



# Analisi e progettazione del software

@nicotb



**SOFTWARE** : programma PC dotato di una documentazione

SI DIVIDE IN

**GENERICO**

↳ per un bacino d'utenza

**PERSONALIZZATO**

↳ costruito per un cliente

⇒ un SW si può costruire

deve essere:

da zero

da una base (altro sw)

personalizzando altri programmi

- \* MANTENIBILE
- \* AFFIDABILE
- \* EFFICIENTE
- \* ACCETTABILE

La disciplina che produce nuovi sw rispettosi delle linee guida è quella dell'**INGEGNERIA DEL SOFTWARE**

↓  
teorie e metodi progettistici

L'approccio in sé è definito invece come **PROCESSO SOFTWARE**

→ riguarda la costruzione, il rilascio e la manutenzione del codice, definisce i ruoli del team e l'organizzazione temporale

↳ le ATTIVITÀ fondamentali del processo sono:

- { \* ANALISI DEI REQUISITI → ricavati dal linguaggio
- \* PROGETTAZIONE → soluzione naturale che soddisfa i requisiti
- \* SVILUPPO
- \* CONVALIDA → il prodotto "finito" soddisfa i requisiti?
- \* RILASCIO
- \* EVOLUZIONE → aggiornamenti in base a nuove esigenze/opportunità

nell'OTTICA OBJECT ORIENTED, si definiscono classi di dominio per descrivere concetti od oggetti del problema in fase di ANALISI, mentre in fase di PROGETTAZIONE si definiscono classi software per modellare i concetti in classi e oggetti software utilizzabili nel codice

La modellazione e gestione software si può gestire mediante programmi specifici, e si può rappresentare graficamente mediante UML, un linguaggio visuale

↳ molto flessibile e si adatta praticamente a tutto (piattaforma, processo, domini...)

classi e oggetti



UML può modellare sia l'aspetto statico che quello dinamico di una struttura ad oggetti



Si usa sia in ANALISI che in PROGETTAZIONE, e in particolare si distingue tra software e concetti

ciclo di vita  
e interazioni  
degli oggetti

concetti o oggetti  
del mondo reale  
(scissi dal codice)

MODELLO DI DOMINIO

i componenti software come classi



MODELLO DI PROGETTO

# Processi Software

ne esistono diversi tipi

## \* Processo software a cascata

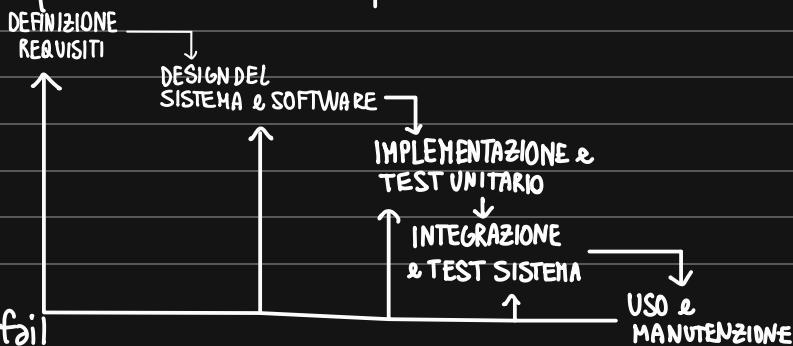
→ esecuzione sequenziale di attività predefinite

!! CRITICO !!

NON è flessibile ai cambiamenti; i requisiti devono essere stabili

↓  
non adatto a progetti

complessi, alto rischio di fail



"Il feedback arriva solo alla fine"

## \* Sviluppo ITERATIVO, INCREMENTALE, EVOLUTIVO



modello di organizzazione del processo software, multiple ITERAZIONI di durata fissa (time-boxed in settimane)



Si svolgono tutte le attività di processo e si produce software esegibile

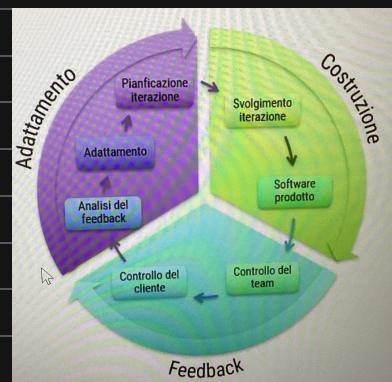


il materiale prodotto sarà utile all'iterazione successiva (INCREMENTALE), che verrà influenzata dalla precedente per definire gli obiettivi e l'EVOLUZIONE del progetto



in questo modo si ha margine di modifica/cambiamento dei requisiti e restano STABILLI nel tempo

- ✓ flessibile alle modifiche
- ✓ riduce i rischi maggiori (gestiti subito)
- ✓ progresso visibile
- ✓ feedback precoce
- ✓ gestisce meglio la complessità



Com'è fatta un'ITERAZIONE ?

Sì sceglie un sottoinsieme di REQUISITI, si procede con PROGETTAZIONE, poi IMPLEMENTAZIONE e infine TESTING.

⇒ Sono TIME-BOXED, un'iterazione dura tra le 2 e 6 settimane e variano per carico di lavoro



nelle fasi iniziali del progetto si dedicherà più tempo all'analisi dei requisiti

⇒ i requisiti vengono FISSATI per tutta l'iterazione per un lavoro senza interruzioni

↳ a metà iterazione si può tentare un "cambio piano" anche in base al tempo/lavoro rimasto

I rischi sono fondamentali e la pianificazione ne è influenzata parecchio, è fondamentale stabilizzare quanto prima il nucleo di un ARCHITETTURA

Il cliente determina la priorità dei requisiti

## \* Unified Process (UP) → visuale UML

↳ un processo ITERATIVO, INCREMENTALE ed EVOLUTIVO per lo sviluppo ad oggetti



FLESSIBILE e APERTO ad altri metodi iterativi



CASI D'USO e RISCHI ne fanno da pilastro



quelli maggiori sono trattati nelle prime iterazioni: per stabilizzare l'architettura e i test sono fondamentali come il coinvolgimento utente

