **gestione della complessità**.

Il tutto porta a una minor probabilità di fallimento del progetto.



## Iterazione

Ogni iterazione comporta la scelta di un **sottoinsieme di requisiti**, la **progettazione**, l'**implementazione** e il **testing**. È time-boxed: ha durata fissa, in caso di problemi si adatta la quantità di lavoro, non di tempo. In genere un'iterazione dura **2-6 settimane**. La suddivisione del tempo tra le varie attività varia in base alla fase di sviluppo: all'inizio si dedicherà più tempo all'analisi dei requisiti, mentre più avanti verrà dedicato più tempo all'implementazione, per esempio.

Durante un'iterazione i **requisiti vengono fissati** (durante la pianificazione) e sono bloccati durante tutta la durata della stessa; questo è fatto per permettere al team di lavorare senza interruzioni. Eventualmente a metà iterazione il team può cambiare il piano della stessa valutando il lavoro rimasto e la fattibilità dello stesso entro i tempi dell'iterazione. La **pianificazione è guidata dal rischio** (bisogna ridurre i rischi maggiori e stabilizzare quanto prima il nucleo dell'architettura) e **dal cliente** (si sviluppano prima le caratteristiche per lui prioritarie).

## Unified Process (UP)

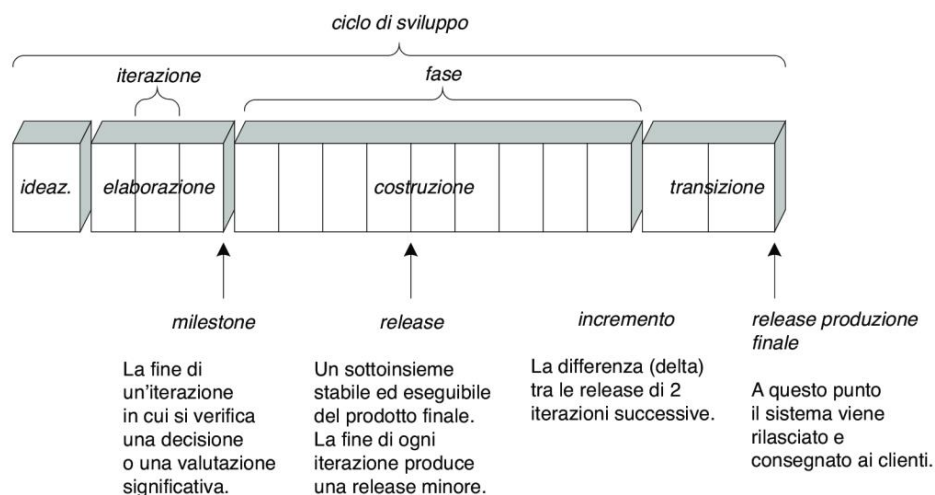
Diffuso **processo iterativo, incrementale ed evolutivo** per lo sviluppo del **software orientato agli oggetti**. Prende elementi anche da altri processi software (Extreme Programming, Scrum...), è flessibile (tutto è opzionale o quasi, a parte il codice – lo scenario di sviluppo descrive quali pratiche si adottano) e aperto alle pratiche di altri metodi iterativi. È **pilotato dai Casi d'Uso (requisiti) e dai rischi, incentrato sull'architettura**.

Le pratiche fondamentali di UP, tra le altre, comprendono l'affrontare durante le prime iterazioni i **rischi maggiori** e creare un'architettura coesa; verificare la qualità attraverso **test** ove possibile realistici fin da subito; coinvolgere continuamente l'utente e gestire le richieste di cambiamento, con attenzione particolare ai requisiti. Inoltre come strumento di modellazione visuale si utilizza **UML**.

## Fasi

Up divide le varie iterazioni (organizza il lavoro) in quattro gruppi detti fasi:

- **Ideazione:** avvio del progetto, visione approssimativa, studio economico, portata, stime approssimazione dei costi e dei tempi;
- **Elaborazione:** realizzazione del nucleo dell'architettura, visione raffinata, identificazione di gran parte dei requisiti;
- **Costruzione:** realizzazione delle capacità operative iniziali e successiva preparazione al rilascio;
- **Transizione:** completamento del prodotto.





### Attenzione alla differenza fra fase e disciplina!

UP ha 4 **fasi** in totale, che comprendono ciascuna più iterazioni:

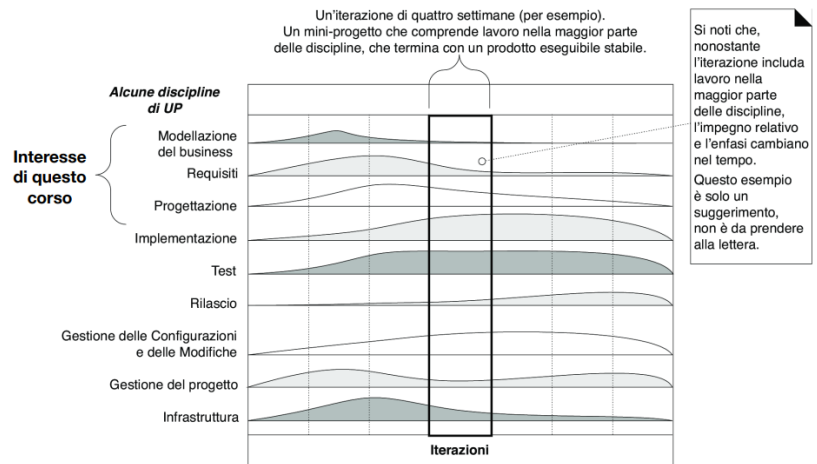
**ideazione – elaborazione – costruzione – transizione.**

Le **discipline** sono le varie attività che vengono svolte all'interno dell'iterazione (modellazione del business, requisiti, progettazione, implementazione...).

## Iterazioni

Ogni **iterazione** è una **finestra di tempo** all'interno della quale si completa un mini-progetto (ad eccezione della prima iterazione – fase di ideazione – dove non si produce software eseguibile) che include:

- Pianificazione;
- Analisi e progettazione;
- Costruzione;
- Integrazione e test;
- Release (insieme di manufatti previsti e approvati, base per il lavoro futuro).



Il prodotto finale è dato da **molteplici iterazioni** (che si possono sovrapporre in parallelo per grandi squadre). Il tempo dedicato ad ogni disciplina nel "mini-progetto" di un'iterazione varia in base al numero di iterazioni già fatte e alla fase in cui è il progetto: all'inizio si dedica più tempo ai requisiti e più avanti all'implementazione e al testing, per esempio.

## Cattive pratiche di UP

- Si tenta di definire tutti i requisiti all'inizio (cascata)
- Si realizza il nucleo dell'architettura e non lo si testa
- Si pensa che gli UML siano dettagliati a livello codice (livello di dettaglio linguaggio di programmazione)
- Si pensa che UP consista nel produrre più elaborati possibile
- Si pensa che ogni fase abbia una sola attività e/o viceversa
- Si cerca di finire il progetto prima dell'implementazione

## Metodo agile

Lo **sviluppo agile** è una **forma di sviluppo** iterativo che incoraggia l'agilità – ovvero una **risposta rapida e flessibile ai cambiamenti**, adottabile da qualsiasi processo iterativo.

Tra i suoi principi troviamo:

- Sviluppo e pianificazione **iterativa**;
- Consegne **incrementali**;
- Valori agili: **semplicità**, leggerezza, **valore delle persone**, facilità di comunicazione... *\*cof cof\**
- Pratiche agili: **modellazione UML** solo delle parti complesse del progetto, programmazione a coppie, **Test Driven Development**, **refactoring**...

## UP agile

Si può adottare **UP con spirito agile**, oltre alle solite cose si hanno:

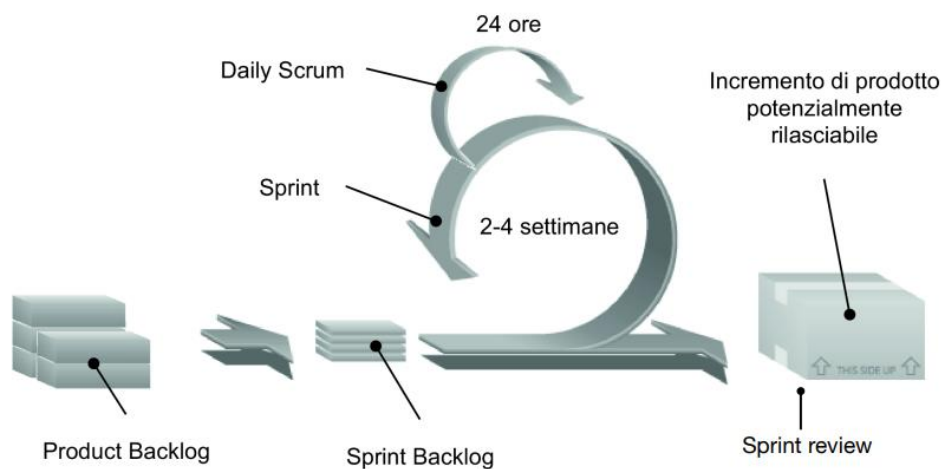
- Un piccolo insieme di attività ed elaborati;
- I **requisiti** e la **progettazione** non vengono completati prima dell'**implementazione**, ma emergono in modo adattivo durante una serie di iterazioni, anche sulla base di **feedback**;
- Applicazione di UML secondo lo spirito della modellazione agile (ossia **fare solo ciò che serve**).

## SCRUM

**Metodo agile** che consente il rilascio di prodotti software con il più alto valore per i clienti nel minor tempo possibile; si occupa perlopiù dell'**organizzazione** e della **gestione del progetto**. Uno "**scrum**" è un **incontro quotidiano** per il team dove si stabiliscono le priorità del giorno basandosi sul **backlog** (ossia la lista delle cose da fare); le iterazioni sono dette "**sprint**" (da 2 a 4 settimane) e la quantità di lavoro rimanente fattibile in un singolo sprint è chiamata "velocity".

Ci sono vari ruoli:

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



# UNIFIED PROCESS

## Overview

In questo corso tratteremo dell'Unified Process, occupandoci principalmente delle fasi di ideazione ed elaborazione con le relative discipline ed elaborati.

### Tabella dell'Unified Process – Fasi, discipline, elaborati

In giallo viene indicata la fase in cui viene iniziato l'elaborato, in verde quella in cui (eventualmente) viene raffinato. Trattiamo solo delle discipline a noi utili.

			Fasi			
Disciplina	Artefatto	Descrizione	Ideazione (I1)	Elaborazione (E1...En)	Costruzione (C1...Cn)	Transizione (T1...T2)
Modellazione del Business	<i>Modello di Dominio</i>	Diagramma delle classi concettuali.				
Requisiti	<i>Modello dei Casi d'Uso</i>	<ul style="list-style-type: none"> <li>- <b>Diagramma dei Casi d'Uso</b>: rappresentazione visiva delle relazioni fra attori e Casi d'Uso;</li> <li>- <b>Casi d'Uso</b>: descrivono in linguaggio naturale ciò che il sistema deve fare (requisiti funzionali);</li> <li>- <b>Diagrammi di Sequenza di Sistema</b>: identificano i messaggi per le operazioni di sistema, saranno i messaggi iniziali dei Diagrammi di Interazione;</li> <li>- <b>Contratti</b>: complementari ai Casi d'Uso per chiarire cosa devono compiere gli oggetti di dominio (post-condizioni da raggiungere).</li> </ul>				
	<i>Visione</i>	Stime di costi e tempi di sviluppo.				
	<i>Specifiche Supplementare</i>	Descrizione dei requisiti non funzionali.				
	<i>Glossario</i>	Raccolta di termini del progetto con la definizione.				
Progettazione	<i>Modello di Progetto</i>	<ul style="list-style-type: none"> <li>- <b>Diagramma delle Classi Software</b>: diagramma delle classi che verranno implementate, con loro dipendenze e attributi.</li> <li>- <b>Diagrammi di Interazione</b>: mostrano le interazioni, tramite messaggi, di oggetti.</li> </ul>				
	<i>Documento sull'Architettura Software</i>	Comprende le varie viste dell'architettura ( <b>logica</b> , di rilascio, di processo...); definisce le "scelte importanti" del progetto (su larga scala) riguardanti i principali problemi architetturali e le scelte di progettazione del sistema (con motivazione).				
	<i>Modello dei Dati</i>	Diagramma con gli schemi dei database e le specifiche del salvataggio-recupero di dati.				
Implement.	...					
Gestione del progetto	...					
...	...					

In UP va ricordato che è tutto **praticamente opzionale**. Questo è lo schema sul libro, ma molto più bello.

Qui sotto c'è il diagramma degli **elaborati di UP** – non sono tutti e vengono mostrate solo alcune relazioni, ma è comunque utile.



# PRIMA FASE: L'IDEAZIONE

Ideazione <i>"Immaginare la portata del prodotto, la visione e lo studio economico"</i>	
N. iterazioni	1
Durata realistica	Non più di qualche settimana per la maggior parte dei progetti
Cosa si dovrebbe fare	<p><b>Discipline: modellazione del business, requisiti</b></p> <p>Si dovrebbero fare 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 (se si ritiene che si possa proseguire con il progetto, che sia fattibile).</p>
Documenti prodotti o modificati	<ul style="list-style-type: none"> <li>• <b>Visione e studio economico:</b> descrive obiettivi e vincoli di alto livello, lo studio economico e fornisce un sommario del progetto;</li> <li>• <b>Modello dei Casi d'Uso:</b> descrive i requisiti funzionali (SSD e Contratti non vengono fatti!);</li> <li>• <b>Specifiche supplementari:</b> altri requisiti non funzionali, in particolare quelli che hanno impatto significativo sul nucleo dell'architettura;</li> <li>• <b>Glossario:</b> dizionario dei dati e terminologia chiave del dominio;</li> <li>• <b>Lista dei rischi e Piano di gestione dei rischi:</b> descrive rischi di tutti i tipi e le idee per attenuarli;</li> <li>• <b>Prototipi e proof of concept:</b> per chiarire la visione e validare tecniche;</li> <li>• <b>Piano dell'iterazione:</b> piano della prima iterazione della fase di elaborazione;</li> <li>• <b>Piano delle fasi e piano di sviluppo del software:</b> ipotesi approssimative sulla durata e lo sforzo della fase di elaborazione e le risorse necessarie (strumenti, persone, formazione...);</li> <li>• <b>Scenario di sviluppo:</b> descrizione della personalizzazione dei passi e degli elaborati di UP per il progetto.</li> </ul>
Errori comuni	<ul style="list-style-type: none"> <li>• Dura più di qualche settimana;</li> <li>• Provi a definire molti requisiti;</li> <li>• Ci si aspetta che i piani e le stime siano affidabili;</li> <li>• Viene definita l'architettura del sistema;</li> <li>• Vieni influenzato dal pensiero a cascata nelle attività;</li> <li>• Manca la visione o lo studio economico;</li> <li>• I nomi di molti attori o Casi d'Uso non sono stati identificati;</li> <li>• La maggior parte dei Casi d'Uso (o nessuno) sono stati definiti in dettaglio.</li> </ul>

In UP l'**ideazione** è il **passo iniziale** che permette di definire la visione del progetto. Lo scopo dell'ideazione è quello di raccogliere informazioni per avere una **visione comune** e decidere se il progetto è **fattibile** ed è il caso di procedere con un'indagine seria nella fase di elaborazione.

È composta da un'unica **iterazione**, detta **iterazione zero**, differente dalle altre perché non punta a creare software eseguibile, ma si concentra sull'avvio di attività ed elaborati. Oltre a fare una **stima approssimativa dei costi e dei tempi di sviluppo** comprende anche l'**analisi del 10% circa dei requisiti funzionali** (Casi d'Uso) e di quelli **non funzionali più critici** (anche se la maggior parte vengono definiti nella fase di elaborazione, in parallelo alle prime attività di qualità-produzione e di test). È molto importante notare che nella fase di ideazione viene **pianificata la prima iterazione della fase di elaborazione**, dove si inizierà a produrre software eseguibile.

Lo scopo dell'ideazione non è dunque quello di definire tutti i requisiti (seppur viene fatto un primo workshop sui requisiti): non sono fatti per la maggior parte in modo dettagliato, servono a fornire una visione approssimativa dei requisiti funzionali richiesti (quindi non si modella molto con UML); verranno poi raffinati nella fase di elaborazione. È tuttavia importante **definire gran parte degli attori** perché questo aiuta a definire la portata del progetto.

# Requisiti

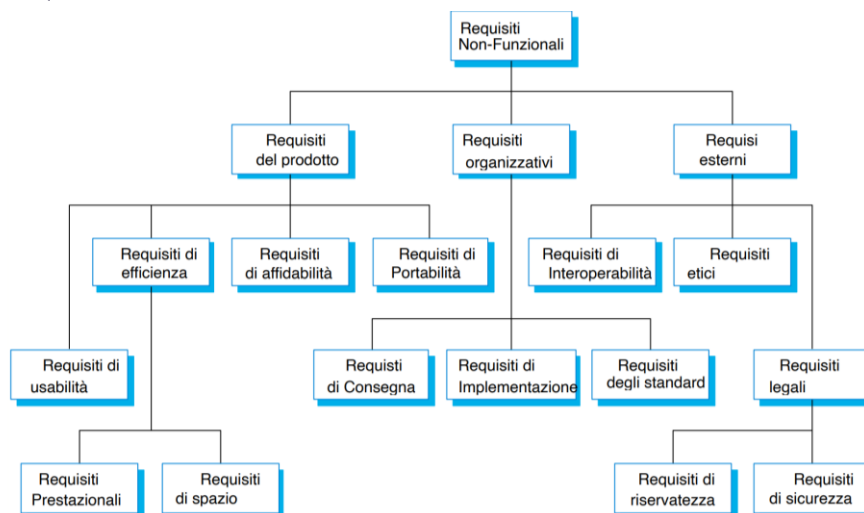
## I tipi di requisiti

Ogni sistema software deve risolvere uno o più determinati problemi e per far ciò deve fornire un certo numero di **funzionalità** e possedere alcune **caratteristiche di qualità**. Un **requisito** è una **capacità** o una **condizione** a cui un sistema, e quindi il progetto, deve essere **conforme**.

I requisiti devono essere **completi** rispetto alle richieste del cliente, **non ambigui** e **coerenti** fra loro. Ci sono due tipi di requisiti:

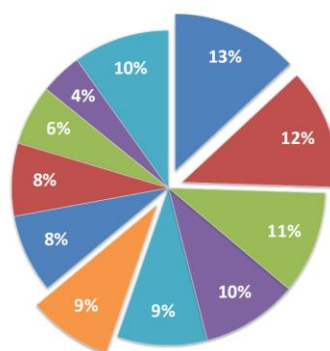
- **Requisiti funzionali:** definiscono **funzionalità o servizi** che il sistema deve fornire, risposte che l'utente si aspetta in determinate condizioni e output attesi a fronte di determinati input. Gli elaborati prodotti in merito sono i **Casi d'Uso**;
- **Requisiti non funzionali:** sono requisiti relativi alle **proprietà del sistema**. Possono riguardare vincoli del sistema a livello di affidabilità e rapidità o vincoli su I/O, oppure vincolano gli standard di qualità del progetto, o l'IDE da usare, ecc; possono riguardare quindi una singola parte dell'architettura o tutto il sistema. Gli elaborati prodotti sono le **Specifiche Supplementari**.

I requisiti non funzionali sono spesso più vincolanti rispetto ai requisiti funzionali; un requisito non funzionale può generare una serie di requisiti funzionali correlati che servono a soddisfarlo.



I requisiti sono estremamente importanti in un progetto. UP si basa sulla **flessibilità**: i primi requisiti vengono individuati (insieme al cliente) in prima battuta nell'ideazione e il resto per la maggior parte durante l'elaborazione. Non è da escludere che possano essere **modificati** più tardi, ma i requisiti modellando con UP **tendono a stabilizzarsi con il tempo**.

## Cause del fallimento dei progetti software



- **Requisiti incompleti**
- **Mancanza di coinvolgimento dell'utente**
- **Mancanza di risorse**
- **Aspettative non realistiche**
- **Mancanza di supporto dell'esecutivo**
- **Cambiamento dei requisiti & specifiche**
- **Mancanza di pianificazione**
- **Caduta di interesse**
- **Assenza del management IT**
- **Carenza di conoscenza tecnologica**
- **Altro**

il 34% delle cause dei fallimenti dei progetti software riguardano l'attività dei requisiti

Statistica fondamentale: il **25% dei requisiti** di un progetto **cambia** dopo l'ideazione.



## Gli obbiettivi

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

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

Gli **obbiettivi** sono utili agli sviluppatori in quanto trasmettono le **intenzioni degli utenti**; sulla base di questi possono essere **definiti** ulteriori **requisiti non funzionali verificabili**, che servono a verificare con criteri oggettivi tali obbiettivi.

## Acquisizione e redazione dei requisiti

UP incoraggia un'acquisizione dei requisiti **agile**, tramite:

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

Gli **elaborati dei requisiti** devono essere scritti tramite **frasi in linguaggio naturale** (formato testuale) comprensibili, integrati eventualmente con diagrammi, sia per gli sviluppatori che per i clienti. Va ideato un **formato standard** per la struttura testuale del requisito (per esempio, viene indicato il tipo di requisito? Comincia con un verbo? Ha un titolo?). Si utilizza il verbo **deve** per i requisiti obbligatori, **dovrebbe** per quelli desiderabili; è utile l'utilizzo di **enfasi testuale** (grassetto, corsivo, evidenziatore, sottolineatura...) per identificare le parti importanti e non va assolutamente usato gergo informatico (poco chiaro al cliente).

## [D] I Casi d'Uso

I **Casi d'Uso** sono storie scritte, preferibilmente di lunghezza ridotta, che costituiscono **dialoghi** fra un **attore** o più e un **sistema** che svolge un **compito**. Vengono utilizzati per **scoprire e registrare i requisiti funzionali** (ma possono eventualmente descrivere anche alcuni requisiti non funzionali verificabili) e fanno da base a molti elaborati successivi. I Casi d'Uso non sono diagrammi, sono dei testi; non sono orientati agli oggetti. Un Caso d'Uso definisce un contratto relativo al comportamento del sistema.

UP incoraggia l'utilizzo e la modellazione dei **Casi d'Uso**; ricordiamo che nella pianificazione iterativa la progettazione è proprio guidata dai requisiti. UP definisce come elaborato il **Modello dei Casi d'Uso**, ossia l'insieme di tutti i Casi d'Uso (detto **testo dei Casi d'Uso**) descritti ed eventualmente un **diagramma UML** degli stessi, nell'ambito della disciplina dei Requisiti. Se presenti, il Modello dei Casi d'Uso comprende anche i **Diagrammi di Sequenza di Sistema** e i **Contratti**. Si può stilare una tabella attori-obbiettivi per avere una visione d'insieme.

Durante l'ideazione solo il 10%-20% circa dei Casi d'Uso, ossia quelli più influenti sull'architettura, dovrebbero essere dettagliati; questo in quanto, iniziata la fase di elaborazione, la progettazione e l'implementazione iniziano dai Casi d'Uso o dagli scenari più significativi.

## Perché li usiamo?

Scriviamo i Casi d'Uso poiché:

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

## Definizioni utili

Diamo alcune definizioni utili per i Casi d'Uso:

- **Attore**: qualcosa o qualcuno **dotato di un comportamento**. Anche il sistema in discussione (SuD - System Under Discussion) è considerato un attore, quando ricorre ai servizi di altri sistemi;
- **Scenario** (istanza di Caso d'Uso): **sequenza specifica di azioni e interazioni** tra il sistema e alcuni attori; Uno scenario è un percorso all'interno del Caso d'Uso, sia di **successo** che di **fallimento**. Uno scenario è formato da una sequenza di passi, che possono essere di 3 tipi:
  - Un'interazione tra attori.
  - Un cambiamento di stato da parte del sistema.
  - Una validazione.
- **Caso d'Uso**: È una **collezione di scenari** correlati, sia di successo che di fallimento.

In particolare, all'interno di un Caso d'Uso possiamo distinguere diversi attori:

- **Attore primario**: **Utilizza** direttamente i servizi del SuD, affinché vengano raggiunti degli obiettivi utente;
- **Attore finale**: Vuole che il SuD **venga utilizzato** affinché vengano raggiunti dei suoi obiettivi;
- **Attore di supporto**: Offre un **servizio** al SuD;
- **Attore fuori scena**: Ha un **interesse nel comportamento** del Caso d'Uso, ma non è nessuno dei 3 tipi precedenti, né interviene all'interno del Caso d'Uso.

Molto spesso **attore primario e finale** di un Caso d'Uso **coincidono**, perché l'attore primario usa direttamente il SuD per raggiungere i propri obiettivi, e dunque è anche l'attore finale.

## Tipologie di Casi d'Uso

Possiamo scrivere i Casi d'Uso (ricordiamo che sono scritti, non disegnati!) a diversi **livelli di dettaglio e formalità**:

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

Inoltre distinguiamo i vari Casi d'Uso in base al loro **livello**, che permette di identificare la **"scala" del Caso d'Uso**:

- **Livello di obiettivo utente**: consente all'utente di raggiungere un proprio **obiettivo**;
- **Livello di sotto-funzione**: rappresenta solo una **funzionalità** nell'uso del sistema, sono interazioni di dettaglio;
- **Livelli di sommario**: riguarda un obiettivo più ampio, non direttamente raggiungibile con un singolo utilizzo del sistema. Sono utili per comprendere il **contesto** di più Casi d'Uso.



Per quanto riguarda l'organizzazione visiva del testo possiamo avere varie forme di rappresentazione, come il formato a 2 colonne, che enfatizza la conversazione tra gli attori e il sistema. Noi usiamo il **formato testuale a tabella**, livello dettagliato (ma più corto: non scriviamo 3-10 pagine, solo una o due, i nostri sistemi sono piccoli).

Per quanto riguarda la forma mentis, ragionare a **scatola nera** è l'approccio più usato: non si descrive il comportamento interno del sistema, ma si descrivono le sue responsabilità. In questo modo nella stesura dei Casi d'Uso è possibile **specificare che cosa** deve fare il sistema (= analisi) **senza decidere come** il sistema lo farà (=progettazione).

### Template per un Caso d'Uso dettagliato

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

### Stesura di un Caso d'Uso

Per stilare un Caso d'Uso efficace bisogna:

- Scegliere i **confini** di sistema (anche attraverso attori esterni);
- Identificare gli **attori primari**;
- Identificare gli **obiettivi** di ciascun attore primario;
- Definire i **Casi d'Uso** che **soddisfano gli obiettivi** degli utenti; il loro nome va scelto in base all'obiettivo.

Bisogna scrivere in modo **essenziale** ma **completo** (chiara indicazione di soggetto-verbo-oggetto e subordinate), **non fraintendibile**, e concentrarsi sullo **scopo reale** dell'utente (risultati che per l'utente hanno valore). La narrativa di un Caso d'Uso viene espressa a livello di **intenzioni dell'utente** e delle **responsabilità del sistema**, anziché con riferimento ad azioni concrete: questo significa che non dobbiamo fornire alcun dettaglio implementativo nei Casi d'Uso. Va stilato un **Caso d'Uso** generalmente **per ogni obbiettivo utente**; un'eccezione comune sono le operazioni CRUD che vengono raggruppate in un caso chiamato "gestisci X".

## Verificare l'utilità di un Caso d'Uso

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

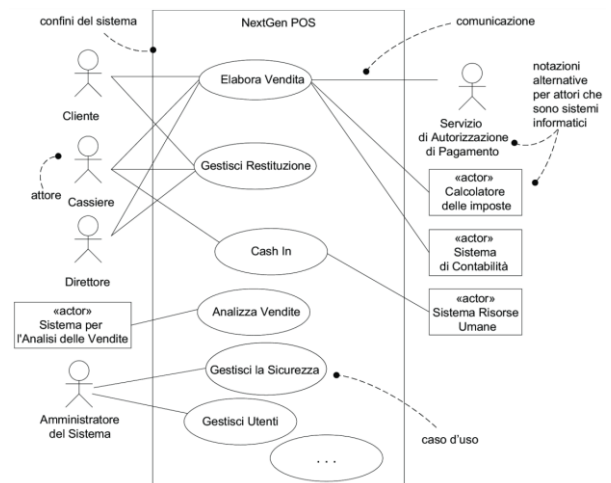
- Il **test del capo**. Se il tuo capo ti chiede "Cos'hai fatto tutto il giorno?" e tu rispondi "*nomeCasoD'uso!*" il tuo capo è felice? Se un Caso d'Uso non supera il test del capo, ma è un Caso d'Uso complicato, vale comunque la pena tenerlo;
- Il **test EBP, Elementary Business Process**, si concentra sul valore aziendale misurabile. Si controlla che **non sia un singolo passo** (tra i 5 e i 10), che sia un'attività **eseguibile in una sola sessione** di lavoro, in risposta a un evento business che **genera valore business** osservabile e misurabile, e che lasci il sistema in uno **stato stabile e coerente**;
- Il **test della dimensione**: un Caso d'Uso non deve essere composto da un solo "passo", deve avere tra le 3 e le 10 pagine di testo.

I test sopraelencati possono essere **violati** nel caso in cui abbiamo dei Casi d'Uso a livello di **sotto-funzione** (attività secondarie) che risultano utili a evitare ripetizioni o a spiegare operazioni piccole ma complesse.

## [D] Diagramma dei Casi d'Uso

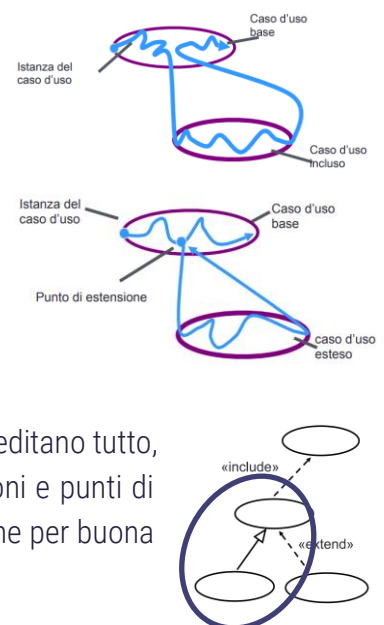
I diagrammi dei Casi d'Uso e le relazioni tra i Casi d'Uso sono secondari rispetto al testo degli stessi.

Un **diagramma dei Casi d'Uso** costituisce un buon **diagramma di contesto**, che serve a mostrare il quadro generale e i confini del sistema, ciò che giace al suo esterno e come viene utilizzato. Solitamente include solo i **Casi d'Uso a livello di obiettivo utente**. In linea ideale le associazioni dovrebbero essere orientate da chi inizia l'interazione all'oggetto della stessa. Se non è orientato, entrambe le parti possono iniziare l'interazione.



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

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



## FASE 2.1: ELABORAZIONE – 1° ITERAZIONE

Elaborazione 1° iterazione	<i>"Costruire il modello dell'architettura, risolvere gli elementi ad alto rischio, definire la maggior parte dei requisiti e stimare il piano di lavoro e le risorse complessive."</i>		
N. iterazioni	1 (2...n per l'intera fase)	Durata realistica	2-6 settimane (alcuni mesi per l'intera fase)
Cosa si dovrebbe fare	<b>Discipline: modellazione del business, requisiti, progettazione, implementazione</b> Si dovrebbe programmare, verificare e testare il nucleo dell'architettura (altresì detto baseline dell'architettura o architettura eseguibile); si stabilizzano la maggior parte dei requisiti (circa 80%) e si attenuano i rischi maggiori.		
Documenti prodotti o modificati	<ul style="list-style-type: none"> <li>• <b>Modello di Dominio:</b> visualizza i concetti del dominio sottoforma di diagramma;</li> <li>• <b>Modello di Progetto:</b> è l'insieme dei diagrammi che descrivono la progettazione logica, quindi comprende il Diagramma delle Classi Software, i Diagrammi di Interazione, il diagramma dei package e così via;</li> <li>• <b>Documento dell'Architettura:</b> riassume le varie viste dell'architettura e comprende tutte le decisioni significative riguardanti il progetto, incluse le motivazioni;</li> <li>• <b>Modello dei dati:</b> schema delle basi di dati e informazioni sul salvataggio e il recupero dei dati (oggetti);</li> <li>• <b>Storyboard dei Casi d'Uso, Prototipi UI:</b> sono la descrizione dell'interfaccia utente, della navigazione e dell'accessibilità etc;</li> <li>• Vengono raffinati il modello dei Casi d'Uso (e cominciate gli SSD e i Contratti!), la visione, le specifiche supplementari e il glossario.</li> </ul>		
Errori comuni (per la fase d'elaborazione)	<ul style="list-style-type: none"> <li>• Ha una durata superiore ad "alcuni" mesi per la maggior parte dei progetti;</li> <li>• Ha una sola iterazione (con rare eccezioni per problemi di chiara comprensione);</li> <li>• Le fondamenta dell'architettura e gli elementi rischiosi non vengono affrontati;</li> <li>• Non produce un'architettura eseguibile;</li> <li>• Il codice non è di qualità di produzione;</li> <li>• È considerata principalmente una fase di analisi dei requisiti, che precede una fase di implementazione, la costruzione;</li> <li>• Prova a fare una progettazione completa prima dell'implementazione;</li> <li>• Feedback e adattamento minimi; gli utenti non sono coinvolti in modo continuo;</li> <li>• Non c'è test precoce e realistico;</li> <li>• È considerata una fase in cui vengono prodotti prototipi per dimostrare idee, e non la baseline dell'architettura eseguibile;</li> <li>• Non ci sono vari brevi workshop dei requisiti, che adattano e raffinano i requisiti basandosi sul feedback delle iterazioni precedenti e corrente.</li> </ul>		

Durante l'**elaborazione** si esegue un'**indagine seria e completa**; è composta da 2...\* iterazioni, ciascuna dalle 2 alle 6 settimane. Non è la fase della costruzione ma, in pieno spirito iterativo, incrementale ed evolutivo (che prevede la produzione di codice prima della conclusione dell'analisi dei requisiti), **si produce l'eseguibile** del nucleo dell'architettura, che ha già **qualità di produzione** e viene già **testato realisticamente**.

L'iterazione numero 1 è **centrata sull'architettura**, è **guidata dal rischio**, e affronta gli aspetti più difficili e rischiosi del progetto. Nel libro e nel corso l'iterazione 1 è guidata dagli obiettivi di apprendimento (concetti OOA/D = Object Oriented Analysis and Design).

I requisiti che si andranno a realizzare e implementare nella prima iterazione sono parte dei requisiti/Casi d'Uso dettagliati durante l'ideazione, nello specifico quelli scelti nella pianificazione dell'iterazione fatta appunto durante l'iterazione 0. Per le **pianificazioni delle iterazioni successive** si andranno a scegliere i **requisiti da realizzare** in base al **livello di rischio**, alla **copertura** (Casi d'Uso che riguardano l'intero sistema e non funzioni specifiche) e alla **criticità** (valore di business per il cliente). Ai Casi d'Uso viene talvolta dato un voto che viene usato insieme ai precedenti criteri per stabilire la priorità degli stessi.

## Modellazione del Business

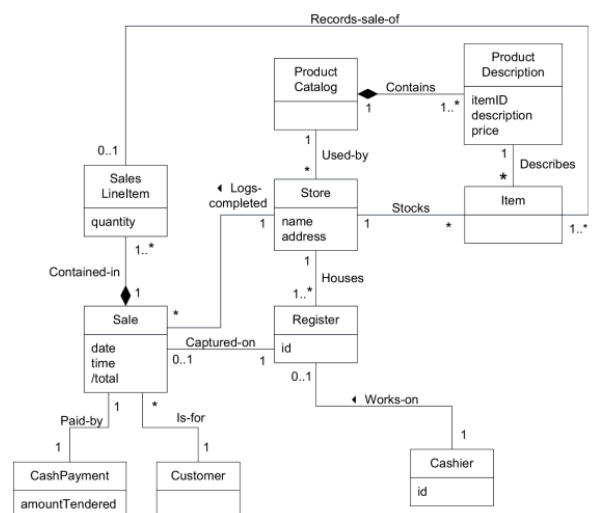
### Analisi a oggetti

L'analisi riguarda l'investigazione e la comprensione del problema che il sistema deve risolvere. L'analisi orientata agli oggetti 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**, ossia i cambiamenti di stato del sistema conseguenze delle funzioni (Contratti).

### [D] Modello di Dominio

Il **Modello di Dominio** è il documento principale dell'analisi a oggetti. È una **rappresentazione visuale** (diagramma) delle **classi concettuali** (non software), che mostrano i concetti del mondo reale nonché le loro relazioni tra di essi nel dominio di interesse; può essere visto come un “**dizionario visuale**”. Influenza molti altri elaborati, tra cui il più notevole è il Diagramma delle Classi Software, incluso nel Modello di Progetto (senza togliere il fatto che influenza tutto lo strato di dominio a livello software); la realizzazione o l'espansione del Modello di Dominio è **limitata dai requisiti dell'iterazione corrente** (non si fa di più, né di meno). Viene usato per modellare il dominio all'interno del quale opera il sistema.



Nel Modello di Dominio troviamo:

- Le **classi concettuali**;
- Gli **attributi** delle classi concettuali (eventualmente inesistenti);
- Le **associazioni** tra le classi concettuali.

### Perché lo usiamo?

Usiamo il Modello di Dominio:

- **Nell'analisi** per **comprendere il dominio** del sistema da realizzare e per definire un **linguaggio comune** sulle parti interessate al sistema;
- **Nella progettazione** come **fonte d'ispirazione** per la progettazione dello strato del dominio.

## Le classi concettuali

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

- Un **simbolo**: parola o immagine che rappresenta la classe;
- Un'**intensione**: definizione in linguaggio naturale della classe e delle sue proprietà;
- Un'**estensione**: l'insieme degli oggetti descritti dalla classe.

(Sì, è intensione, non intenzione!)

In altre parole una *classe* è un **descrittore** per un **insieme di oggetti** che possiedono le **stesse caratteristiche**. Le istanze di una classe sono dette **oggetti**. I nomi delle classi concettuali sono da scegliere accuratamente: infatti andranno a influenzare i nomi delle corrispettive classi software nel Modello di Progetto. Utilizzare nomi di classi software nello strato del dominio ispirati ai nomi del Modello di Dominio comporta un cosiddetto "basso salto rappresentazionale" e una conseguente facilità di lettura e comprensione d'insieme.

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

## Gli attributi

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

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

La visibilità di default è **private**; se il valore di un attributo è derivabile da altri attributi si aggiunge il prefisso "/" al suo nome. La maggior parte degli attributi dovrebbe essere di un tipo primitivo, non complesso: Boolean, Date, Number, Character, String, Time; le quantità numeriche dovrebbero avere il proprio tipo di dati (money, length...).

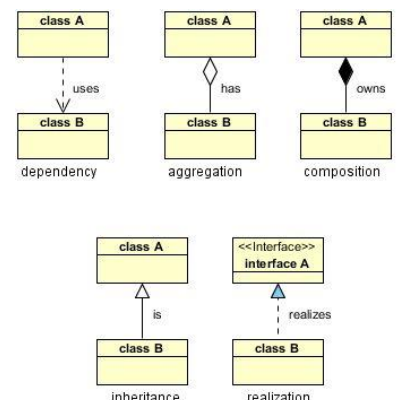
## Le classi descrizione

Una **classe descrizione** è una speciale **classe concettuale** che **contiene informazioni** su qualcos'altro. Su RSAD sono i **Datatype**. Conviene introdurre una classe descrizione quando:

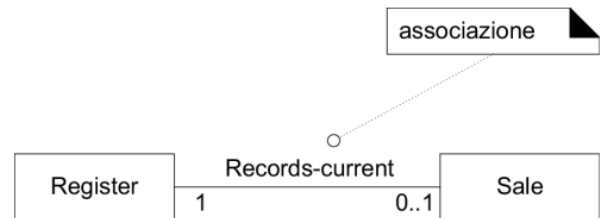
- È necessaria una **descrizione** di un articolo o servizio, **indipendentemente dall'attuale esistenza di istanze** di tali articoli/servizi;
- L'**eliminazione delle istanze** degli oggetti che sono descritti darebbe luogo a una **perdita di informazioni**;
- Si vuole **ridurre la ridondanza** di informazioni.