Con UP si dividono le varie iterazioni in quattro FASI:

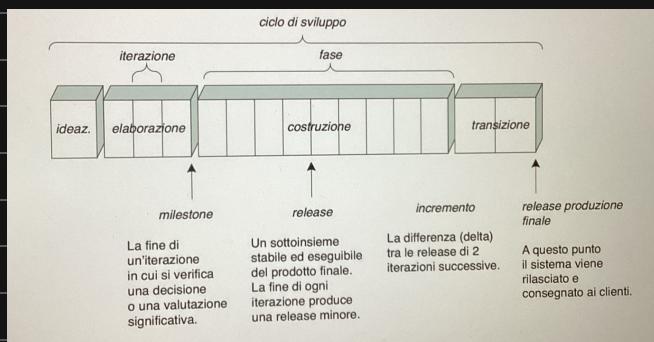
\* **IDEAZIONE**: avvio del progetto, studio e stime sui costi e tempi

\* **ELABORAZIONE**: sviluppo del nucleo architettura, in quadatura migliore e identif. maggiori requisiti

\* **COSTRUZIONE**: realizzazione capacità, preparazione al rilascio

\* **TRANSIZIONE**: completamento del prodotto

Le fasi sono costituite da  
**ITERAZIONI**, che a loro volta  
comprendono **DISCIPLINE**



Com'è fatta un'ITERAZIONE in UP?

Una sorta di mini progetto in una finestra di tempo (esclusa la prima iterazione, quella di IDEAZIONE)

- ↳ include:
- PIANIFICAZIONE
  - ANALISI e PROGETTAZIONE
  - COSTRUZIONE
  - TEST e INTEGRAZIONE
  - RILASCIO

} Il prodotto finale viene da un'insieme di ITERAZIONI sovrapponibili tra team

varia in base allo stato del progetto

Pratiche da evitare con UP

- \* definire tutti i requisiti all'inizio
- \* non testare il nucleo
- \* far più elaborati possibili e non di qualità
- \* pensare che una fase = una sola attività
- \* finire il progetto prima di implementarlo

## \* Metodo AGILE

Una forma di sviluppo iterativo che promuove l'agilità, una risposta rapida e flessibile ai cambiamenti

- sviluppo e pianificazione iterativa
- consegne INCREMENTALI
- Semplicità, leggerezza...
- pratiche agili: UML solo delle parti complesse, refactoring...

eSEMPIO: UP AGILE, un UP più leggero

## \* Metodo SCRUM

un metodo AGILE che consente il rilascio dei prodotti con il valore più alto nel **minor tempo possibile**



incentrato su ORGANIZZAZIONE e GESTIONE PROGETTO

- requisiti emergono durante le iterazioni e non prima dell'IMPLEMENTAZIONE

- UML solo dove necessario

Lo SCRUM è un incontro quotidiano del team dove si stabiliscono le priorità del giorno consultando i BACKLOG (lista delle cose da fare). Le iterazioni sono chiamate SPRINT (2-4 settimane) e la quantità di lavoro fattibile in uno sprint è detta VELOCITY

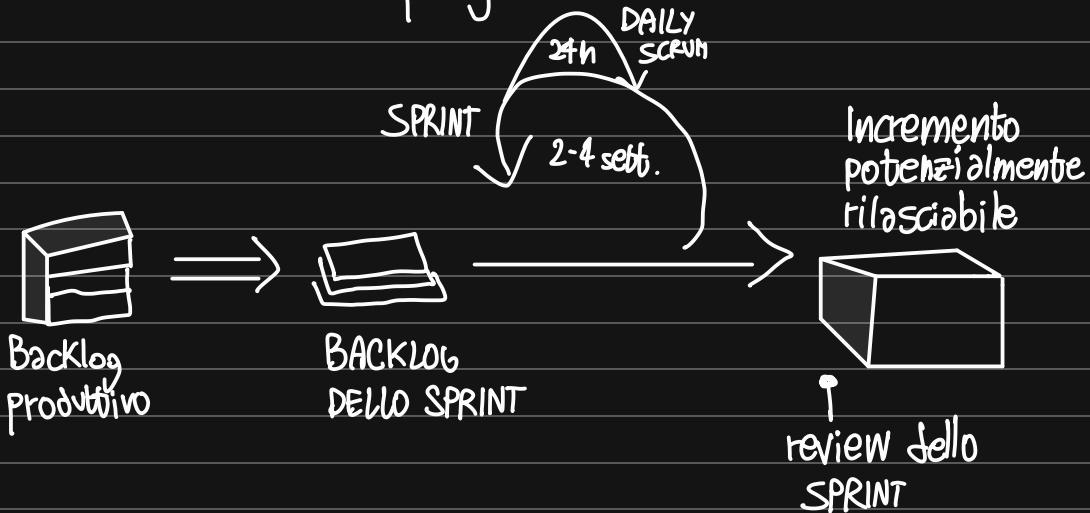
I ruoli sono:

\* SCRUM MASTER: la figura che si occupa di far applicare lo SCRUM

↳ tramite tra team e altri team

\* TEAM DI SVILUPPO: gli sviluppatori (non più di 7 per SCRUM)

\* PRODUCT OWNER: il cliente a cui è rivolto il progetto



# Unified process

## Fase 1 : IDEAZIONE

Quante iterazioni? UNA

Quanto dura? In media qualche settimana

Discipline coinvolte:

- MODELLAZIONE del BUSINESS
- REQUISITI

⇒ L'IDEAZIONE è il passo iniziale in UP per definire la visione del progetto

→ si raccolgono le informazioni necessarie ad una visione comune e decidere la fattibilità

↓  
UNA sola iterazione chiamata ZERO, cioè quella che avvia tutto

↓  
vengono stimati costi, tempi e il 10% dei REQUISITI FUNZIONALI (i casi d'uso) e i NON funzionali più critici

↳ durante questa fase si pianifica la PRIMA iterazione di:  
**ELABORAZIONE** (in cui si produce SW)

Importante quindi definire gli ATTORI principali in questa fase per definire la portata del progetto.

# Requisiti

→ capacità o condizioni a cui il progetto deve essere conforme

Devono essere COMPLETI, NON AMBIGUI e COERENTI

Si dividono in

**NON FUNZIONALI**

↓  
relativi alle proprietà di  
sistema

↳ qualità del progetto, IDE, vincoli ecc...

**SPECIFICHE SUPPLEMENTARI**

I requisiti, in generale, sono  
**FONDAMENTALI** in UP

↓

essendo un processo flessibile, i primi sono individuati  
(collaborando col cliente) in **IDEAZIONE** e il resto in **ELABORAZ.**

possono essere modificati più tardi, ma **col tempo si STABILIZZANO**

|| 25% dei requisiti cambia dopo l'ideazione

**FUNZIONALI**: i servizi che il sistema deve offrire, le risposte (output) a determinati input

↓  
**CASI D'USO**

→ talvolta le specifiche suppl. possono portare ad **AGGIUNGERE** casi d'uso che le soddisfino

I requisiti non funzionali sono di due categorie:

\* **OBBIETTIVI**: intenzioni generali, ideali da raggiungere ma non quantificabili (es. **FACILITÀ D'USO**)

\* **REQUISITO NON FUNZ. VERIFICABILE**: una dichiarazione con misure oggettivamente **QUANTIFICABILI**

Lo scopo degli obiettivi è capire le intenzioni dell'utente e definire su di esse requisiti non funz. verificabili su criteri oggettivi

Come acquisisco i requisiti? Attraverso metodi agile, quali:

- Scrittura dei casi d'uso con i **Clienti**
- meeting tra sviluppatori e **clienti** (o loro rappresentanti)
- raccolta **feedback** dai **clienti** a fine iterazione

Gli elaborati dei requisiti vengono scritti in linguaggio naturale chiaro sia al cliente che allo sviluppatore (niente gergo tecnico)

→ **FORMATO STANDARD** → ENFASI TESTUALE su parti fondamentali:  
verbi importanti      DEVE (req. obbligatori)  
                          DOVREBBE (desiderabile)

**Casi d'uso** → storie scritte, di lunghezza ridotta, che rappresentano dialoghi fra un **attore** o più e un **sistema** che svolge un compito

↓  
utili per scopare  
requisiti funzionali e non (verificabili)

sono dei veri e propri testi, contratti relativi al comportamento del sistema

↓  
UP in particolare ne incoraggia l'uso e la modellazione, l'elaborato prodotto è il **MODELLO DEI CASI D'USO** e, eventualmente, il **DIAGRAMMA UML**

↓  
insieme di tutti i casi d'uso descritti, può comprendere anche i **DIAGRAMMI** di **SEQUENZA** di SISTEMA e i **CONTRATTI**

⇒ In ideazione vengono dettagliati solo il 10-20% dei casi d'uso (quelli "chiave" per l'architettura)

I casi d'uso sono **comprendibili** per i clienti, risaltano le **necessità** degli utenti (e obiettivi) e sono utili per test

## Definizioni utili ai casi d'uso

- **ATTORE**: qualcosa/qualcuno dotato di un comportamento  
↓ può essere a sua volta:
  - \* **PRIMARIO**, utilizza in modo diretto il sistema
  - \* **FINALE**, vuole che il sistema sia usato per suoi obiettivi
  - \* **DI SUPPORTO**, offre un servizio al sistema

\* **FUORI SCENA**: ha un interesse nel caso d'uso  
ma non interviene al suo interno

Attore primario e finale spesso **COINCIDONO** perché il primario  
usa il sistema per raggiungere i propri obiettivi;

- **SCENARIO**: sequenza specifica di azioni e interazioni  
tra sistemi e attori  
↳ un **percorso** di successo/fallimento formato  
da passi di tre tipi:
  - un'interazione tra attori
  - un cambio di stato del sistema
  - una validazione

- **CASO D'USO**: una collezione di **scenari** correlati:

## Tipi di caso d'uso

——→ **Formati**

\* **BREVE**: riepilogo coinciso di un solo  
paragrafo, relativo soltanto  
allo scenario di **successo**

\* **INFORMATIVO**: più paragrafi per **vari scenari**

\* **DETtagliato**: tutto nel dettaglio, inclusi passi  
e variazioni  
↳ dalle 3 alle 10 pagine

# → Livello

- **OBBIETTIVO UTENTE**: il caso d'uso consente di raggiungere un proprio obiettivo
- **SOTTO-FUNZIONE**: il caso d'uso riguarda una sola funzione di sistema
- **SOMMARIO**: obiettivo più ampio, raggiungibile in più utilizzi di sistema

L'approccio per descrivere il sistema è detto a **SCATOLA NERA**: non si descrive il comportamento del sistema ma le sue responsabilità

↳ nei casi d'uso quindi si specifica cosa fa e non come

→ si scelgono i confini  
di sistema

→ si identificano gli attori  
primari

→ si identificano i loro  
obiettivi

→ si definiscono i casi d'uso che soddisfano i loro obiettivi ✓

I casi d'uso devono essere **essenziali** (niente fronzoli) e **completi**, ogni obiettivo utente deve avere un suo caso d'uso

↳ non vanno inseriti dettagli implementativi

## Diagramma dei casi d'uso

Fornisce contesto ai casi d'uso, più per aver un quadro generale completo, si includono solo quelli a livello utente



associazioni orientate da chi inizia all'oggetto



Tre tipi di relazione:

- **INCLUDE**: relazione tra un caso d'uso e un altro caso d'uso considerato incluso nel caso base (quindi eseguito con esso)



- **EXTEND**: Caso d'uso che estende un caso base: se soddisfatta una condizione, viene eseguito per poi tornare al punto di estensione del caso base