## Le associazioni tra classi concettuali

Un'**associazione** è una **relazione semantica tra classi**, che comporta connessioni tra le rispettive istanze. Le **istanze di un'associazione** sono dette **collegamenti**. Le associazioni vengono mostrate se le stesse devono essere memorizzate nel sistema, indipendentemente dalla loro durata, e sono significative (non ridondanti o derivabili). Un'associazione non rappresenta una decisione sulle variabili d'istanza di oggetti software, né su vincoli di chiave esterna nelle basi di dati o sui flussi di dati: in sostanza le relazioni tra classi concettuali, per via della loro natura semantica, non corrispondono alle relazioni tra le classi software, anche se possono influenzare i percorsi di navigazione e la visibilità.



Un'associazione è rappresentata da una **linea**, eventualmente orientata per indicare il senso di lettura (ma non implica nulla su visibilità o altro), **etichettata** con il nome della relazione, che può avere eventualmente dei **ruoli** (vicino alle classi) e le **cardinalità** (o **molteplicità**; vengono spesso trasposte a livello software).



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

- **Aggregazione**: tipo di associazione/relazione intero-parte. L'aggregato può **esistere indipendentemente** dalle parti e viceversa; le parti possono essere **condivise** da più aggregati. L'aggregato è **incompleto** se mancano alcune delle sue parti;
- **Composizione**: la composizione è un tipo forte di aggregazione intero-parte e implica che:
  - Ogni istanza della parte deve **appartenere sempre e solamente** a una istanza del composto;
  - Il composto ha la **responsabilità** (operazioni CRUD) rispetto alle sue parti;
  - Il composto può rilasciare una delle parti se un altro composto se ne prende la responsabilità.

Relazione	Esempio associazione		
	<i>POS Nextgen</i>	<i>Monopoly</i>	<i>Airport</i>
A è un membro di B	<i>Cashier—Store</i>	<i>Player— Monopoly</i>	<i>Game Pilot—Airline</i>
A è una transazione correlata a un'altra transazione B	<i>CashPayment—Sale</i>	<i>Cancellation— Reservation</i>	---
A è un elemento/riga di una transazione B	<i>SalesLineItem—Sale</i>	---	---
A è un prodotto o servizio per una transazione (o riga per l'articolo) B	<i>Item—SalesLineItem</i>	---	<i>Flight—Reservation</i>
A è un ruolo relativo a una transazione B	<i>Customer—Payment</i>	---	<i>Passenger—Ticket</i>
A è una parte fisica o logica di B	<i>Drawer—Register</i>	<i>Square—Board</i>	<i>Seat—Airplane</i>
A è contenuto fisicamente o logicamente in B	<i>Item—Shelf</i>	<i>Square—Board</i>	<i>Passenger—Airplane</i>
A è una descrizione per B	<i>ProductDescription—Item</i>	---	<i>FlightDescription—Flight</i>
A è una sottounità organizzativa di B	<i>Department—Store</i>	---	<i>Maintenance—Airline</i>
A utilizza o gestisce o possiede B	<i>Cashier—Register</i>	<i>Player—Piece</i>	<i>Pilot—Airplane</i>
A è vicino/prossimo a B	<i>SalesLineItem— SalesLineItem</i>	<i>Square—Square</i>	<i>City—City</i>

### Stesura del Modello di Dominio

Per realizzare il Modello di Dominio, dopo aver stabilito un determinato livello di astrazione, bisogna:

- **Trovare le classi concettuali**: per farlo si può
  - Usare **modelli esistenti**, ossia modelli di dominio ben congegnati e resi pubblici;
  - Identificare **nomi e locuzioni nominali** tramite l'analisi linguistica;
  - Utilizzare un **elenco di categorie comuni**;



- **Disegnare** le classi concettuali sfruttando **UML**;
- Aggiungere gli **attributi** e poi le **associazioni**.

Non esiste un elenco di classi concettuali corretto: eventuali **errori** saranno **sistemati nelle iterazioni successive** (quindi il Modello di Dominio va sempre tenuto aggiornato). Più che chiedersi se il Modello di Dominio è corretto, ci si dovrebbe chiedere se è **utile**.

### (Extra) [D] Diagramma degli Oggetti

Il **Diagramma degli Oggetti** è un modello che mostra un **insieme di oggetti**, con i loro attributi e relazioni in un dato momento. È simile a un diagramma delle classi, ma mostra oggetti, collegamenti e valori. La notazione UML è la stessa del Diagramma delle Classi.

## Requisiti

Nella fase di **Elaborazione** vengono **raffinati i Casi d'Uso** e il diagramma dei Casi d'Uso e **cominciati gli SSD e i Contratti**. Da notare che in tutto, la disciplina dei requisiti (e quindi l'analisi) non dovrebbe prendere più di qualche giorno, sul totale (qualche mese al più) richiesto dalle varie iterazioni della fase d'Elaborazione.

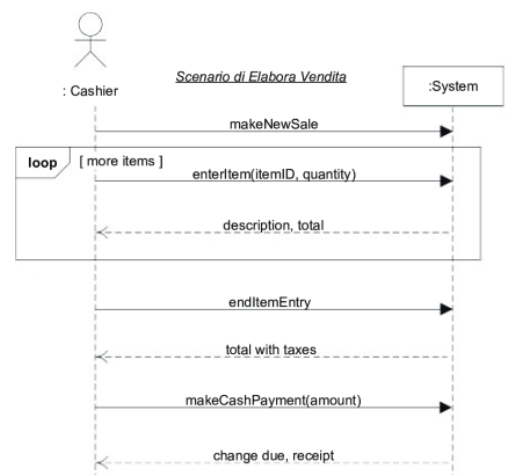
### [D] Diagrammi di Sequenza di Sistema (SSD)

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

Un sistema software reagisce tipicamente a 3 cose: **eventi esterni** da parte di attori, **eventi temporali**, **guasti o eccezioni**. Noi negli SSD ci occuperemo solo dei primi (rarissimamente dei secondi): nel momento in cui l'utente genera un cosiddetto **evento di sistema**, quest'ultimo esegue le **operazioni di sistema** nell'ordine indicato dagli SSD. Un'operazione di sistema è *una trasformazione, un'interrogazione che un soggetto o componente può essere chiamato ad eseguire* (definizione ufficiale UML). Siamo ancora a **livello di dominio**: non dobbiamo ragionare come se stessimo creando dei metodi (per esempio, i parametri delle operazioni - se presenti - non hanno tipo); come rimarcato dalla definizione di UML, è un'astrazione e non un'implementazione (**un metodo software è l'implementazione di un'operazione**).

UML fornisce la notazione dei **Diagrammi di Sequenza** per **illustrare interazioni tra attori e le loro operazioni**. Un Diagramma di Sequenza di Sistema mostra per un particolare scenario di un Caso d'Uso gli eventi generati dagli attori e le relative operazioni di sistema, in ordine. È buona norma disegnare **un SSD per lo scenario principale** di successo di ciascun Caso d'Uso, nonché per gli scenari alternativi più frequenti o complessi.

Non c'è motivo, a parte in casi particolari, di iniziare gli SSD nella fase di Ideazione; la maggior parte degli SSD vengono **creati durante l'elaborazione**, quando è utile identificare i dettagli degli eventi di sistema per chiarire quali sono le principali operazioni che il sistema deve gestire. UP non specifica di usare gli SSD, ma i creatori stessi ne riconoscono l'**utilità**.



## Linee guida

Gli eventi di sistema dovrebbero essere espressi a un **livello astratto**, di intenzione e cominciare con un **verbo**. I parametri dei messaggi dovrebbero **evitare** di essere oggetti o **concetti complessi** (meglio sceglierli semplici e spiegarli nel Glossario) o comprendere il tipo. Per quanto riguarda le risposte del sistema vanno mostrate come un elenco di dati o informazioni restituite e non come azioni.

## [D] Contratti

I **Contratti** delle operazioni descrivono nel dettaglio i **cambiamenti agli oggetti di dominio** come risultato dell'esecuzione di un'operazione di sistema. Le **operazioni di sistema** – come già detto - sono **funzionalità pubbliche** che il sistema, a scatola nera, offre tramite la sua interfaccia; l'insieme delle operazioni di sistema costituisce l'**interfaccia di sistema pubblica**. I principali input per i contratti sono le operazioni di sistema identificate negli SSD e il Modello di Dominio.

UP non indica come obbligatori i Contratti, ma come **complementari ai Casi d'Uso**; questi ultimi infatti costituiscono l'elaborato principale per la descrizione dei requisiti e i Contratti vanno stilati solo se sono necessari o utili.

I Contratti vengono iniziati e per la maggior parte conclusi nella fase d'**Elaborazione**.

## Struttura di un Contratto

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

## Post-condizioni

Le **post-condizioni** descrivono i **cambiamenti nello stato degli oggetti** nel Modello di Dominio, motivo per cui questa sezione è la più importante all'interno di un contratto. Non sono azioni, ma **osservazioni sullo stato degli oggetti alla fine dell'operazione**. Se la sezione è vuota spesso non ha senso alcuno scrivere il contratto e aumentare il numero di elaborati. Abbiamo tre tipi di post-condizioni:

- Creazione/distruzione di istanze;
- Modifiche dei valori degli attributi;
- Formazione o rottura di collegamenti (istanze di associazione) tra istanze.

Per esprimere le post-condizioni (anche le pre-condizioni) possiamo usare il **linguaggio naturale** o un linguaggio apposito chiamato **OCL** (*Object Constraint Language*) che mette in evidenza i vincoli dell'oggetto.



# Dai Requisiti alla Progettazione

L'analisi dei requisiti OO serve a *fare la cosa giusta*, la disciplina della **progettazione** porrà invece l'accento sul *fare la cosa bene* progettando una soluzione che soddisfa i requisiti. Nello sviluppo iterativo in ogni iterazione (tranne quella dell'Ideazione) avviene il **passaggio** da un interesse **incentrato su requisiti** (o analisi) a un **interesse via via incentrato principalmente sulla progettazione e sull'implementazione**.

## Architettura Software

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

Questa definizione ufficiale ci permette di inquadrare il significato di **Architettura Software**: in pratica raccoglie tutte le **decisioni importanti**, su più punti di vista, prese per sviluppare il progetto.

### (Extra) Architetture a N+1 viste

Si parla di architettura a **N+1 viste** nel Documento dell'Architettura Software, in quanto questa è la struttura più usata per redigerlo: si descrivono N viste diverse (**N "categorie" di decisioni importanti e aspetti strutturali**) più una fissa, la **vista dei Casi d'Uso**, che serve a descrivere l'intenzione generale e lo scopo del progetto attraverso dei Casi d'Uso significativi a livello di sistema. È molto comune l'architettura a **6 viste**:

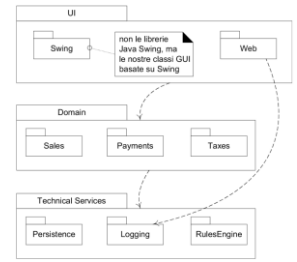
- **Architettura logica**: si preoccupa di dividere le classi software in strati/package/sottosistemi in base alla loro semantica e appartenenza a un determinato strato/package/sottosistema a livello software;
- **Architettura di processo**: classi UML e Diagrammi di Interazione che mostrano i processi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

Quindi quando parliamo di **Architettura logica** stiamo guardando solo uno **specifico aspetto dell'architettura software**.

### [D] Architettura logica

Ora che passiamo dall'analisi alla progettazione cominciamo a pensare su **larga scala**. A questo livello la progettazione di un sistema orientato agli oggetti è basata su diversi strati architetturali: l'**Architettura logica** si preoccupa di **dividere le classi software** in **strati/packages/sottosistemi** in base alla loro **semantica** e **appartenenza** a un determinato strato/package/sottosistema **a livello software**. È detta logica perché non vengono prese decisioni su come questi elementi siano distribuiti su processi/computer (sono decisioni che fanno parte dell'architettura di rilascio).

L'Architettura logica può essere illustrata sotto forma di **diagrammi dei package UML**, come parte del **Modello di Progetto**; si possono quindi annidare i package e illustrare le dipendenze tra i package tramite una linea tratteggiata che punta verso il package da cui si dipende. Un package rappresenta un **namespace**, così è possibile avere più classi con lo stesso nome in package diversi.



Possiamo dividere l'Architettura logica secondo più criteri:

- **Livello (tier)**: solitamente indica un nodo fisico di elaborazione;
- **Strato (layer)**: una sezione verticale dell'architettura;
- **Partizione (partition)**: una divisione orizzontale di sottosistema di uno strato.

### Architettura logica a strati

Uno stile comune per l'Architettura logica è l'**architettura a strati**, dove ciascuno **strato** è un **gruppo** a grana molto grossa di classi/package/sottosistemi che ha delle **responsabilità coese** rispetto ad un aspetto importante del sistema. Gli **strati inferiori** sono **servizi generali e di basso livello**, facilmente **riusabili**, quelli **superiori** sono orientati a **servizi specifici per l'applicazione**. Gli strati più alti ricorrono ai servizi degli strati più bassi.

In uno strato le **responsabilità** degli oggetti devono essere **fortemente coese** e non mischiarsi con responsabilità di altri strati. Ci sono strumenti di reverse-engineering che permettono, a partire da un base di codice, di generare automaticamente un diagramma dei package per verificarlo.

Ci sono due tipi di architetture a strati:

- **A strati stretta**: uno strato può richiamare solamente i servizi di uno **strato immediatamente sottostante**;
- **A strati rilassata**: uno strato può richiamare servizi di strati **più bassi di diversi livelli**.

Solitamente in un'applicazione abbiamo i seguenti strati:

- **Presentazione o Interfaccia Utente (UI)**;
- **Logica applicativa** o strato del dominio: elementi che rappresentano concetti del dominio (focus di questa parte del corso);
- **Servizi tecnici**.

### Perché gli strati?

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

- Modifiche del codice che si estendono a tutto il sistema: gli strati permettono quasi sempre modifiche locali senza dover modificare altri strati (indipendenza delle modifiche);
- Intrecci tra logica applicativa e UI (separation of concerns);
- Intrecci tra servizi / logica di business con logica applicativa (separation of concerns);
- Accoppiamenti tra diverse aree di interesse (low coupling).

Riassumendo i vantaggi:

- **Separations of concerns**;
- **Miglior coesione**;
- **Incapsulamento della complessità**;
- **Sostituibilità** degli strati;
- **Riusabilità** delle funzioni degli strati bassi;

- Distribuibilità di alcuni strati;
- Sviluppo degli strati da parte di **team di sviluppo differenti**.

### **Strato: la logica applicativa**

La **logica applicativa** è lo **strato** che riguarda gli **oggetti di dominio**. È tuttavia utile suddividere la logica applicativa in due strati:

- **Strato del dominio**: gli oggetti di dominio;
- **Strato application**: gli oggetti che gestiscono il workflow da e per gli oggetti di dominio, soprattutto dall'UI (strato opzionale ma molto consigliato).

L'approccio consigliato nella creazione di oggetti software per lo strato del dominio consiste nel creare oggetti con nomi e informazioni simili a quelli del Modello di Dominio. In questo modo si ha un **basso salto rappresentazionale** e **facilità di comprensione**.



### **Principio di separazione Modello-Vista**

Gli oggetti non UI non devono essere connessi o accoppiati direttamente a oggetti non UI: **non si deve mettere logica applicativa in un oggetto dell'interfaccia utente**.

## **Progettazione**

### **Progettare a oggetti**

Gli sviluppatori possono approcciarsi in modi diversi alla progettazione:

- **Codifica**: progettare mentre si codifica, con TTD e refactoring possibilmente.
- **Disegno, poi codifica**: da diagrammi UML a codice.
- **Solo disegno**: uno strumento che converte disegno in codice (in qualche modo).

In un'ottica agile va ridotto il costo aggiuntivo del disegno e **modellare** per **comprendere** e **comunicare**, anziché documentare. Si modella **in gruppi**, portando avanti più documenti in parallelo (Diagrammi di Interazione e Diagramma delle Classi Software in primis).

Oltre all'abbozzo tramite disegni a parete possiamo considerare l'utilizzo di strumenti CASE per UML (gratuiti o a pagamento), che possono addirittura rivelarsi complementari. Sono da preferire strumenti CASE che possano integrarsi agli IDE più conosciuti e possibilmente con la funzione di reverse-engineering (da fare all'inizio di ogni iterazione con il codice dell'iterazione precedente). In ogni caso al **disegno UML** è bene **dedicarci al più alcune ore, massimo un giorno** se l'iterazione è di circa 3 settimane.

### **Modelli statici e dinamici**

Ci sono due tipi di modelli per gli oggetti:

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

### (Extra) Schede CRC

Le **schede CRC** (Class, Responsibility, Collaboration) sono piccoli fogli di cartoncino che riportano, scritte, le **responsabilità** e i **collaboratori** di una classe (una scheda corrisponde a una classe).

## [D] Diagrammi di Interazione

I **Diagrammi di Interazione** illustrano il **modo** in cui gli oggetti **interagiscono** attraverso lo **scambio di messaggi**. Un'interazione è una **specificazione** di come alcuni oggetti si **scambiano messaggi** nel tempo per eseguire un compito nell'ambito di un certo contesto. Un'interazione è **motivata dalla necessità** di eseguire un determinato compito, dato da un messaggio (detto messaggio trovato) che dà il via all'interazione tra un oggetto e altri partecipanti.

UML offre per modellarli i **Diagrammi di Sequenza**, che sono utilizzati per la modellazione dinamica degli oggetti (anche oggetti di dominio: li abbiamo già visti negli SSD).

Ci sono 4 tipi di Diagrammi di Interazione:

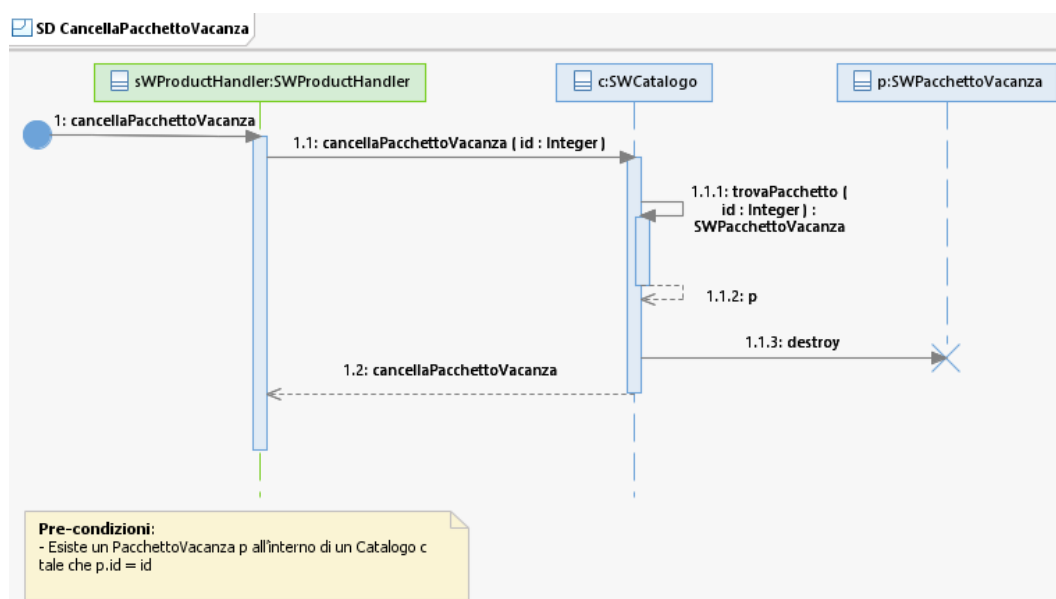
- Diagrammi di Sequenza (SD);
- Diagrammi di Comunicazione (CD);
- Diagrammi di Interazione generale;
- Diagrammi di temporizzazione.

I **Diagrammi di Sequenza** mostrano le interazioni tra **linee di vita verticali**; i **Diagrammi di Comunicazione** mostrano le iterazioni tra gli oggetti in un **formato a grafo o rete**. Tra i due non esiste una scelta corretta in senso assoluto, sebbene la specifica, e le notazioni, di UML siano maggiormente incentrate sui Diagrammi di Sequenza.