• GENERALIZZAZIONE: ereditano tutto, possono aggiungere o sovrascrivere



gli attributi, invece, ereditano  
valori e relazioni:



## FASE 2.1: ELABORAZIONE (1^ ITERAZIONE)



si costruisce il modello dell'architettura, vengono risolti i fattori ad alto rischio e si stima il lavoro e risorse necessarie

DURATA: 2-6 settimane (per iterazione)

Vengono prodotti: \* MODELLO DI DOMINIO

\* MODELLO DI PROGETTO  
(diagrammi della progettazione logica)

CLASSI SW  
ITERAZIONE  
PACKAGE

\* DOCUMENTO

DELL'ARCHITETTURA: riassume le varie viste dell'architettura e comprende tutte le decisioni importanti + motivazioni

\* MODELLO DEI DATI: schema della base di dati

\* STORYBOARD DEI CASI D'USO E PROTOTIPI UI (interfaccia, navigabilità...)

I CASI D'USO vengono raffinati e si iniziano SSD e contratti.

Durante l'elaborazione si esegue un'indagine seria e completa della durata di 2 o più iterazioni, e si produce l'eseguibile del nucleo dell'architettura (e testato)

1° ITERAZIONE → centrata sull'ARCHITETTURA e guidata dal RISCHIO

Si implementano i casi d'uso dettagliati in precedenza

→ si affrontano gli aspetti più critici del progetto

↳ quelli scelti in pianificazione dell'iterazione

↳ i requisiti per le fasi successive sono quelli ancora a rischio

Discipline dell'elaborazione - 1

# MODELLAZIONE DEL BUSINESS

**Analisi ad oggetti:** l'investigazione e la comprensione del problema che il sistema deve risolvere



gli oggetti sono i concetti del dominio

si trattano 3 aspetti:

\* INFORMAZIONI DA GESTIRE : MODELLO DI DOMINIO

\* FUNZIONI : SSD

\* COMPORTAMENTO: cambiamenti di stato del sistema come conseguenza delle funzioni

si rappresenta in UML

↳ CONTRATTI

**MODELLO DI DOMINIO:** rappresentazione visuale (diagramma) delle classi concettuali (non sw)

viene modellato il dominio  
in cui opera il sistema ed è  
l'influenza per gli altri elaborati (tra cui classi sw)

→ concetti del mondo reale

Si consulta : \* in ANALISI per comprendere il dominio del sistema da realizzare e definire un linguaggio comune

\* In PROGETTAZIONE come spunto per modellare lo stato di dominio

Al suo interno troviamo



\* CLASSI CONCETTUALI: descrittore di oggetti con caratteristiche in comune



nomi chiari rispetto alla realtà trattata

Esistono le classi DESCRIZIONE, contenenti informazioni su qualcosa' altro (datatype, ad esempio INDIRIZZO (via, paese, cap))

\* ATTRIBUTI: proprietà elementate degli oggetti, ciascuno ha proprio valore, tipo (boolean, string...) e visibilità



public, private, protected, package



DEFAULT

\* ASSOCIAZIONI: relazione semantica tra classi, connessioni tra le loro istanze

→ - SONO SIGNIFICATIVE

- non corrispondono alle relazioni livello SW



Esistono associazioni: × AGGREGAZIONE, relazioni intero / parte in cui l'aggregato ESISTE indipendentemente dalle sue parti e viceversa



× **COMPOSIZIONI**: un tipo forte di aggregazione, ogni parte owns appartiene ad un solo composto e il composto è responsabile delle parti.

—→ × **GENERALIZZAZIONI**

Discipline dell'elaborazione - 2

## REQUISITI

↓  
dopo aver raffinato i casi d'uso, si cominciano gli **SSD** e i **CONTRATTI**

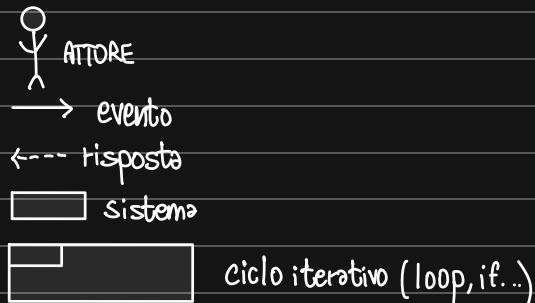
**DIAGRAMMI DI SEQUENZA**: elaborato che mostra gli eventi I/O  
**DI SISTEMA (SSD)** tra ATTORI e SISTEMA

il sistema reagisce agli eventi esterni: l'utente genera un evento di sistema e il sistema esegue le operazioni nell'ordine dettato dagli SSD

↳ Si ragiona ancora per DOMINIO, non per METODI  
(si astrae il sistema, scatola nera)

—→ Si modella prima lo **SCENARIO PRINCIPALE** di successo e poi gli alternativi

## LEGENDA



**CONTRATTI**: descrivono nel dettaglio i cambiamenti agli oggetti del dominio in seguito alle **operazioni di sistema** (input)

↓  
non sono obbligatori ma  
complementano i casi d'uso  
↳ si stilano solo i necessari

funzioni pubbliche offerte tramite  
interfaccia pubblica

## Struttura di un contratto

Contratto <Nome>

OPERAZIONE → Nome, parametri e tipo

CASI D'USO → Dove si verifica  
COINVOLTI

PRE-CONDIZIONI → Condizioni per eseguire l'operazione

POST-CONDIZIONI → Condizioni vere terminata l'esecuzione,  
cambi di stato degli oggetti del dominio

↓  
tre tipi di cambi di stato

- ✗ CREAZIONE/DISTRUZIONE DI Istanze
- ✗ ATTRIBUTI MODIFICATI (valori)
- ✗ FORMAZIONE/ROTTURA DI ASSOCIAZIONI di Istanze