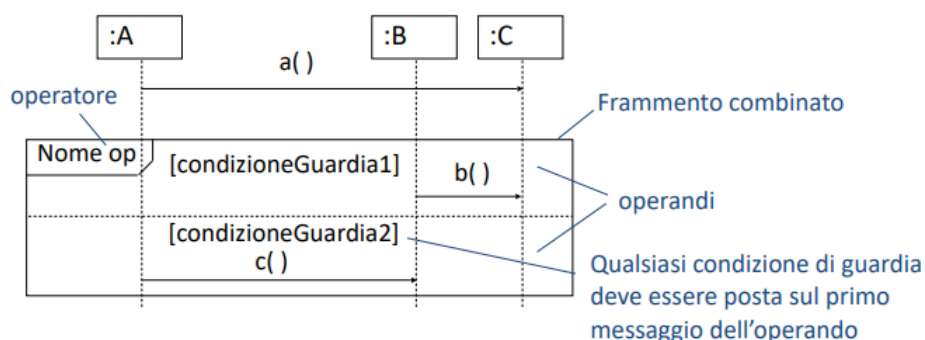
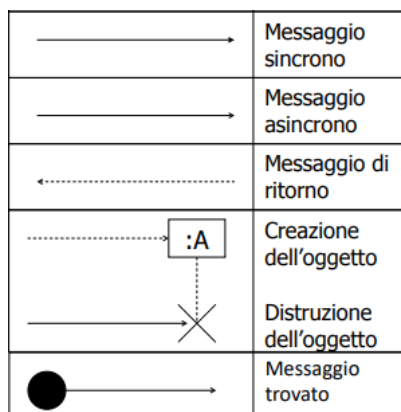
La scelta tra i due può essere dettata dalla necessità di ottimizzare l'utilizzo dello spazio di disegno (i Diagrammi di Comunicazione sfruttano meglio lo spazio sia verticale che orizzontale, mentre quelli di sequenza si sviluppano obbligatoriamente in orizzontale per i nuovi oggetti).

### [D] Diagrammi di Sequenza (SD)

Non ci dilunghiamo con i **Diagrammi di Sequenza** UML, che dovremmo aver già noti dagli SSD; qui recuperiamo le differenze dato che non siamo più a livello di dominio ma a **livello software**. Questo è un esempio di un SD preso dal progetto "Travel On": si noti il messaggio di destroy, che distrugge l'oggetto p, che era stato ritornato come risultato dell'operazione "trovaPacchetto" (lo sappiamo grazie al messaggio di ritorno p).



Più in generale, per le frecce e per i frammenti combinati o frames (nota: la punta della freccia dei messaggi sincroni è piena →, mentre quella dei messaggi asincroni è vuota →):



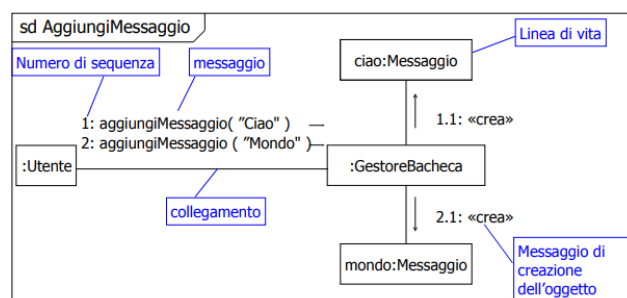
Operatore	Nome	Significato
opt	Option	C'è un singolo operando che viene eseguito se la condizione è vera (if...then)
alt	Alternatives	Viene eseguito l'operando la cui condizione è vera. Si può usare la parola chiave "altrimenti" o "else" per il caso di default (switch...case)
loop	Loop	Ha una sintassi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

Inoltre ricordiamo che a differenza degli SSD:

- Nelle **note** si possono mettere i **vincoli** o **pezzi di codice utili**;
- Le **linee di vita** negli SD rappresentano spesso degli **oggetti**, ma si possono specificare anche **collezioni** di un certo tipo di oggetti; se specificano **interfacce**, si deve proseguire a creare un "**sotto-SD**" per ogni classe che implementa l'interfaccia;
- Ci sono le **autodeleghe**: non ragioniamo più a scatola nera;
- Le **invarianti di stato**: sono rettangoli con i bordi rotondi, posti sulla linea di vita di un oggetto, che ne indicano lo stato (potrebbe essere "pagato" tanto quanto "nonPagato" per un Ordine);
- C'è la notazione "1" per gli oggetti Singleton nel rettangolo in cima alla lifeline.

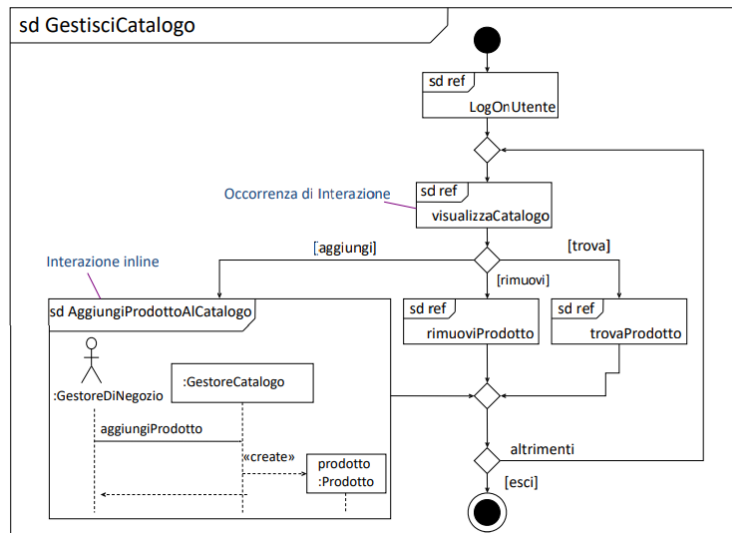
### [D] Diagrammi di Comunicazione (CD)

I **Diagrammi di Comunicazione** sono identici ai Diagrammi di Sequenza, se non per l'organizzazione spaziale. Le **linee di vita** sono **formate** nel momento in cui c'è un messaggio, che porta la firma del metodo; questi sono **numerati** (tranne il messaggio trovato iniziale se si vuole semplificare) per poterli leggere nel giusto ordine. Le frecce indicano la **direzione di invio** del messaggio. I nomi dei frame si indicano prima della signature.



### (Extra) [D] Diagrammi di Interazione Generale

I **Diagrammi di Interazione Generale** modellano il flusso di controllo tra interazioni ad alto livello; mostrano le interazioni e le occorrenze di interazioni e hanno la sintassi dei diagrammi di attività.



## [D] Diagramma delle Classi Software

I **Diagrammi delle Classi** illustrano le **classi**, le **interfacce** e le **relative associazioni**; sono usati per la modellazione statica delle classi e sono stati già introdotti per la realizzazione del Modello di Dominio (dove le classi erano concettuali e non software)

### Definizioni varie (nuove o rilevanti)

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

- Un **identificativo univoco** – la parte di **identità**;
- I **valori degli attributi** – la parte dei dati (detto **stato**);
- Le **operazioni** – la parte **comportamento**.

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

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

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

Abbiamo alcuni **meccanismi di estendibilità** per ampliare gli elementi di modelli a nostra disposizione:

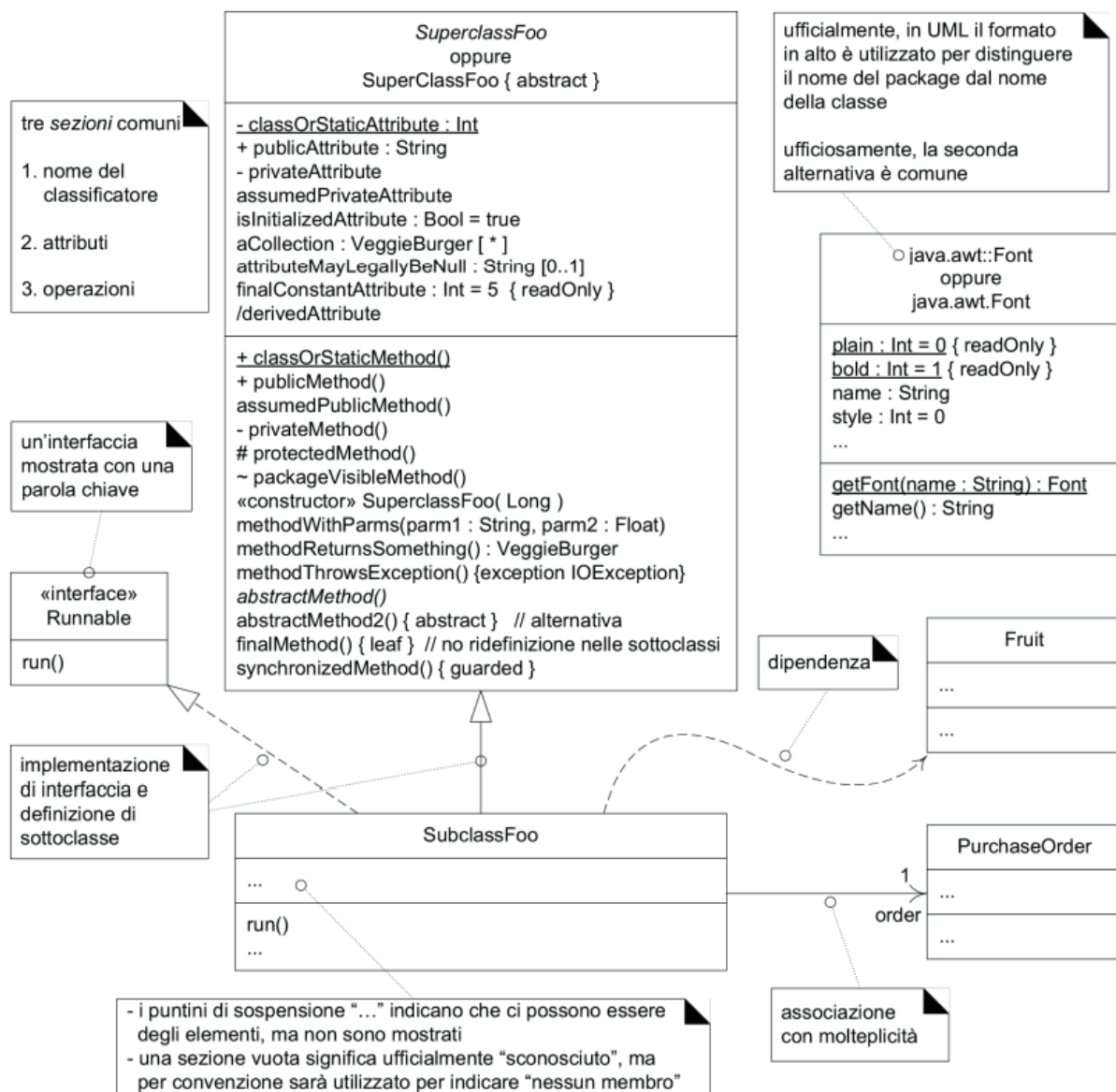
- **Vincoli**: Estendono la **semantica di un elemento** consentendo di aggiungere nuove regole;
- **Stereotipi**: Definiscono un **nuovo elemento di modellazione UML** basandosi su un esistente: è possibile definire la semantica degli stereotipi, che aggiungono nuovi elementi al meta-modello (modello che definisce modelli, come un meta-film è un film che fa vedere come si fanno i film) UML;
- **Tag**: Permettono di estendere la **definizione di un elemento** tramite l'aggiunta di **nuove informazioni specifiche**.

Un **profilo** è un insieme di stereotipi, tag e vincoli che si usa per **personalizzare UML**.

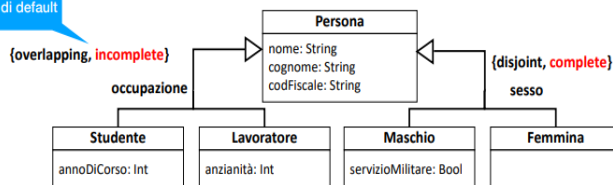


## Ripasso generale UML

Si assume che voi abbiate passato **Programmazione 2** e che sappiate riconoscere un diagramma UML con i vari componenti e leggerlo senza problemi. Tuttavia la memoria è corta, quindi in questa pagina ci sono immagini utili per un ripasso rapido. Va ricordato che nei Diagrammi delle Classi le linee delle dipendenze vanno usate per descrivere le dipendenze tra oggetti, per la visibilità globale, per variabili parametro, per variabili locali e per metodi statici.

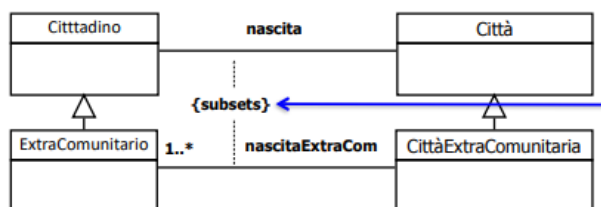


Valore di default



**Overlapping/Disjoint:** i classificatori possono/non possono avere istanze in comune.

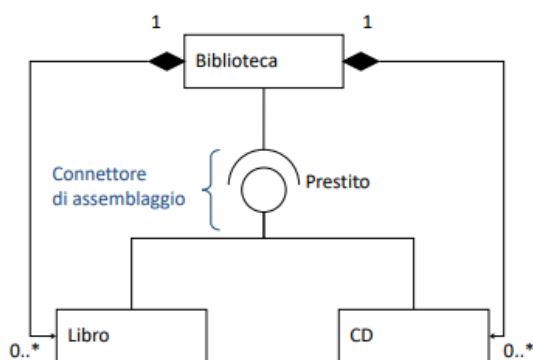
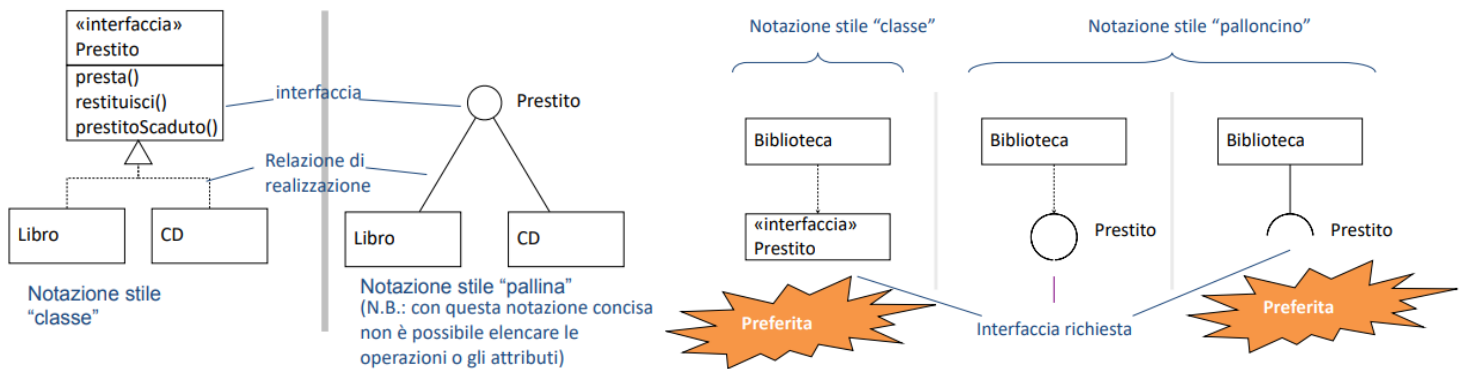
**Complete/incomplete:** l'unione delle istanze delle sottoclassi è/non è uguale all'insieme delle istanze della superclasse.



Ci sono vari tipi di specializzazione, tra cui di associazione (qui a fianco), di attributi (si limitano i valori dello specializzato), di operazioni (overriding).

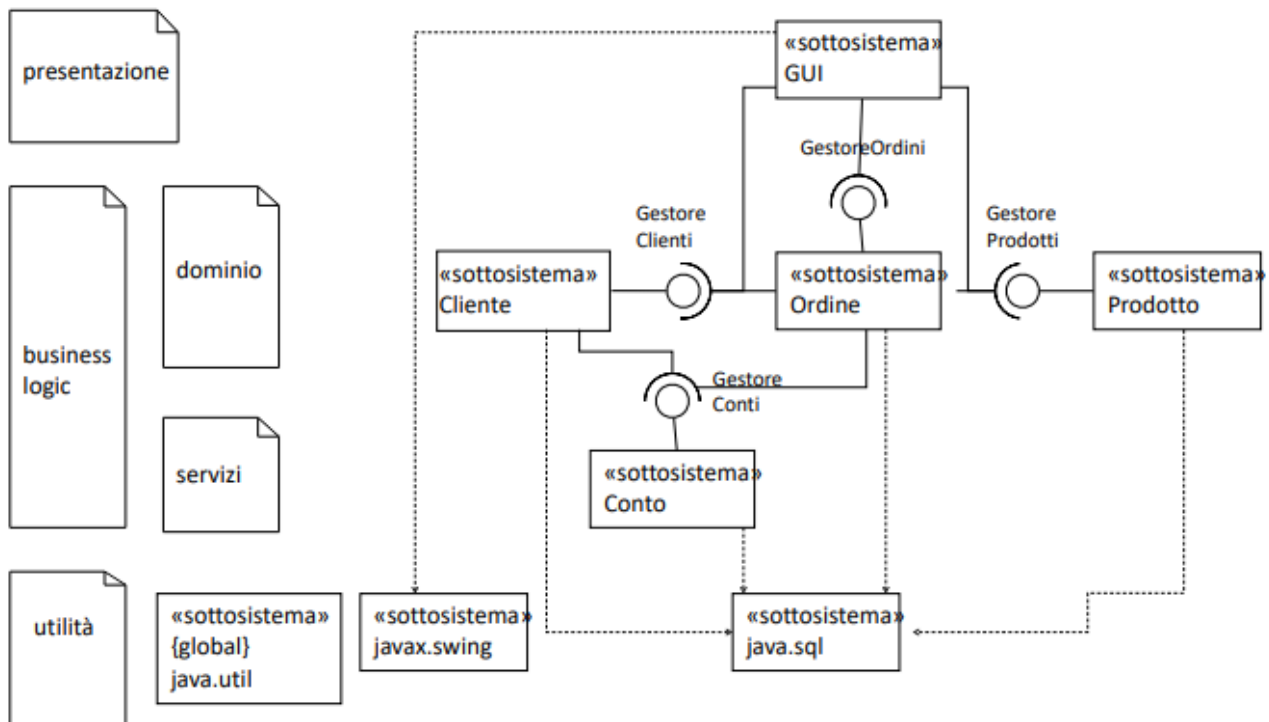
## Interfacce

Un'interfaccia è un insieme di **funzionalità pubbliche** identificate da un **nome**; separa le specifiche di una funzionalità dall'implementazione della stessa. Un'interfaccia definisce un **contratto** (non Contratto!) e tutti i classificatori che la realizzano lo devono rispettare. La novità per noi è la notazione "pallina" o "lollipop" o "palloncino" (o "<insert any round object>"). Di seguito vari esempi:



Biblioteca **richiede** (vuole che si usi) l'interfaccia prestito (**la coppa**); Libro e CD **forniscono** (implementano) l'interfaccia prestito (**pallino**).

Per fare un esempio naturale del tutto fuori luogo, la ragazza richiede, il ragazzo fornisce. Sì, in quel senso. I simboli (coppa e pallino) sono adeguati alla dotazione di entrambi, se si nota.





## [D] Macchine a Stati

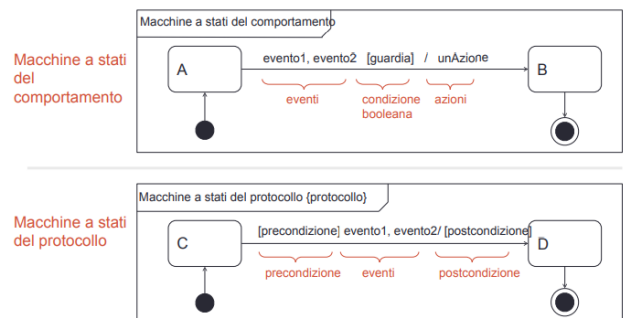
Le **Macchine a Stati** sono dei diagrammi che possono essere utilizzati per modellare il **comportamento dinamico** di classificatori quali classi, casi d'uso, sottosistemi e interi sistemi. Un Diagramma di Macchina a Stati essenzialmente mostra il **ciclo di vita** di un oggetto, modellato attraverso una successione di **stati** (condizione di un oggetto in un lasso di tempo), **transizioni** (relazioni tra stati etichettate; quando si verifica l'evento l'oggetto cambia stato) ed **eventi**. Possono anche rappresentare protocolli. UP non li indica come elaborati, vanno realizzati solo se sono utili.