## Dai Requisiti alla Progettazione

↓  
analizzare i requisiti serve a fare la cosa giusta

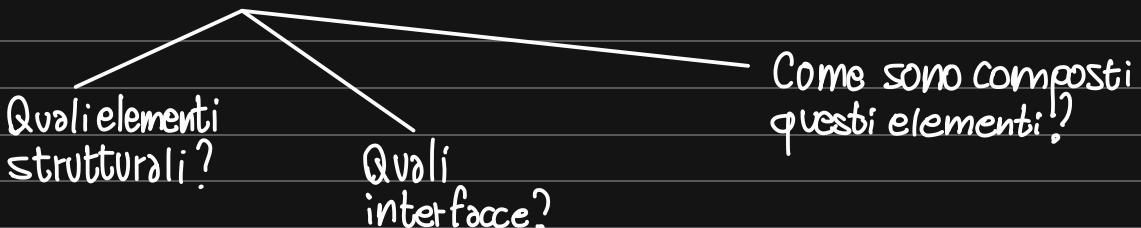
→ progettare, invece, serve a far la cosa **bene**

↓  
soluzione che soddisfi i requisiti

I passaggi tra le due sono frequenti in  
ogni iterazione (requisiti → prog)

## ARCHITETTURA SOFTWARE

↓  
insieme delle decisioni significative sull'organizzazione  
di un sistema SW



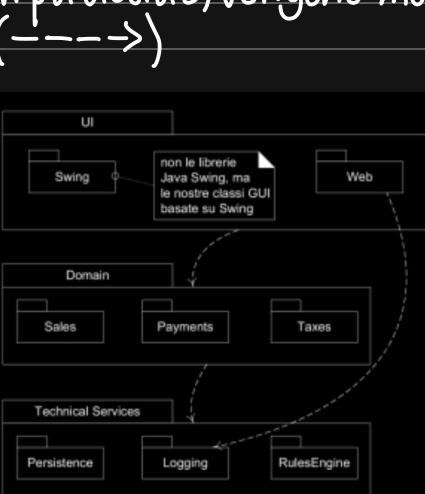
In breve: RACCOLIE TUTTE le **DECISIONI** importanti prese

Esistono viste specifiche riguardo l'architettura per isolare i livelli specifici SW

↓  
**ARCHITETTURA LOGICA** → divide le classi SW in strati/package/sottoinsiemi

non vengono prese  
qui le decisioni sulla distribuzione  
tra processi e computer

Si rappresenta in UML e fa parte del Modello di Progetto



esso è un namespace

si possono aver più classi con stesso nome in diversi package

"collezione di entità"

Si può dividere l'AL in:

- **Livelli (tier):** indica un nodo fisico di elaborazione

- **Strato (layer):** sezione verticale

- **Partizione (partition):** divisione orizzontale di sottosistemi di uno strato

# LOGICA A STRATI → stile di architettura logica



Ciascuno strato è un gruppo grosso di classi/package e sottosistemi che ha responsabilità condivise nei confronti del sistema

↑ Superiori : servizi specifici applicativi

Inferiori : servizi generali e a basso livello

Gli strati comunicano tra loro, in particolare gli alti ricorrono ai servizi dei più bassi

→ in base al

\* Uno strato può richiamare solo i servizi di uno strato immediatamente sottostante (strati stretti)

\* Uno strato può richiamare servizi di strati più bassi di diversi (strati rilassata)

UI

LOGICA APPL.

SERVIZI TECNICA

Perché usati?

✓ INDIPENDENZA delle MODIFICHE  
permettono modifiche locali al codice

✓ SEPARAZIONE tra LOGICA APPLICATIVA  
e UI/BUSINESS

✓ ACCOPPIAMENTI  
TRA DIVERSE AREE DI INTERESSE

Strati più comuni.

✓ RIVILITIZZO DI FUNZIONI  
✓ SVILUPPO DA PARTE DI PIÙ TEAM

Un esempio rilevante di strato è quello della **LOGICA APPLICATIVA**

↳ riguarda gli oggetti di dominio e si divide in due sotto strati:

STRATO DEL  
DOMINIO



oggetti del dominio

STRATO  
APPLICATION



oggetti che gestiscono  
il workflow da e verso gli  
oggetti di dominio

**BEST PRACTICE:** Create oggetti  
con nomi e info simili al Modello di Dominio  
per facilitare la comprensione della rappresentazione

**NOTA BENE:** non bisogna mettere logica applicativa in un oggetto

UI

↳ modello — controllet — vista

Discipline dell'elaborazione -  
**PROGETTAZIONE**

E si stono diversi approcci alla progettazione.

• **CODIFICA:** progettare durante il coding, mediante TDD e  
refactoring



tool di reverse-engineering

• DISEGNO → CODIFICA : Convertire UML a codice ↗

• SOLO DISEGNO → UNO STRUMENTO converte il disegno in codice

↳ provoca costi aggiuntivi

La modellazione della progettazione è da fare in gruppi ed esistono modelli:

\* STATICI: Diagramma delle classi, package e deployment

\* DINAMICI: Diagrammi di sequenza, comunicazione, Macchine a stati e Attività



qui applichiamo  
PATTERN e progettazione  
guidata dalle RESPONSABILITÀ

↳ riguarda la logica,  
comportamento  
e corpo dei metodi

## DIAGRAMMI DI INTERAZIONE

Illustrano il modo in cui gli oggetti interagiscono tramite scambio di messaggi per eseguire un compito determinato



l'interazione viene cominciata da  
un found message

Quattro tipi di  
Diagrammi:

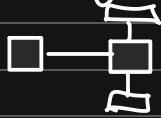
\* Sequenza (SD)  $\xrightarrow{\text{UML}}$  \* Comunicazione (CD)