Gli **oggetti** il cui comportamento è modellato possono essere di due tipi:

- **Indipendenti dallo stato**: risponde sempre in uno stesso modo ad un determinato evento;
- **Dipendenti dallo stato**: risponde in modo diverso in base allo stato in cui si trova.

### Rappresentazione di una Macchina a Stati

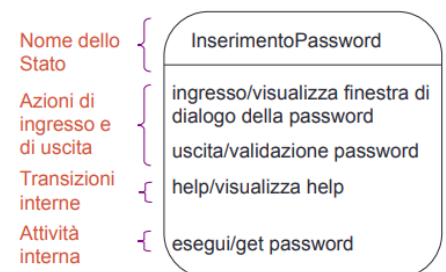
Un Diagramma di Macchina a Stati di UML illustra i possibili **stati** per un oggetto (ossia le **combinazioni dei valori degli attributi**, delle sue **relazioni** e delle **attività in corso**) attraverso dei **rettangoli arrotondati**, insieme al **comportamento** dell'oggetto in termini di **transizioni** – **freccie etichettate** con delle azioni istantanee non interrompibili – fra gli stati in risposta agli eventi durante la sua vita. Un ciclo comincia di solito dallo **stato iniziale** (pallino pieno) e finisce quando arriva nello **stato finale** (pallino pieno con bordo).



Si noti come nelle Macchine a Stati del comportamento ci sia una guardia dopo gli eventi, mentre nelle Macchine a Stati del protocollo si mettono una pre-condizione e una post-condizione.

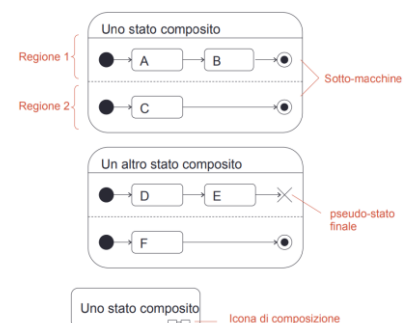
### Gli stati

Uno **stato** ha un **nome**, delle **azioni di ingresso e di uscita** (archi entranti e uscenti), può avere delle **transizioni interne** (come se fossero transizioni a cappio, non cambiano lo stato) e delle **attività interne**, che richiedono un tempo finito e sono interrompibili (sono ciò che viene fatto mentre l'oggetto è in quello stato). Si noti dall'immagine la sintassi per ogni tipo di azione/transizione/attività. Esistono anche stati più complessi: parliamo degli **stati composti** e degli **pseudostati**.



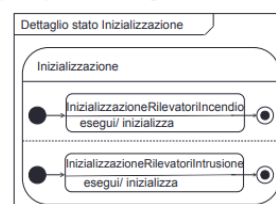
### Stati composti

Sono **stati** che includono una (stati **composti semplici**) o più **regioni** (stati **composti ortogonali**) che hanno all'interno delle **sotto-macchine**. Lo stato finale ferma solo la regione in cui si trova; lo stato finale di stato composto (vedere gli pseudostati) termina l'intero stato composto. Si possono indicare "collapsati" mettendo l'icona di stato composto nel rettangolo.

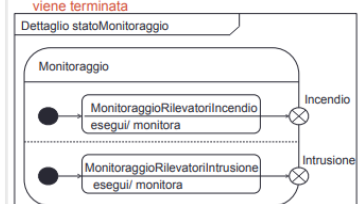


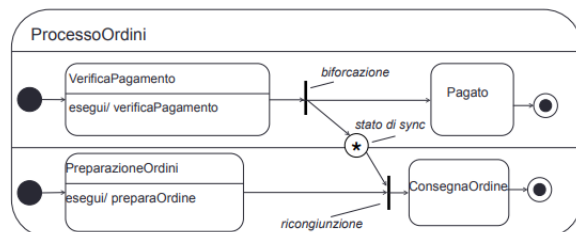
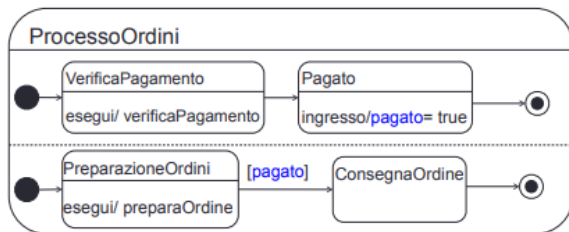
Per le **Macchine a Stati ortogonali**, se le regioni hanno tutte uno stato finale la terminazione è sincrona; se invece hanno pseudostati di uscita, lo stato composto termina quando una delle regioni entra nel suo pseudostato di uscita. Le varie regioni possono **comunicare** tra loro attraverso dei **flag** o stati di **sync**.

Uscita sincronizzata – si esce dallo stato composto quando tutte le regioni sono terminate



Uscita asincrona – si esce dallo stato composto quando una regione termina. L'altra sotto-macchina viene terminata

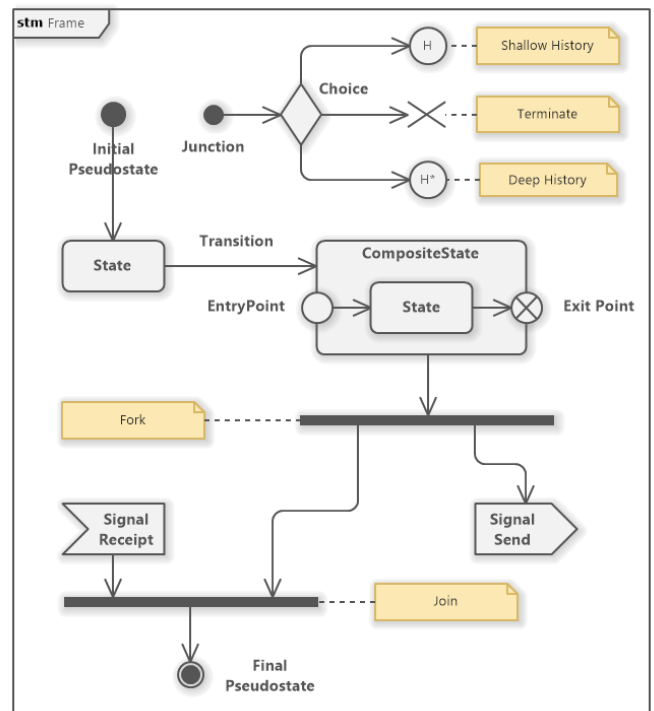




## Pseudostati

Si possono avere diversi **pseudostati** (stati che in realtà un oggetto non avrà mai):

- **Giunzione:** ricongiunge più archi diretti verso lo stesso stato, si indica con un piccolo pallino pieno;
- **Selezione:** è uno stato con un solo input e vari output mutuamente esclusivi con guardia, nel momento in cui il flusso arriva viene preso il ramo con la condizione vera. Si indica con un quadrato ruotato di 45° (detto diamante,  $\diamond$ );
- **Di ingresso:** pallino vuoto sul bordo di uno stato composito che indica l'entrata nello stesso;
- **Di uscita:** pallino vuoto con una x che funge da punto di uscita per uno stato composito;
- **Con memoria semplice:** stato che ricorda l'ultimo sotto-stato quando si è lasciato lo stato composito, si indica con un pallino vuoto con all'interno una H;
- **Con memoria multilivello;** come lo stato di memoria semplice, ma può ricordarsi i sotto-sotto-stati di eventuali sotto-stati dello stato composito (quindi memorizza gli ultimi stati di tutti gli stati composti presenti nello stato composito in cui è collocato, quest'ultimo compreso); la H diventa H\*;
- **Finale per gli stati composti:** lo stato finale di un intero stato composito è indicato con una croce.

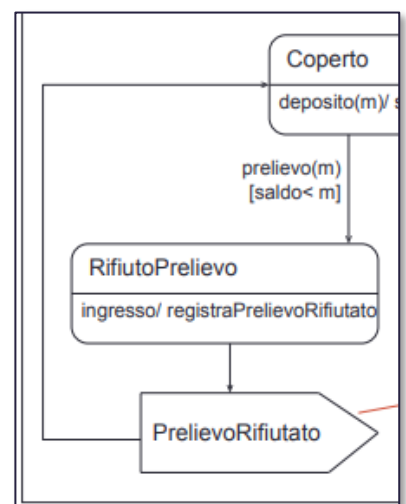


Inoltre possiamo avere delle linee di **fork** e **join**, che ci permettono di **suddividere** e **ricongiungere** (sincronizzandole) linee di flusso diverse (anche se non le vediamo qui, tanto quanto nei Diagrammi di Attività).

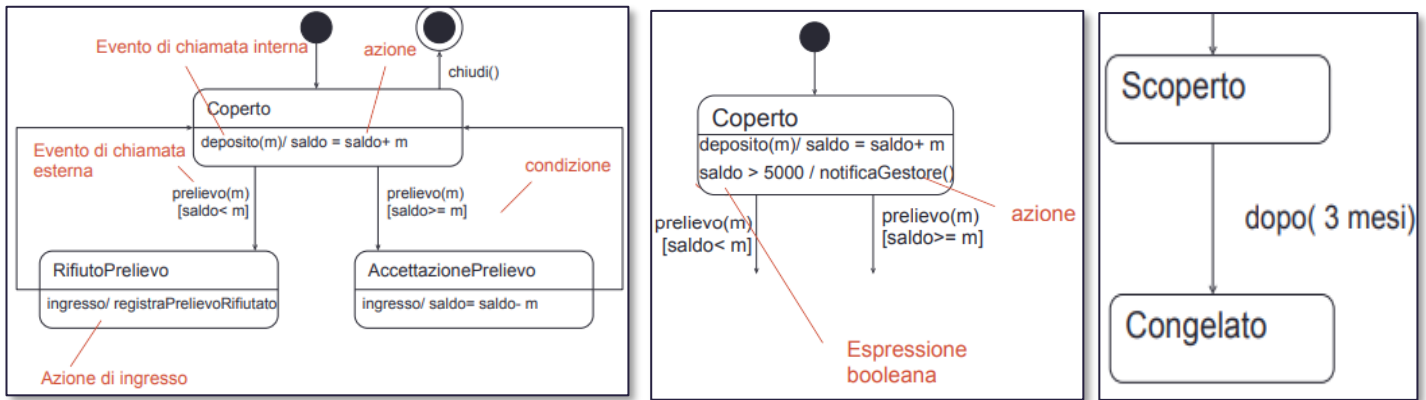
## Eventi

Abbiamo vari tipi di **eventi** a disposizione nelle Macchine a Stati:

- **Eventi di Chiamata:** sono chiamate per specifiche operazioni o una sequenza di operazioni. Possono essere interni o esterni, con o senza guardia;
- **Eventi di Segnale:** comporta l'invio o l'attesa della ricezione (entrambi asincroni) di un segnale, viene indicato con una bandierina. Può contenere gli attributi del pacchetto di informazioni da spedire;
- **Evento di variazione:** nel momento in cui una condizione viene verificata all'interno di uno stato, viene eseguita l'azione indicata dopo lo slash; non cambia di stato;
- **Evento temporale:** verificano condizioni temporali, dopo( ) o quando( ).



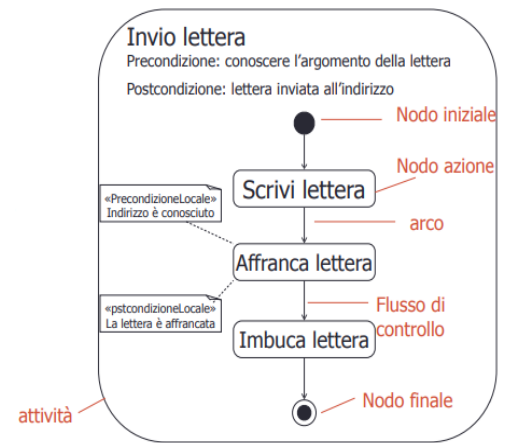
(immagini: a fianco, evento di segnale; sotto, evento di chiamata, di variazione, temporale)



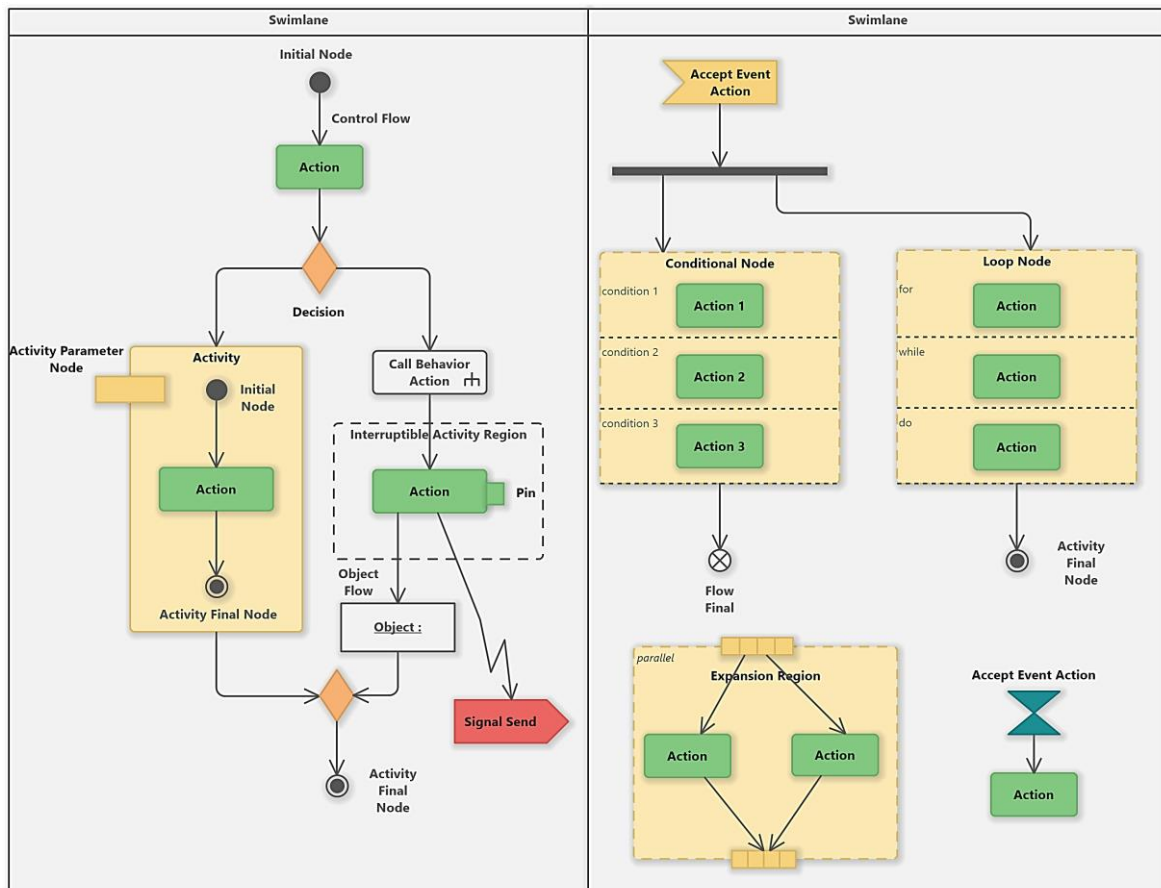
## [D] Diagramma delle Attività

Un **Diagramma di Attività** mostra le attività (reti di nodi connessi da archi) di un processo; si usano per la **visualizzazione dei flussi**, non per niente vengono chiamati anche "Diagrammi di Flusso Object Oriented". Si possono anche usare per descrivere flussi di dati o oggetti, in tal caso prende il nome di Diagramma dei flussi di dati (DFD, Data Flow Diagram); viene usata la stessa notazione dei Diagrammi di Attività.

In UP i Diagrammi di Attività non sono obbligatori, servono esclusivamente per **visualizzare il workflow** di un processo, di un oggetto o di dati nel caso questo sia complicato. Viene usata la stessa notazione UML usata per le Macchine a Stati, se non per il fatto che gli stati diventano dei nodi che vengono suddivisi in varie partizioni e altre differenze minori.

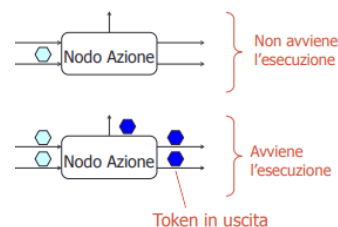


Questa infografica mostra i principali componenti di un Diagramma di Attività.




## Token game

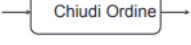
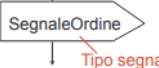
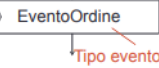

Un **token** è un **oggetto**, un **insieme di dati** o un **flusso di controllo**. Viene **passato da nodo a nodo** attraverso le transizioni per rappresentare il flusso. Un nodo azione rilascia dei token su tutti gli archi in uscita quando tutti gli archi in entrata hanno un token; un nodo di controllo gestisce i token in entrata in base alla sua funzione; un nodo oggetto funziona come pila o coda di token oggetto e ne rilascia uno alla volta in base alla politica indicata.



## Nodo azione

I **nodi azione** rappresentano **unità discrete di lavoro atomiche** all'interno dell'attività; quando c'è un token per ogni arco in entrata viene eseguita l'azione del nodo e mandato un token in ogni nodo d'uscita.




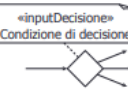



Un nodo azione può **invocare un'attività** (nodo rastrello ) , un **comportamento** o un'**operazione**. Eventualmente contiene pezzi di codice dove "self" indica le caratteristiche del contesto dell'attività.

Sintassi	Semantica
	Nodo azione di chiamata – invoca un'attività, un comportamento o un'operazione. Tipo di nodo azione più comune
 Tipo segnale	Invia segnale di azione – invia un segnale in modo asincrono. Può accettare i parametri di input necessari per creare il segnale
 Tipo evento	Accetta un evento – aspetta gli eventi individuati dal suo oggetto proprietario ed emette l'evento sull'arco in uscita. E' attivo quando riceve un token nel suo arco in entrata. Se non c'è alcun arco entrante, inizia quando la sua attività contenente inizia
 Attendi 30 min espressione temporale	Accetta un evento temporale: risponde al tempo. Genera eventi temporali secondo la sua espressione temporale

## Nodo controllo

I **nodi controllo** controllano il **flusso attraverso l'attività**. La gestione dei token varia in base allo specifico nodo.

(Tabella a fianco)

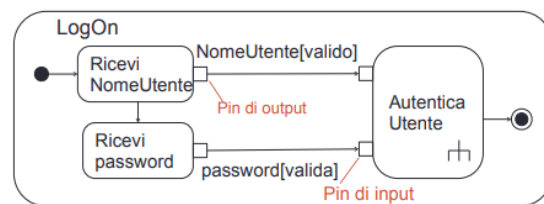
Sintassi	Semantica	
	Nodo iniziale – indica dove inizia il flusso quando invocata un'attività	
	Nodo finale dell'attività – termina un'attività	Nodi finali
	Nodo finale del flusso – termina un flusso specifico all'interno di un'attività. Gli altri flussi non vengono influenzati	
 «inputDecisione» Condizione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

## Nodo Oggetto

I **nodi oggetto** rappresentano **pile/code di token oggetti** usati nell'attività. Si può indicare un limite massimo di token, la politica, eventuali collezioni, criteri di selezione o stato dell'oggetto.

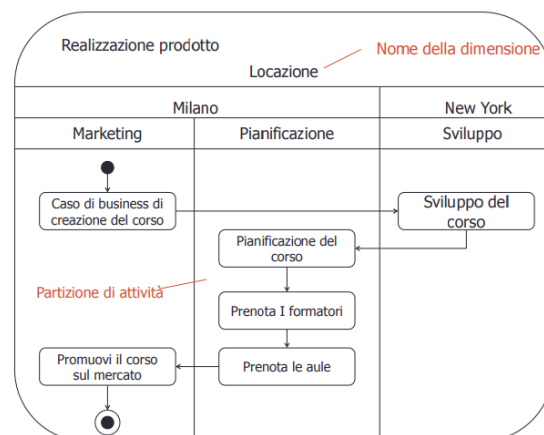
## Pin

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



## Partizioni

Una **partizione** è un **raggruppamento** ad alto livello di **azioni correlate**; possono essere **gerarchiche**, verticali o orizzontali, e riferirsi a diverse entità. Se non possono essere mostrate normalmente si può fare con del testo (si indicano le partizioni a cui appartiene il nodo prima del suo nome).



## Progettazione guidata dalla responsabilità (RDD)

È importante pensare in termini di **responsabilità**, **ruoli** e **collaborazioni** quando si progetta a oggetti; UP incoraggia fortemente questa pratica. Una **responsabilità** è un'astrazione di ciò che un oggetto **fa** o **rappresenta**; la **progettazione guidata dalle responsabilità** (RDD, *Responsibility-Driven Design*) considera un progetto OO come una **comunità di oggetti**, con **responsabilità**, che **collabora** per fornire delle funzionalità.

Le responsabilità sono assegnate alle classi durante la **progettazione di oggetti** (anche se si deducono già dal Modello di Dominio) e sono di due tipi:

- **Di fare** (es. eseguire un calcolo o coordinare attività di altri oggetti);
- **Di conoscere** (es. conoscere i propri dati privati o gli oggetti correlati).

Una **responsabilità** non è la stessa cosa di un metodo: anzi i **metodi** sono **implementati per adempiere alle responsabilità**. I metodi agiscono da soli o collaborano con altri metodi e oggetti per realizzare le responsabilità.

La RDD viene fatta iterativamente come segue:

- **Identificare le responsabilità**, una alla volta;
- Chiedersi **a quale oggetto** software **assegnarla**;
- Chiedersi **come** fa l'oggetto a **soddisfarla**.

Per l'**assegnazione delle responsabilità** è utile fare riferimento ai **principi** definiti dai **Pattern GRASP**.

## Pattern GRASP

I **Pattern GRASP** (General Responsibility Assignment Software Patterns) descrivono dei **principi di base** per la progettazione di oggetti e l'**assegnazione di responsabilità**. Nell'ambito di UML si può iniziare a pensare alle responsabilità e ad applicare i principi durante il disegno dei Diagrammi di Interazione.

Un **pattern** è una **coppia problema/soluzione** ben conosciuta e con un **nome** (per facilitare comprensione, memorizzazione e comunicazione). Può essere applicato in nuovi contesti, contiene consigli su come applicarlo e discute compromessi, implementazioni, variazioni etc. I pattern si basano su **soluzioni e principi già applicati e dimostratisi corretti** in diverse situazioni, si fondano quindi sull'esperienza e non sulla sperimentazione.

Di seguito l'elenco dei **9 Pattern GRASP**:

- *Creator*;
- *Information Expert*;
- *Low Coupling*;
- *Controller*;
- *High Cohesion*;
- *Pure Fabrication*;
- *Polymorphism*;
- *Indirection*;
- *Protected Variations*.



**Tabella dei Pattern GRASP**

	<b>Problema</b>	<b>Soluzione</b>	<b>Note</b>
<b>Creator</b>	Chi <b>crea</b> un oggetto A?	Assegna alla classe B la <b>responsabilità di creare</b> un'istanza della classe A se: - B <b>contiene o aggrega</b> con una composizione oggetti di tipo A; - B <b>registra</b> A; - B <b>utilizza strettamente</b> A; - B possiede i dati per l' <b>inizializzazione</b> di A.	Se la creazione è complessa, usare altre soluzioni come <b>Abstract Factory</b> (pattern GoF).
<b>Information Expert</b>	Qual è un <b>principio di base</b> per <b>assegnare responsabilità</b> agli oggetti?	Assegna una <b>responsabilità</b> alla classe che possiede le <b>informazioni necessarie</b> per soddisfarla.	Spesso è necessario far collaborare molti <b>esperti parziali</b> delle informazioni. Un'analogia con il mondo reale: normalmente si assegnano le responsabilità alle persone che posseggono le informazioni necessarie per svolgere un compito. In alcuni casi, le soluzioni suggerite da Information Expert hanno <b>problemi di accoppiamento e di coesione</b> , come il salvataggio in una base di dati: in questo caso occorre mettere la logica della base di dati in un altro luogo, separato dalla logica applicativa.
<b>Low Coupling</b>	Come <b>ridurre l'impatto</b> dei cambiamenti?	Assegna le <b>responsabilità</b> in modo tale che l' <b>accoppiamento</b> (l'accoppiamento è la misura di quanto un elemento è collegato, conosce o si basa su altri elementi) non necessario rimanga <b>basso</b> . Usa questo principio per valutare le alternative.	Le forme di accoppiamento da un tipo X a un tipo Y comprendono: - la classe X <b>ha un attributo</b> di tipo Y o <b>referenzia una collezione</b> di oggetti di tipo Y; - un oggetto di tipo X <b>richiama operazioni</b> di un oggetto di tipo Y; - un oggetto di tipo X <b>crea</b> un oggetto di tipo Y; - il tipo X ha un metodo che <b>contiene</b> un <b>parametro/una variabile locale/tipo di ritorno</b> di tipo Y; - X è una <b>sottoclasse</b> di Y; - Y è un' <b>interfaccia</b> e X la implementa. Ha come pattern correlato <b>Protected Variations</b> .

	Problema	Soluzione	Note
Controller	Qual è il primo oggetto oltre lo <b>strato UI</b> a ricevere e <b>coordinare</b> ("controllare") un' <b>operazione di sistema</b> ?	Assegna la <b>responsabilità</b> a un oggetto che rappresenta una delle seguenti scelte: - Rappresenta il "sistema" complessivo, un " <b>oggetto radice</b> ", un dispositivo all'interno del quale viene eseguito il software, un <b>punto d'accesso al software</b> o a un sottosistema principale ( <b>Façade Controller</b> ); - Rappresenta uno <b>scenario di un caso d'uso</b> all'interno del quale si verifica l'operazione di sistema ( <b>Use Case Controller</b> , si tratta di una <b>Pure Fabrication</b> ).	Un controller è il primo oggetto oltre lo strato UI che è responsabile di <b>ricevere e gestire</b> un messaggio di un'operazione di sistema. È semplicemente un <b>pattern di delega, da UI a dominio</b> , e un controller deve limitarsi a coordinare le attività. <b>Errori comuni:</b> - dare <b>troppe responsabilità</b> a un controller (Controller <b>gonfi</b> , cioè controller con coesione bassa); - <b>far svolgere</b> parte del <b>lavoro</b> al controller prima di delegarlo; - il controller <b>conserva informazioni</b> sul sistema e sul dominio. <b>Soluzioni:</b> - <b>delegare la gestione</b> delle operazioni di sistema; - <b>aggiungere altri controller</b> .
High Cohesion	Come <b>mantenere</b> gli <b>oggetti focalizzati</b> , <b>comprensibili</b> e <b>gestibili</b> e, come effetto collaterale, sostenere Low Coupling?	Assegna le <b>responsabilità</b> in modo tale che la <b>coesione</b> rimanga <b>alta</b> . Usa questo principio per valutare le alternative.	La <b>coesione funzionale</b> è una misura di quanto fortemente siano <b>correlate le responsabilità</b> di un elemento. <b>Livelli di coesione:</b> - <b>Coesione molto bassa</b> : una classe responsabile in aree funzionali molto diverse; - <b>Coesione bassa</b> : una classe responsabile di un compito complesso in una sola area funzionale (giustificabile per motivi di efficienza in caso di pochi oggetti server distribuiti che forniscono servizi ampi e poco coesi); - <b>Coesione moderata</b> : una classe ha responsabilità leggere in poche aree diverse, logicamente correlate al concetto rappresentato dalla classe ma non l'una con l'altra; - <b>Coesione alta</b> : una classe ha responsabilità moderate in un' <b>unica area funzionale</b> e collabora con le altre (favorisce il riuso).

	<i>Problema</i>	<i>Soluzione</i>	<i>Note</i>
Pure Fabrication	Quale oggetto deve avere la <b>responsabilità</b> quando non si vogliono violare High Cohesion e Low Coupling, o altri obiettivi, ma le <b>soluzioni suggerite da Expert</b> (ad esempio) <b>non sono appropriate</b> ?	Assegnare un <b>insieme di responsabilità altamente coeso</b> a una <b>classe artificiale o di convenienza</b> , che non rappresenta un concetto del dominio del problema, ma piuttosto è una classe inventata per sostenere coesione alta, accoppiamento basso e riuso.	<p><b>Errori comuni:</b></p> <ul style="list-style-type: none"> <li>- <b>utilizzo eccessivo</b>, porta ad una <b>decomposizione del codice</b> (eccesso di oggetti comportamentali senza informazioni richieste per soddisfarli);</li> <li>- assegnare un insieme di responsabilità a una classe artificiale <b>senza però sostenere coesione alta, accoppiamento basso e riuso</b>;</li> <li>- ✨ <i>pensare che le Pure Fabrication siano "magiche" oppure sappiano progettarsi da sole</i> ✨.</li> </ul>
Polymorphism	Come gestire <b>alternative basate sul tipo</b> ? Come creare componenti software <b>inseribili</b> ?	Quando le alternative e comportamenti correlati variano con il tipo, allora assegna la <b>responsabilità del comportamento ai tipi</b> per i quali il comportamento varia, utilizzando <b>operazioni polimorfe</b> .	<p>Le estensioni sono <b>facili da aggiungere</b> ed è possibile introdurre nuove implementazioni <b>senza influire sui client</b>.</p> <p>Tuttavia <b>non</b> si devono <b>progettare</b> variazioni di fatto <b>improbabili</b> che in realtà non si verificheranno mai.</p>
Indirection	Dove assegnare una responsabilità, per <b>evitare l'accoppiamento diretto</b> tra più elementi? Come <b>disaccoppiare gli oggetti</b> in modo da sostenere un alto riuso e un accoppiamento basso?	Si assegna la responsabilità ad un <b>oggetto intermediario</b> che media tra altri componenti o servizi, in modo che non ci sia un accoppiamento diretto tra di essi.	<p>L'<b>intermediario</b> crea una <b>indirection</b> tra le componenti: si ha <b>accoppiamento più basso</b> tra componenti.</p> <p>Esempio di oggetti intermediari tra strato di dominio e servizi esterni: <b>Adapter</b> (vi è anche polimorfismo tra gli oggetti adattatori).</p>
Protected Variations	Come progettare oggetti, sottosistemi e sistemi in modo tale che le <b>variazioni</b> o l' <b>instabilità</b> di questi elementi <b>non si riversino su altri elementi</b> ?	Si identificano i <b>punti</b> in cui sono previste delle <b>variazioni</b> , poi si assegnano le <b>responsabilità per creare una "interfaccia" stabile</b> attorno a questi punti.	<p>Come per il Polymorphism, le estensioni sono <b>facili da aggiungere</b> ed è possibile introdurre nuove implementazioni senza influire sui client. Inoltre è possibile <b>ridurre l'impatto/il costo dei cambiamenti</b>.</p> <p>Non utilizzare Protected Variations per variazioni che non si verificheranno mai.</p>



## Design Pattern (GoF)

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

In questo corso vediamo **7** dei **pattern GoF** (Gang of Four – sì, erano in quattro ad averli ideati), che in totale raccolgono **23 design** classificati in base al loro **scopo** (creazionale, strutturale o comportamentale) e in base al loro **raggio d'azione** (classi o oggetti).

		Scopo		
		Creazionale	Strutturale	Comportamentale
Raggio d'azione	Classi	Factory Method	Adapter	Interpreter Template Method
	Oggetti	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

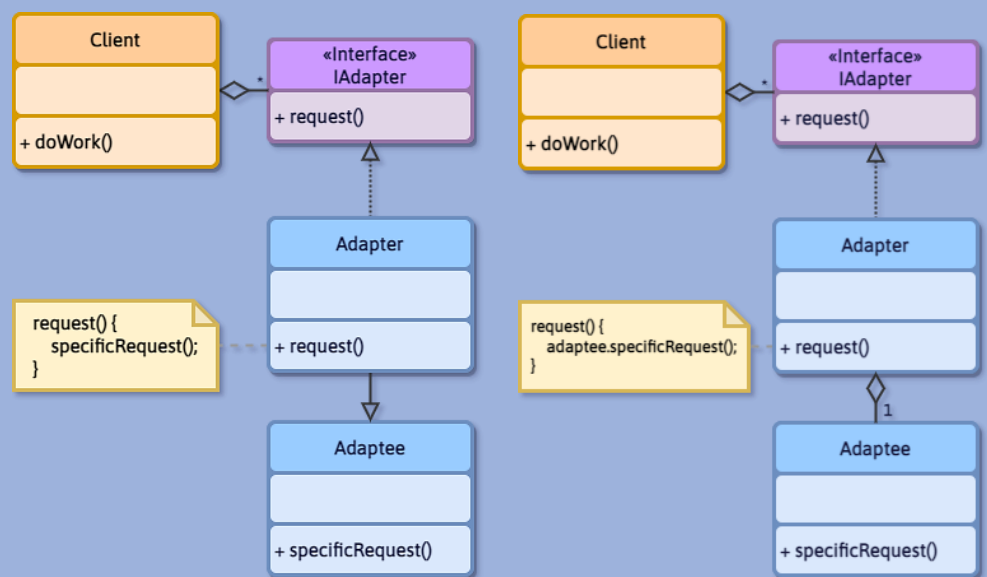
## Schemi dei Pattern GoF

### Adapter

Come gestire **interfacce incompatibili**, o fornire un'interfaccia stabile a **componenti simili** ma con interfacce diverse?

Converti l'interfaccia originale di un componente in un'altra interfaccia, attraverso un **oggetto adattatore intermedio**.

Ci sono due tipi di Adapter: l'Adapter Classe e l'Adapter Oggetto.

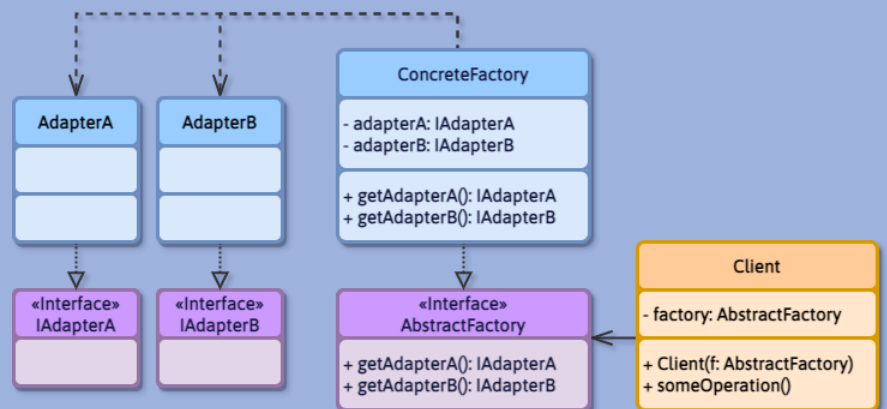


### Factory

Chi deve essere **responsabile della creazione** di oggetti quando ci sono delle considerazioni speciali (es. Adapter), come una **logica di creazione complessa**, quando si desidera separare le responsabilità di creazione per una coesione migliore, e così via?

Crea un oggetto **Pure Fabrication** chiamato **"Factory"** che gestisce la creazione.

La Factory non è un'interfaccia ma una classe, spesso Singleton (vedi prossimo Pattern); vi sono attributi di cui il tipo è un'interfaccia Adapter e metodi getter che restituiscono una qualsiasi implementazione. Qui l'esempio è con gli adapter e dunque ci serve una sola factory che mantiene le istanze degli adapter, ma possono esserci più ConcreteFactory che implementano l'AbstractFactory nel caso siano necessarie varianti degli adapter (un ipotetico AdapterA1 che implementa IAdapterA).



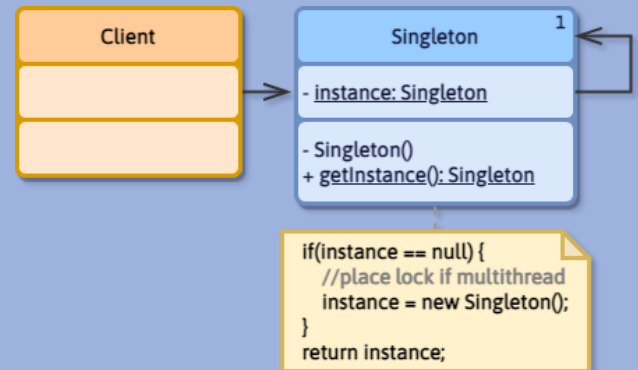
# Singleton

È **consentita** (o richiesta) esattamente **una sola istanza di una classe**, ovvero un "singleton". Gli altri oggetti hanno bisogno di un **punto di accesso globale** e singolo a questo oggetto.

Definisci un **metodo statico** (di classe) della classe che restituisce l'oggetto **singleton**.

In UML si può **scrivere 1 in alto a destra** per indicare una classe Singleton. Si riconosce una classe Singleton da un attributo privato statico singleton chiamato "instance" (o uniqueInstance), dal costruttore privato e dal metodo pubblico statico "getInstance" senza parametri. **È importante mettere synchronized sul metodo getInstance** per utilizzare un lock e quindi evitare di creare due istanze in caso di thread concorrenti.

**Non è efficace nel caso di applicazioni distribuite** poiché ciascuna VM potrebbe avere un'istanza diversa della classe Singleton. Viene spesso utilizzato per oggetti **Factory** e **Facade**.



**Implementazione non lazy:** si costruisce subito la Factory

```

public class ServicesFactory {
    private static ServicesFactory _instance
        = new ServicesFactory();

    private ServicesFactory() {
        //construct object . . .
    }

    public static ServicesFactory
    getInstance() {
        return _instance;
    }
    //...
}
  
```

**Implementazione lazy:** si costruisce solo se viene richiesta

```

public class ServicesFactory {
    private static ServicesFactory _instance;

    private ServicesFactory() {
        // construct object . . .
    }

    public static synchronized ServicesFactory
    getInstance() {
        if (_instance == null) {
            _instance = new ServicesFactory();
        }
        return _instance;
    }
    //...
}
  
```

## Strategy

Come progettare per gestire un **insieme di algoritmi o politiche variabili** ma **correlati**? Come progettare per consentire di modificare questi algoritmi o politiche?

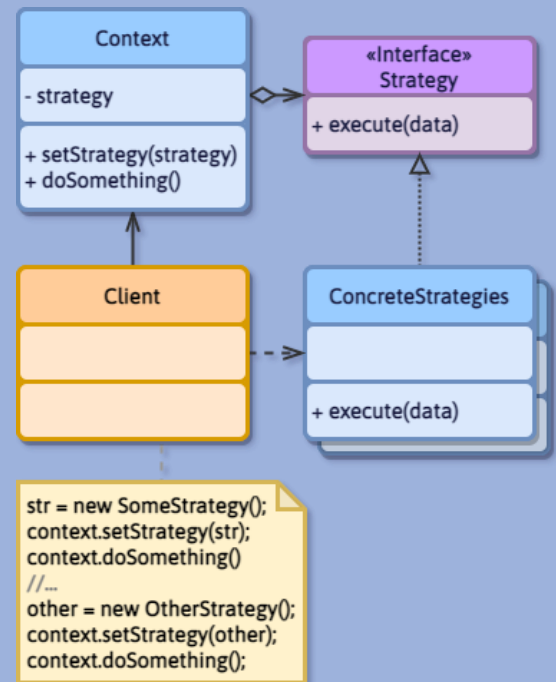
Definisci ciascun algoritmo/politica/strategia in una **classe separata**, con un'**interfaccia comune**.

Esempio: fornire una logica più complessa per la determinazione del prezzo scontato di una vendita.

Leggendo lo schema:

- Context mantiene un riferimento a una delle strategie concrete e comunica con questo oggetto solo mediante l'interfaccia Strategy;
- l'interfaccia Strategy è comune a tutte le strategie concrete;
- ConcreteStrategies indica le classi che implementano variazioni di un algoritmo;
- Context richiama il metodo sull'oggetto strategy associato ogni volta che ha bisogno di eseguire l'algoritmo. Non conosce il tipo dinamico della strategy o come viene eseguito l'algoritmo;
- il Client crea un oggetto specifico strategy e lo passa al Context. Quest'ultimo offre un setter che permette ai client di sostituire a runtime la strategy associata.

Strategy si basa su **Polymorphism** e **fornisce Protected Variations** rispetto agli algoritmi variabili. Sono spesso create da una **Factory**.



## Observer

Diversi tipi di oggetti subscriber (abbonato) sono **interessati ai cambiamenti di stato o agli eventi** di un oggetto publisher (editore). Ciascun subscriber vuole **reagire in un suo modo proprio** quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?