\* Interazione generale \* Temporizzazione

Le interazioni sono mostrate tra i lifeline verticali negli SD



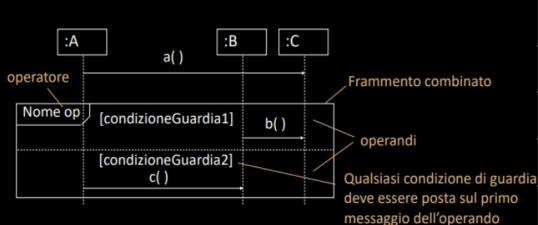
Nei CD invece avvengono in un formato a grafo tra oggetti



## DIAGRAMMI DI SEQUENZA (SD)

Del tutto simili agli SSD, ma la differenza è che si lavora a livello **SOFTWARE** e non di dominio (quindi si specificano tutti i componenti)

	Messaggio sincrono
	Messaggio asincrono
	Messaggio di ritorno
:A	Creazione dell'oggetto
X	Distruttore dell'oggetto
●	Messaggio trovato



↓  
non più a scatola nera

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

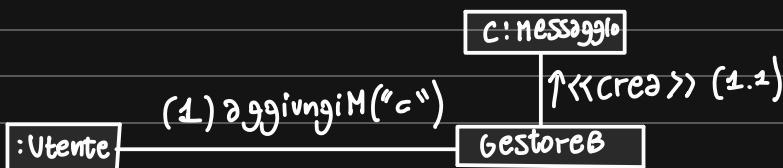
\* Le linee di vita rappresentano oggetti o anche collezioni di oggetti

\* Nelle note si possono mettere vincoli/pezzi di codice utili

\* Si possono inserire gli stati degli oggetti (invarianti di stato)

## DIAGRAMMI DI COMUNICAZIONE (CD)

Identici agli SD, ma le lifeline sono create nel momento in cui c'è un messaggio che ne invoca il metodo di creazione, vengono numerati per seguire il giusto ordine

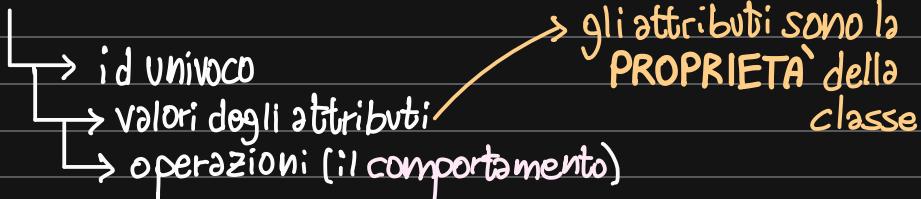


## DIAGRAMMI DELLE CLASSI SOFTWARE

Illustra le classi, interfacce e relative associazioni delle classi statiche (sw, statiche)

⇒ come è composto?

\* OGGETTO: istanza di una classe



\* CLASSIFICATORE: descrive caratteristiche comportamentali e strutturali (classi, interfacce, attori...)

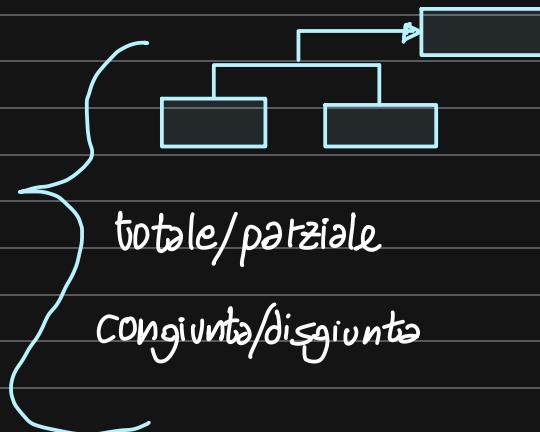
\* PAROLA CHIAVE : decoratore testuale per classificare un elemento

→ <<actor>>, <<interface>>  
& abstract, & ordered

## RIPASSO UML!

- - - - - → : dipendenza

— → : generalizzazione



## INTERFACCIA

Insieme di funzioni pubbliche che separano funzionalità e implementazione

↳ Ogni interfaccia ha un CONTRATTO e chi la usa deve rispettarlo



rappresentazione a pallino

→ non strettamente necessari in UP

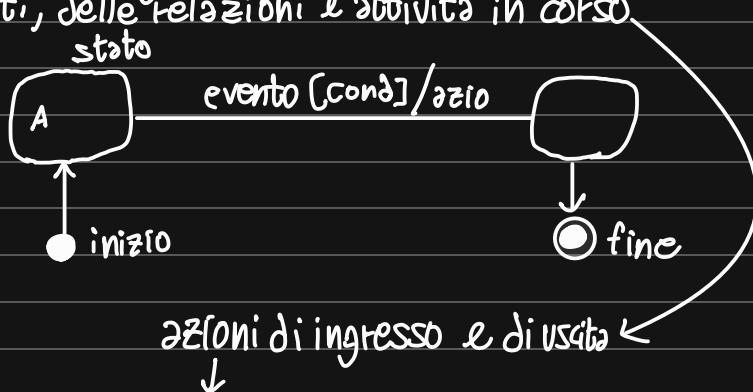
## MACCHINE A STATI

Diagrammi che modellano il comportamento dinamico di classificatori

↳ in pratica, si modella il **ciclo di vita** di un oggetto tramite una successione di stati, transizioni ed eventi

Gli oggetti quindi possono essere **DIPENDENTI** dallo stato in cui si trovano o **INDIPENDENTI**

Lo **STATO** di un oggetto non è altri che la combinazione di valori degli attributi, delle relazioni e attività in corso.



azioni di ingresso e di uscita  
oltre a ciò avvengono transizioni e attività interne in quel determinato stato dell'oggetto

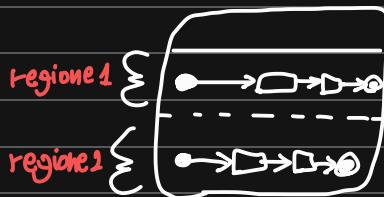
Gi sono stati più complessi: **COMPOSITI** e **PSEUDOSTATI**

**COMPOSITI**: stati che includono una o più regioni che hanno all'interno sottomacchine

se tutte le regioni hanno  
lo stato finale, la terminazione è  
sincrona

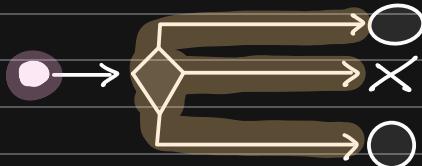
- Raggiunto lo stato finale di una regione, termina solo quella regione
- Raggiunto lo stato finale del composito, termina tutto lo stato composito

Le regioni possono comunicare  
tramite flag o synch



## PSEUDOSTATI

- **GIUNZIONE**: ricongiunge più archi nello stesso stato



- **SELEZIONE**: uno stato con solo input e vari output esclusivi,  
ciascuno con una condizione da rispettare
- **DI INGRESSO**: l'entrata di uno stato composito



- **USCITA**: punto di uscita per uno stato composito

- **MEMORIA**:

→ **SEMPLICE** · ricorda l'ultimo sotto stato del  
composito

→ **MULTILIVELLO**: come lo stato semplice, ma può ricordare anche gli strati

- **FINALE**: stato finale per gli stati compositi.

## EVENTI

- di chiamata: chiamate per specifiche operazioni o una sequenza
- di segnale: comporta l'invio o l'attesa della ricezione di un segnale
- temporale: verificano condizioni temporali, per esempio un dopo o un quando

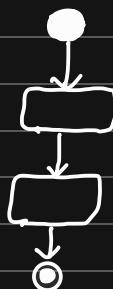
## DIAGRAMMA DELLE ATTIVITÀ

Mostra le attività di un processo, utili per visualizzare i flussi di esso tramite nodi e collegamenti

### Non obbligatori in UP

↳ utili per visualizzare il workflow di un processo/oggetto o dati

I nodi possono trasmettersi tra loro dei token, che possono essere oggetti/dati/fluxo di controllo



Ci sono vari tipi di nodi:

\* **AZIONE**: invocano attività, comportamenti o operazioni

\* **CONTROLLO**: controllano il flusso (inizio, fine, decisione...)

\* **OGGETTO**: rappresentano pile/code di token

Si possono raggruppare/suddividere i nodi tramite **PARTIZIONI**

## PROGETTAZIONE GUIDATA DALLE RESPONSABILITÀ

La responsabilità è un'astrazione di quello che un oggetto fa o rappresenta

↳ progettare avendo come guida la **RESPONSABILITÀ (RDD)** significa vedere il progetto come una comunità di oggetti responsabili e che collaborano per fornire le funzionalità

I metodi di questi oggetti sono implementati per adempiere alle responsabilità

Gli step sono: IDENTIFICARE → A QUALE OGGETTO  
RESPONSABILITÀ VA ASSEGNAТА?

→ COME FA QUESTO OGGETTO  
A SODDISFARLA?

Per scegliere come assegnare le responsabilità, ci si appella ai Pattern GRASP.

## PATTERN GRASP

Descrivono i principi base di progettazione e responsabilità degli oggetti.

non a caso ; PATTERN sono coppie di problemi/soluzione ben conosciuti e identificate da un NOME

Sono 9.

Grasp - 1

**INFORMATION** : pattern che indica di assegnare le responsabilità ad una classe con le info necessarie al compito

↳ se un oggetto ha i dati per un calcolo, il calcolo lo fa lui

Grasp - 2

**CREATOR** : assegna le responsabilità di creare un'istanza di una classe alla classe che detiene le informazioni per inizializzarla o che la usa molto

Grasp - 3

## CONTROLLER

: si crea un oggetto Controller per gestire gli eventi di sistema e coordinare le risposte alle richieste degli utenti:  
↳ gestisce le richieste in entrata e le instrada ai gestori appropriati

Grasp - 4

## LOW COUPLING

: si mantiene basso l'accoppiamento tra le classi per aumentare flessibilità e manutenibilità  
↓  
quindi si evita la dipendenza stretta tra classi reciproche

Grasp - 5

## HIGH COHESION

: si mantiene alta la coesione dentro le classi per tener chiare e precise le responsabilità  
( cercare di aver solo le funzioni essenziali alla resp.)

Grasp - 6

POLIMORFISMO : si assegnano le responsabilità alle classi derivate piuttosto che alle classi base  
↓  
in questo modo le derivate gestiscono da sole le variazioni provocate da loro al comportamento base

Grasp - 7

## PURE FABRICATION

: si crea una classe artificiale per mantenere il low coupling e la high cohesion (e ha lei le resp.)  
↳ la classe non è un concetto del dominio

Grasp - 8

**INDIRECTION** : si introduce un intermediario per ridurre l'accoppiamento diretto tra due classi o componenti

Grasp - 9

**PROTECTED VARIATIONS** : si proteggono le parti di un sistema dalle variazioni tramite astrazioni e encapsulamento



Si modella un'interfaccia stabile

## DESIGN PATTERN

Soluzioni generali a problemi comuni sw, a differenza dei GRASP mostrano anche lo schema delle classi e non sono solo consigli

Ogni pattern ha CLASSE e SCOPO.