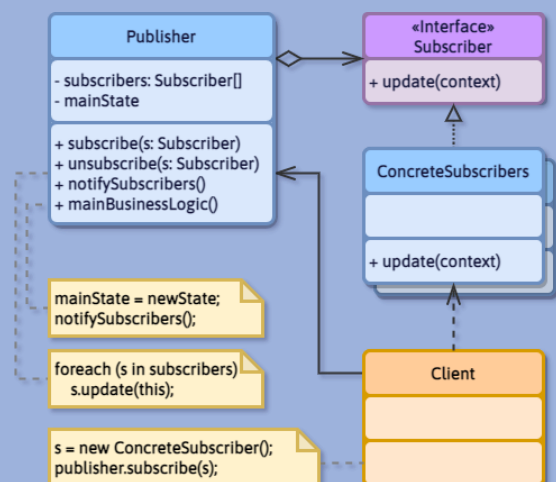
Definisci un'**interfaccia subscriber** o **listener** (ascoltatore). Gli oggetti subscriber implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

Fornisce un modo per accoppiare in modo debole gli oggetti che devono comunicare, in quanto:

- i publisher conoscono i subscriber solo mediante un'interfaccia;
- i subscriber possono registrarsi/cancellarsi dinamicamente con il publisher.

Qui è il client a creare e aggiornare Publisher e Subscriber.

Observer si basa su **Polymorphism** e fornisce **Protected Variations** in termini di protezione del publisher dalla conoscenza della classe specifica degli oggetti con cui comunica.



## Composite

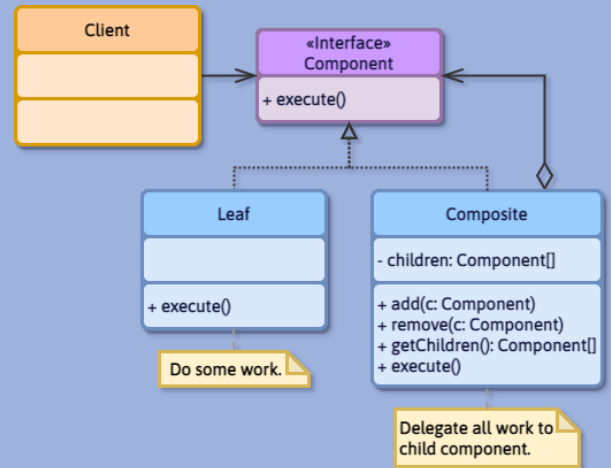
Come trattare un **gruppo o una struttura composta** di oggetti (polimorficamente) dello stesso tipo nello stesso modo di un oggetto non composto (atomico)?

Definisci le **classi per gli oggetti composti** e atomici in modo che implementino la **stessa interfaccia**.

Leggendo lo schema:

- l'interfaccia Component descrive operazioni comuni a entrambi gli elementi semplici e complessi dell'albero;
- una Leaf è un elemento base di un albero che non ha sotto-elementi. Di solito, fanno la maggior parte del lavoro reale (non possono delegarlo);
- il Container (Composite) è un elemento che ha sotto-elementi: foglie o altri container. Un container non conosce le classi concrete dei suoi figli. Comunica con i sotto-elementi solo mediante l'interfaccia Component. Quando riceve una richiesta, delega il lavoro ai suoi sotto-elementi, processa risultati intermedi e poi restituisce il risultato finale al Client;
- il Client comunica con tutti gli elementi tramite l'interfaccia Component.

**Composite** è spesso **utilizzato con Strategy**, si basa su **Polymorphism** e **fornisce Protected Variations** su un client.

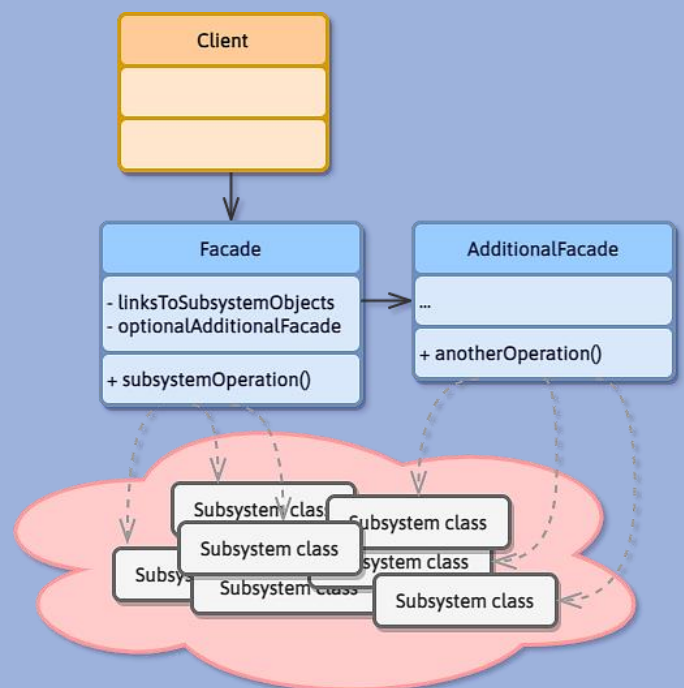


## Facade

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?

Definisci un **punto di contatto singolo** con il sottosistema, ovvero un oggetto **facade** (facciata) che copre il sottosistema. Questo oggetto facade presenta un'interfaccia singola e unificata ed è **responsabile della collaborazione** con i componenti del sottosistema.

Si tratta di un oggetto front-end che rappresenta il punto di entrata singolo ai servizi di un sottosistema. Normalmente si accede a un Facade attraverso il pattern Singleton. **Fornisce Protected Variations** dall'implementazione di un sottosistema, aggiungendo un **oggetto Indirection** che aiuta a **sostenere Low Coupling**. Un **facade controller** è un tipo **Facade**.



## Progettare la visibilità

La **visibilità** è la capacità di un oggetto di "vedere" o di **avere un riferimento** ad un altro oggetto. Se un oggetto mittente desidera inviare un messaggio a un oggetto destinatario, allora deve avere un qualche tipo di riferimento a quest'ultimo.

La visibilità può essere ottenuta dall'oggetto A all'oggetto B in quattro modi:

- Visibilità per **attributo** (B è un attributo di A);
- Visibilità per **parametro** (B è un parametro di un metodo di A);
- Visibilità **locale** (B è un oggetto locale di un metodo di A);
- Visibilità **globale** (B è in qualche modo visibile globalmente).

La visibilità per **attributo** e **globale** sono relativamente **permanenti**, mentre quella per **parametro** e **locale** sono relativamente **temporanee** poiché persistono solo nell'ambito del metodo.

## Dalla Progettazione all'Implementazione

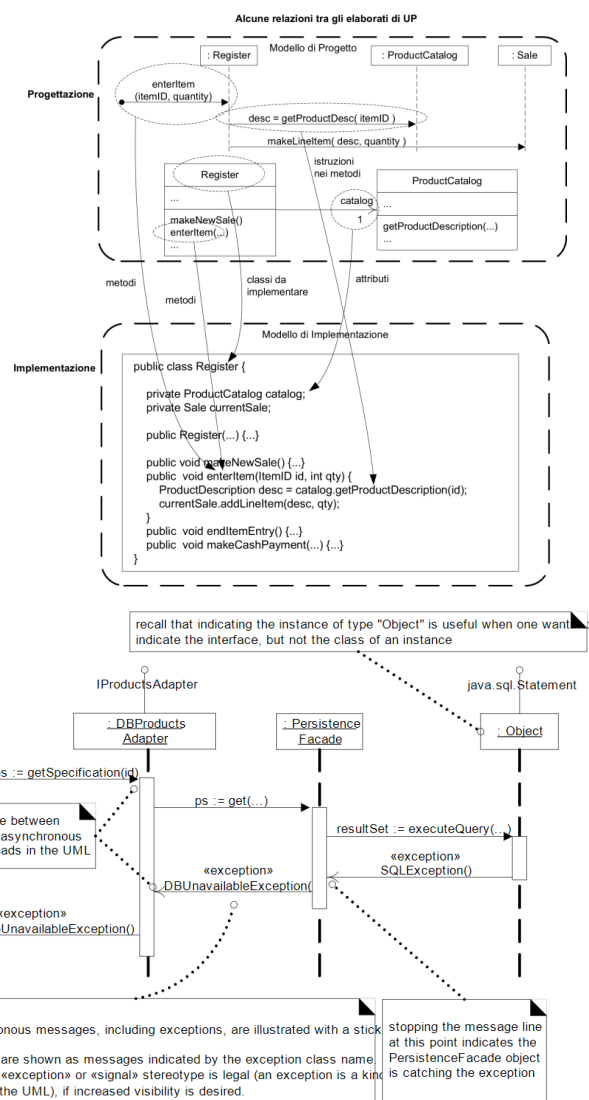
Gli elaborati creati durante la disciplina Progettazione sono utilizzati come input per il **processo di generazione del codice** (disciplina dell'Implementazione).

L'**Implementazione** in un linguaggio **Object Oriented** richiede la scrittura di codice per:

- Definire **classi** e **interfacce**;
- Dichiarare **variabili di istanza**;
- Definire **metodi** e **costruttori**.

Si **implementa** a partire dalla **classe meno accoppiata** fino a quella più accoppiata.

In UML le **eccezioni** possono essere indicate nelle **stringhe di proprietà dei messaggi** e delle **dichiarazioni delle operazioni**.



*Nota per l'orale: se un oggetto ha un attributo che implementa un'interfaccia, si dichiara la variabile d'istanza corrispondente in termini dell'interfaccia, non di una classe concreta:*

```
private List<SalesLineItem> lineItems = new ArrayList<>();
```



# Testing

*In verità non sappiamo a che disciplina appartengano TDD e Refactoring (non c'è sulle slide, né sul libro), siamo andati a intuito. RIP*

## Sviluppo guidato dai test (TDD)

Il **Test-Driven Development** (TDD), noto anche come **sviluppo preceduto dai test**, è una **best practice** applicabile a UP. L'idea di base consiste nello **scrivere prima i test**, immaginando che il codice da testare sia stato già scritto, e successivamente scrivere il codice per far passare il test.

Presenta i seguenti vantaggi:

- I test vengono **effettivamente scritti**;
- Scrivendo prima i test il programmatore ha già un'idea più chiara di quello che il programma dovrà fare ed ha più chance di successo, la prende come una **sfida**; nel momento in cui si testa il programma e i test passano, il programmatore si sente **soddisfatto** ed è più **motivato** a proseguire;
- **Chiarezza dell'interfaccia** e del **comportamento** dettagliati;
- **Verifica automatica, ripetibile e dimostrabile**;
- Fiducia nel **cambiare** le cose.

## Tipi di test

Abbiamo i seguenti tipi di test:

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

## Test di unità

Un test di unità consiste in:

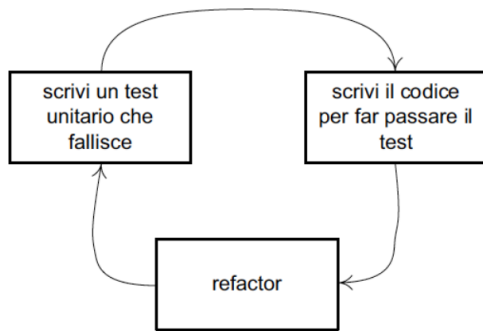
- **Preparazione**: creare l'**oggetto da verificare** (quest'ultimo chiamato **fixture**) e preparare altri oggetti/risorse necessari per l'esecuzione del test;
- **Esecuzione**: far **eseguire operazioni** alla fixture, richiedendo lo specifico comportamento da verificare;
- **Verifica**: **valutare**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

Il framework per test unitari più diffuso è la famiglia xUnit, per molti linguaggi (es. JUnit per Java).

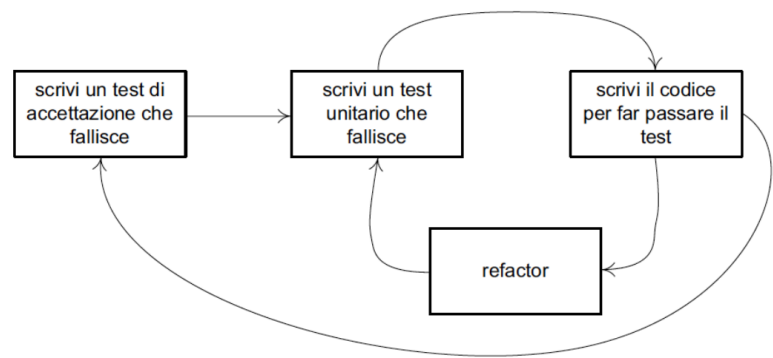
## Tipi di cicli per i test unitari

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

- **Scrivere un test unitario che fallisce**, per dimostrare la mancanza di una funzionalità o di codice (non scrivere codice di produzione prima di aver scritto un test unitario che fallisce e scrivere test unitari solo in quantità sufficiente a far fallire il test);
- Scrivere il **codice** più semplice possibile per **far passare il test**;
- **Riscrivere o ristrutturare (refactor)** il codice, migliorandolo, oppure **passare a scrivere il prossimo test unitario** (il refactor può essere effettuato anche dopo diversi test).



Il ciclo di base del TDD per i test unitari.



Il doppio ciclo del TDD.

Possiamo avere **due tipi di ciclo** nel TDD: **ciclo di base** o **doppio**. Quello base riflette il ciclo sopra descritto; quello doppio estende il ciclo di base. In quello **doppio** ci sono i **cicli di lavorazione brevi** (da pochi minuti/qualche ora) relativi ai **test unitari**, mentre il **ciclo più esterno** (con durata qualche giorno/un'intera iterazione di sviluppo) riguarda i **test di accettazione** (ciascuno dei quali potrebbe riguardare ad esempio un'intera esecuzione di uno specifico scenario di caso d'uso).

## Refactoring

Il **refactoring** è un **metodo strutturato e disciplinato**, utilizzato per **riscrivere/ristrutturare** del **codice esistente** senza modificarne il comportamento esterno (preservare il comportamento è importante, altrimenti si avrebbe probabilmente il fallimento di alcuni test unitari esistenti). Questo metodo applica **piccoli passi di trasformazione** (uno alla volta), in combinazione con la **ripetizione dei test** dopo ciascun passo, per dimostrare che il refactoring non abbia provocato una regressione (fallimento). Tutti i test unitari sostengono il processo di refactoring.

I vantaggi del refactoring riguardano:

- **Miglioramento continuo del codice:** è importante migliorare la struttura del codice dopo che è stato scritto, per evitare che diventi disordinato;
- **Preparazione al cambiamento:** il refactoring consente di preparare il codice all'introduzione di nuove funzionalità, nonché all'applicazione di cambiamenti.

Gli obiettivi e le attività del refactoring sono quelli di una buona programmazione:

- **Eliminare il codice duplicato;**
- Migliorare la **chiarezza**;
- **Abbreviare** i metodi lunghi;
- ...

## Quando fare refactoring

Conviene fare il refactoring del codice quando si verifica uno dei seguenti eventi:

- Viene soddisfatta la **Regola del Tre** (un programma deve essere soggetto a refactoring se una stessa porzione di codice è riutilizzata almeno tre volte nel programma: *aka copypaste permesso una sola volta!*);
- Quando si **aggiunge** una **funzionalità**;
- Quando si **corregge un bug** (attenzione, il refactoring non corregge i bug);
- Quando è **in corso** la **revisione** del codice.



## Processo di trasformazione

I passi di trasformazione consistono in:

- Assicurarsi che i **test passino**;
- Trovare un **code smell**;
- Determinare **come migliorare** il codice;
- **Effettuare** le **migliorie**;
- **Eseguire test** per assicurarsi che le cose funzionino ancora correttamente;
- **Ripetere** il ciclo di miglioramento/test finché il code smell rimane.

## Code Smell

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

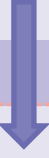

Li suddividiamo in più categorie (restando nell'ambito dell'OO programming; indichiamo i 10 code smell che abbiamo fatto a lezione):

- **Bloaters**: metodi, classi o in generale codice di dimensione sproporzionata (troppo grandi):
  - *Long Method*
  - *Large Class*
  - *Long Parameter List*
- **Object-Orientation Abusers**: implementazioni errate, incomplete o deboli dei principi della programmazione OO:
  - *Switch Statement*
  - *Refused Bequest*
- **Change Preventers**: parti di codice altamente accoppiate che impediscono cambiamenti rapidi:
  - *Shotgun Surgery*
- **Dispensables**: codice inutile, che sarebbe meglio togliere o integrare in altre parti per rendere il codice più pulito, efficiente e comprensibile:
  - *Duplicated Code*
  - *Data Class*
  - *Comments*
- **Couplers**: classi che hanno eccessivi accoppiamenti o classi che delegano eccessivamente:
  - *Feature Envy*

	Code Smell	Sintomo	Condizioni	Refactoring
Bloaters	<b>Long Method</b>	Un metodo ha troppe linee di codice (>10)	Se i può ridurre senza problemi la lunghezza	<i>Extract Method</i>
			Se ci sono problemi con variabili locali	<i>Replace Temp with Query</i>
				<i>Introduce Parameter Object</i>
				<i>Preserve Whole Object</i>
			Se non funzionano i precedenti	<i>Replace Method with Method Object</i>
			Se ci sono condizionali	<i>Decompose Conditional</i>
			Se ci sono dei cicli	<i>Extract Method</i>
	<b>Large Class</b>	Una classe ha troppi metodi	Se si può separare in più classi	<i>Extract Class</i>
			Se parte del comportamento può essere implementato in modo diverso o viene usato solo in pochi casi	<i>Exrtact Subclass</i>
			Se uno o più metodi si possono coerentemente spostare in altre classi	<i>Move Method</i>
	<b>Long Parameter List</b>	Un metodo ha troppi parametri	Se alcuni dei parametri sono risultati di chiamate a metodi di un altro oggetto	<i>Replace Parameter with Method Call</i>
			Se più parametri sono relativi a un oggetto	<i>Preserve Whole Object</i>
			Se i parametri non sono correlati	<i>Introduce Parameter Object</i>
OO Abusers	<b>Switch Statement</b>	Si ha uno switch molto lungo o più if...else	Se si può sfruttare il polimorfismo	<i>Replace Conditional with Polymorphism</i>
			Se si può isolare lo switch e assegnarlo ad una sottoclasse	<i>Extract Method</i>
				<i>Move Method</i>
	<b>Refused Bequest</b>	La sottoclasse non usa tutti i metodi e le proprietà ereditate, la gerarchia non è corretta	Se l'ereditarietà non ha molto senso	<i>Replace Inheritance with Delegation</i>
			Se l'ereditarietà è appropriata e si possono togliere metodi e parametri dalla superclasse per metterli dentro a una nuova superclasse	<i>Extract Superclass</i>
Change Preventers	<b>Shotgun Surgery</b>	Per modificare il codice vanno apportate minime modifiche a molte classi diverse	Se si può riorganizzare il comportamento raggruppando gli elementi in un'altra classe	<i>Move Method</i>
				<i>Move Field</i>
			Se la precedente procedura lascia classi quasi vuote	<i>Inline Class</i>

	Code Smell	Sintomo	Condizioni	Refactoring
Dispensables	<i>Duplicated Code</i>	Lo stesso codice appare in molti luoghi (>2)	Se ha senso rendere il codice un metodo chiamabile	<i>Extract Method</i>
	<i>Data Class</i>	Classe che funge solo da contenitore per i dati	Se si possono integrare nella Data Class i metodi usati altrove per manipolare i dati	<i>Move Method</i>
				<i>Extract Class</i>
	<i>Comments</i>	Troppi commenti	Se il commento spiega un'espressione lunga	<i>Extract Variable</i>
			Se il commento spiega una porzione di codice	<i>Extract Method</i>
			Se il metodo è già estratto ma molto commentato	<i>Rename Method</i>
			Se il commento afferma regole necessarie per il corretto funzionamento del sistema	<i>Introduce Assertion</i>
Couplers	<i>Feature Envy</i>	Un metodo fa accede ai dati di un altro oggetto più che ai propri dati	Se il metodo dovrebbe stare chiaramente da un'altra parte (usa tanti dati di un'altra classe)	<i>Move Method</i>
			Se solo una parte del metodo accede ai dati	<i>Extract Method</i>
			Se tutto il metodo usa tante funzioni di tante classi, e si può "smembrare" per poi suddividerlo tra le stesse	<i>Move Method</i>
				<i>Extract Method</i>

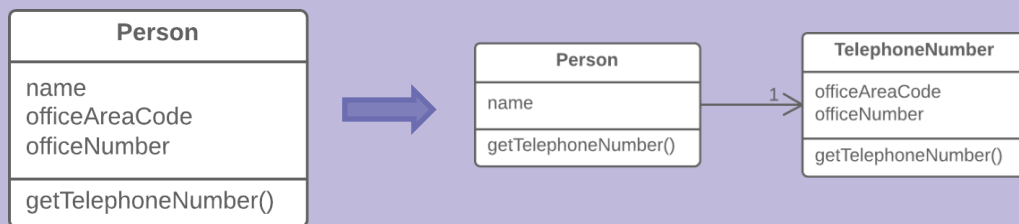
## Tabella dei Refactoring

Refactoring	Prima VS Dopo
<b>Extract Variable</b>  Si assegna un'espressione complessa ad una o più nuove variabili.	<pre>void renderBanner() {     if ((platform.toUpperCase().indexOf("MAC") &gt; -1) &amp;&amp;         (browser.toUpperCase().indexOf("IE") &gt; -1) &amp;&amp;         wasInitialized() &amp;&amp; resize &gt; 0 )     {         // do something     } }</pre>  <pre>void renderBanner() {     final boolean isMacOs = platform.toUpperCase().indexOf("MAC") &gt; -1;     final boolean isIE = browser.toUpperCase().indexOf("IE") &gt; -1;     final boolean wasResized = resize &gt; 0;      if (isMacOs &amp;&amp; isIE &amp;&amp; wasInitialized() &amp;&amp; wasResized) {         // do something     } }</pre>
<b>Extract Method</b>  Si crea un nuovo metodo estraendo codice da uno già esistente.	<pre>void printOwing() {     printBanner();      // Print details.     System.out.println("name: " + name);     System.out.println("amount: " + getOutstanding()); }</pre>  <pre>void printOwing() {     printBanner();     printDetails(getOutstanding()); }  void printDetails(double outstanding) {     System.out.println("name: " + name);     System.out.println("amount: " + outstanding); }</pre>

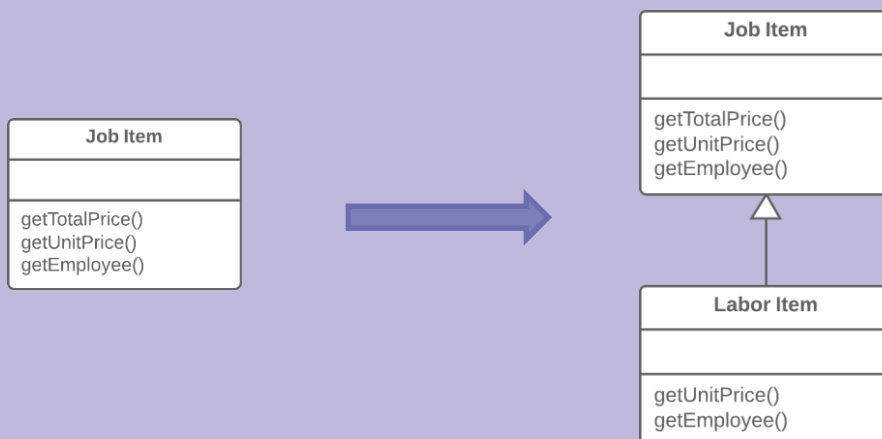


**Refactoring****Prima VS Dopo****Extract Class**

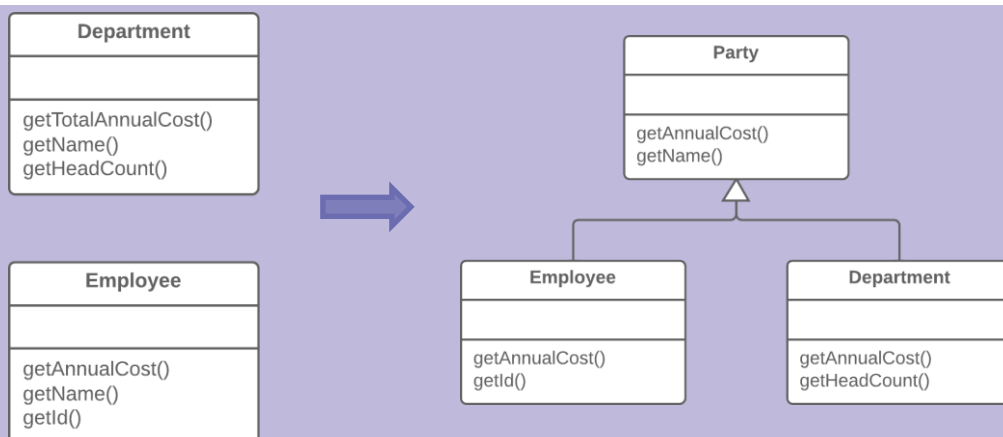
Si estrae una parte della logica o del comportamento di una classe in una nuova classe.

**Extract Subclass**

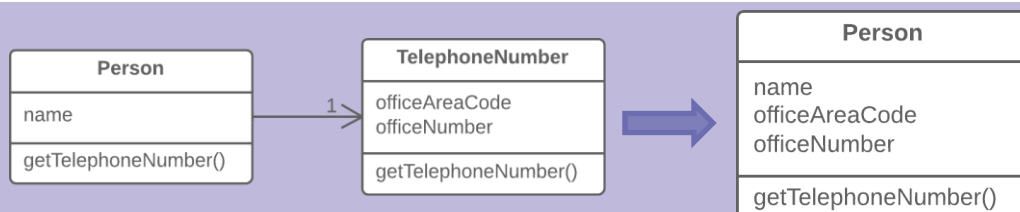
Si estrae parte del comportamento e dei campi che vengono usati raramente o possono essere implementati diversamente per metterli nelle sottoclassi.

**Extract Superclass**

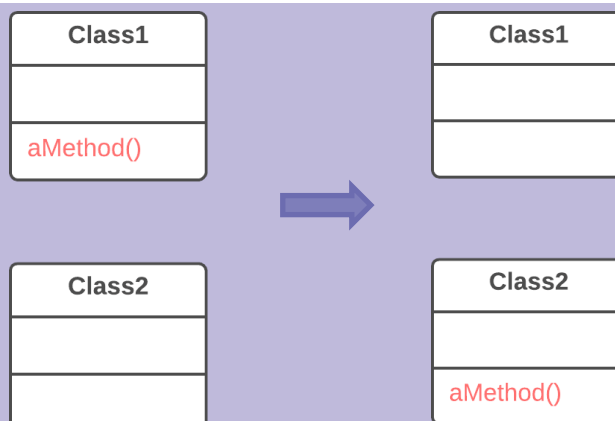
Si estraggono le parti comuni a più classi e si mettono in una nuova superclasse.

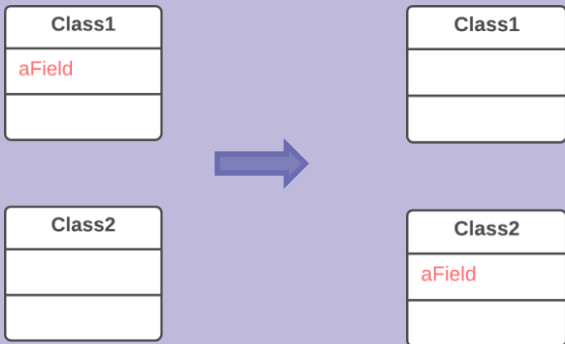
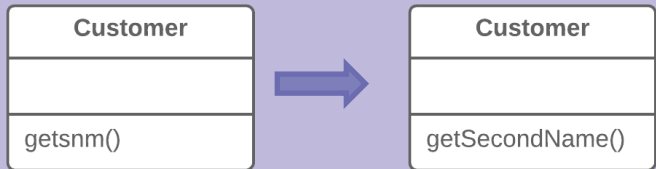
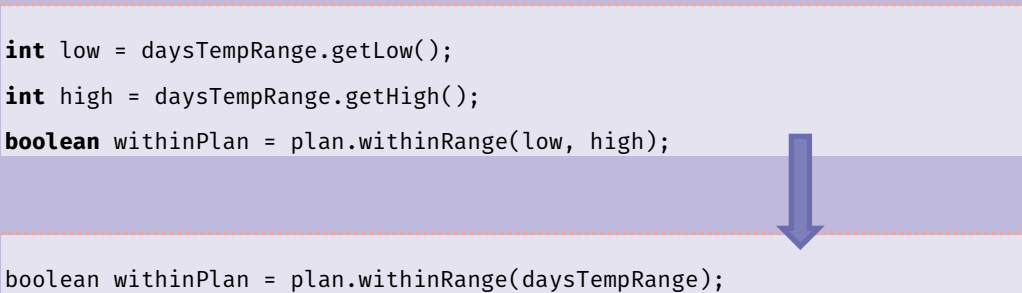
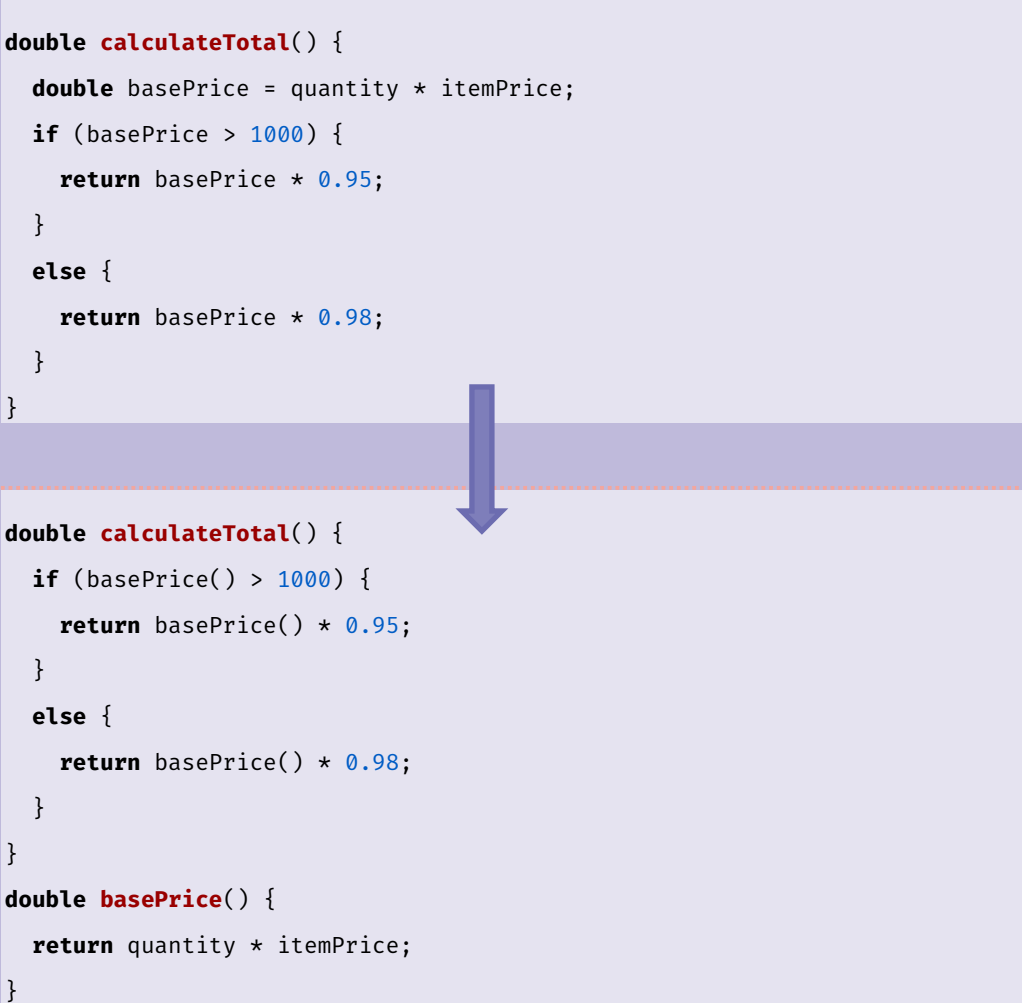
**Inline Class**

Si incorpora una classe quasi vuota in una classe che la sfrutta.

**Move Method**

Si sposta un metodo da una classe a un'altra.



Refactoring	Prima VS Dopo	
<b>Move Field</b>  Si sposta un attributo da una classe a un'altra.		
<b>Rename Method</b>  Si rinomina un metodo.		
<b>Preserve Whole Object</b>  Si passa l'intero oggetto come parametro al posto di passare i suoi attributi.	 <pre>int low = daysTempRange.getLow(); int high = daysTempRange.getHigh(); boolean withinPlan = plan.withinRange(low, high);</pre> <pre>boolean withinPlan = plan.withinRange(daysTempRange);</pre>	
<b>Replace Temp with Query</b>  Si rimpiazza una variabile temporanea con una chiamata a un metodo.	 <pre>double calculateTotal() {     double basePrice = quantity * itemPrice;     if (basePrice &gt; 1000) {         return basePrice * 0.95;     }     else {         return basePrice * 0.98;     } }</pre> <pre>double calculateTotal() {     if (basePrice() &gt; 1000) {         return basePrice() * 0.95;     }     else {         return basePrice() * 0.98;     } }  double basePrice() {     return quantity * itemPrice; }</pre>	

*Refactoring**Prima VS Dopo**Replace Method with Method Object*

Si crea un nuovo oggetto e rimpiazza un metodo con un nuovo metodo appartenente all'oggetto creato.


```
class Order {
    // ...
    public double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // Perform long computation.
    }
}

class Order {
    // ...
    public double price() {
        return new PriceCalculator(this).compute();
    }
}

class PriceCalculator {
    private double primaryBasePrice;
    private double secondaryBasePrice;
    private double tertiaryBasePrice;

    public PriceCalculator(Order order) {
        // Copy relevant information from the
        // order object.
    }


    public double compute() {
        // Perform long computation.
    }
}
```


*Replace Parameter with Method Call*

Si toglie un parametro dal metodo A derivante da una chiamata a un metodo B e lo si sostituisce con una chiamata a metodo B all'interno del metodo A.

```
int basePrice = quantity * itemPrice;
double seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);

int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```



*Refactoring**Prima VS Dopo***Replace Conditional with Polymorphism**

Si rimpiazza un lungo blocco condizionale dividendo i vari casi su sottoclassi diverse e applicando il polimorfismo.

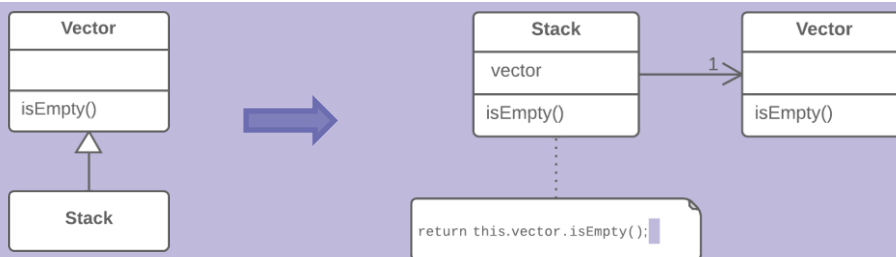
```
class Bird {  
    // ...  
    double getSpeed() {  
        switch (type) {  
            case EUROPEAN:  
                return getBaseSpeed();  
            case AFRICAN:  
                return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;  
            case NORWEGIAN_BLUE:  
                return (isNailed) ? 0 : getBaseSpeed(voltage);  
        }  
        throw new RuntimeException("Should be unreachable");  
    }  
}
```



```
abstract class Bird {  
    // ...  
    abstract double getSpeed();  
}  
  
class European extends Bird {  
    double getSpeed() {  
        return getBaseSpeed();  
    }  
}  
  
class African extends Bird {  
    double getSpeed() {  
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;  
    }  
}  
  
class NorwegianBlue extends Bird {  
    double getSpeed() {  
        return (isNailed) ? 0 : getBaseSpeed(voltage);  
    }  
}  
  
// Somewhere in client code  
speed = bird.getSpeed();
```

**Refactoring****Prima VS Dopo****Replace Inheritance with Delegation**

Si sostituisce una ereditarietà con una relazione di delega.

**Decompose Conditional**

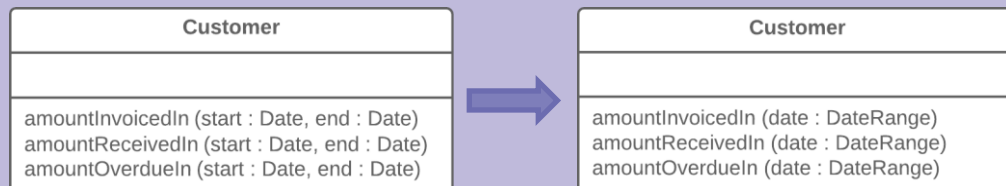
Si scompongono le parti complicate del blocco condizionale in metodi separati: la condizione, il blocco then e il blocco else.

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
    charge = quantity * winterRate + winterServiceCharge;
}
else {
    charge = quantity * summerRate;
}
```

```
if (isSummer(date)) {
    charge = summerCharge(quantity);
}
else {
    charge = winterCharge(quantity);
}
```

**Introduce Parameter Object**

Si raggruppano dei parametri all'interno di un nuovo oggetto in qualità di attributi.

**Introduce Assertion**

Si introduce un'asserzione per esprimere una condizione necessaria per il proseguimento dell'esecuzione.

```
double getExpenseLimit() {
    // Should have either expense limit or
    // a primary project.
    return (expenseLimit != NULL_EXPENSE) ?
        expenseLimit :
        primaryProject.getMemberExpenseLimit();
}
```

```
double getExpenseLimit() {
    Assert.isTrue(expenseLimit != NULL_EXPENSE || primaryProject != null);

    return (expenseLimit != NULL_EXPENSE) ?
        expenseLimit :
        primaryProject.getMemberExpenseLimit();
}
```

# DOMANDE DI TEORIA

Si assume che chi sia arrivato qui riesca a consultare il file per trovare la risposta. Quindi NO, non ci sono le risposte QUI, ma garantiamo che nel file ci sono tutte!

## • Introducimi...

- I processi software
- Unified Process
- Scrum e le differenze con UP
- Il metodo agile
- Sviluppo iterativo, incrementale, evolutivo
- Le attività di processo
- L'ideazione
- L'elaborazione
- L'Analisi OO
- La Progettazione OO

## • Cosa sono...

- I messaggi
- Le operazioni di sistema
- Gli eventi di sistema
- Gli scrum
- Le classi concettuali
- Gli oggetti

## • Spiegami il documento e perché lo usiamo...

- Modello di Dominio
- Caso d'Uso
- Diagramma dei Casi d'Uso
- Diagramma di Sequenza di Sistema
- Contratti
- Diagramma delle Classi Software
- Diagramma dei Package
- Diagrammi di Sequenza
- Macchine a Stati
- Diagramma di Attività
- Diagramma di Comunicazione
- Diagrammi di Interazione

## • Spiegami...

- L'architettura logica
- Il TDD
- Il refactoring
- Le fasi di UP
- Le iterazioni di UP
- Pre-condizioni e Post-condizioni
- Come passiamo dall'Analisi alla Progettazione

## • Quali tipi ci sono di...

- Code smell
- Refactoring
- Pattern GoF (categorie)
- Strati architetturali
- Partizioni degli strati Architetturali
- Test (di unità)
- Controller
- Ruoli in Scrum
- Adapter

## • Spiegami e disegna il pattern GoF

## • Spiegami un pattern GRASP

## • Spiegami come sistemare questo code smell

## • Spiegami come si traduce in codice...

## • Che relazione hanno...

- Analisi e Progettazione
- Modello di Dominio e Modello di Progetto
- Contratti e SSD
- SSD e Casi d'Uso
- SD e Diagramma delle Classi Software

## • Che differenza c'è fra...

- Responsabilità e metodi
- MdD e Diagramma delle Classi Software
- Proprietà e attributo
- Fase e disciplina
- SD e SSD
- Composizione e aggregazione
- Classe astratta e artificiale
- Pattern GRASP e GoF

## • Domanda V/F...

- Il refactoring corregge i bug
- Produciamo codice durante la prima iterazione (iterazione zero)
- Iniziamo a implementare dalla classe che ha più operazioni