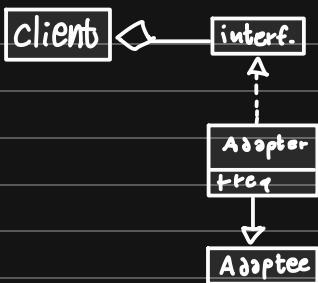
**Adapter ~ scopo** : STRUTTURALE

Consente di convertire l'interfaccia di una classe in un'altra interfaccia che si aspetta il client (per renderle compatibili tra loro)

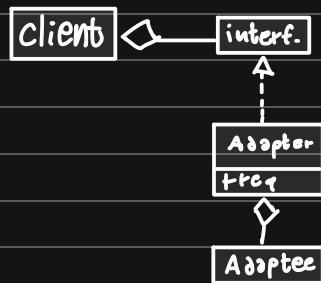
↓  
l'altro componente

→ si fa tramite un ADATTATORE : può essere una classe o un oggetto

## Classe



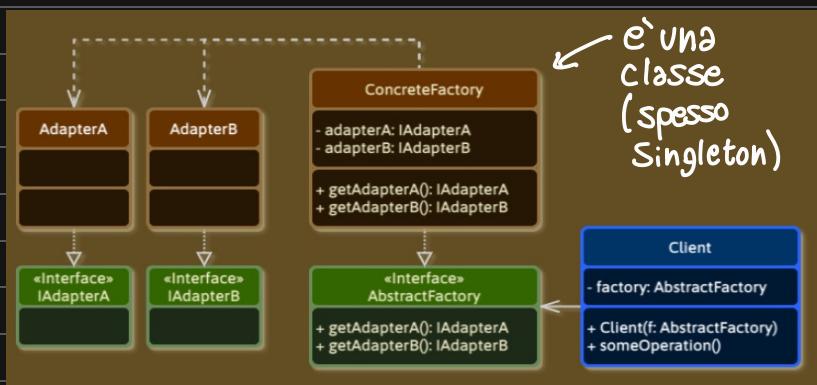
## Oggetto



## FACTORY ~ SCOPO: CREAZIONALE

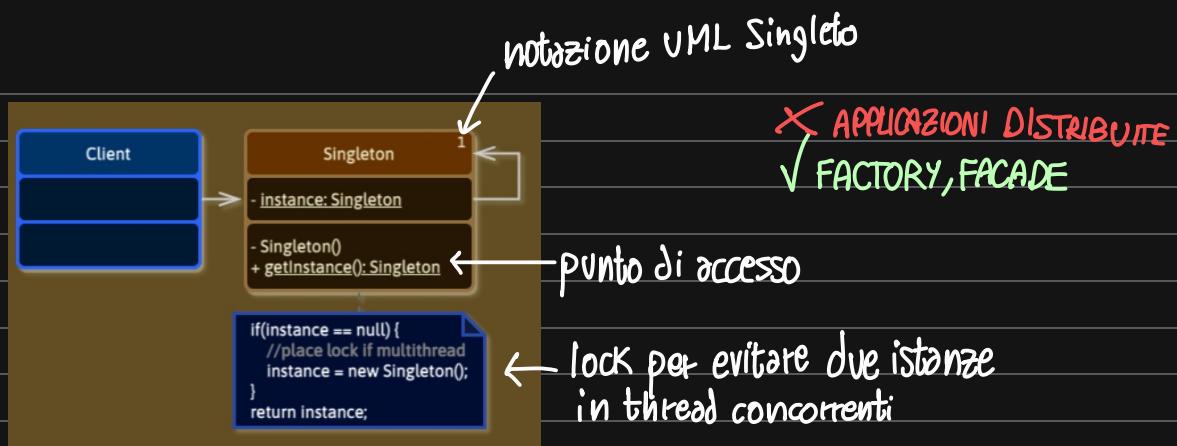
Si crea un oggetto che gestisce la creazione di oggetti quando ci sono considerazioni speciali come una logica complesso, separazione delle responsabilità... (per esempio, per gestire adapter)

→ l'oggetto è un Pure Fabrication chiamato Factory



## SINGLETON ~ SCOPO: CREAZIONALE

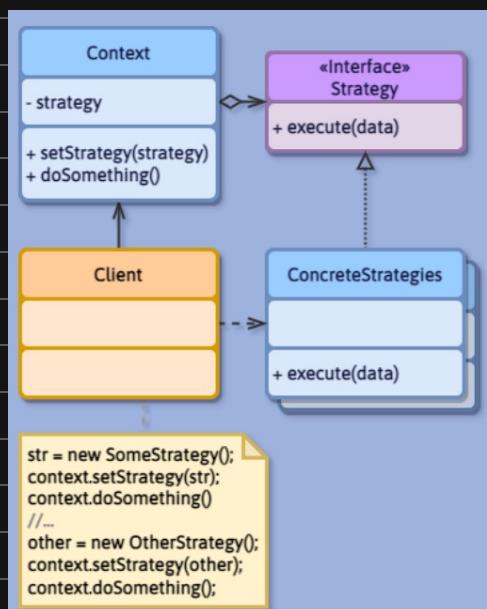
Garantisce che esista una sola istanza di una determinata classe e che esista un solo e unico punto d'accesso globale ad esso



esempio: un Logger che regista gli eventi di varie parti di app

## STRATEGY ~ SCOPO: Comportamentale

Si definiscono algoritmi/politica/strategie in una classe separata ciascuno con un'interfaccia comune



→ Context mantiene riferimento ad una delle strategie concrete e comunica con essa tramite l'interfaccia (comune a tutte le strategie)

↳ indica le classi che hanno variazioni di algoritmo

Context quindi chiama il metodo sull'oggetto strategy ogni volta che ne ha bisogno, non conosce l'esecuzione dell'algoritmo

E' il client a fornire l'oggetto strategy al context.

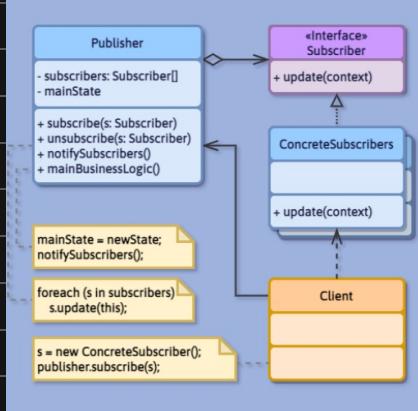
Strategy ha basi nel Polimorfismo

↳ fornisce Protected Variations (rispetto agli algoritmi)  
↳ spesso create da una Factory

## OBSERVER ~ SCOPO: comportamentale

Quando è necessario notificare gli oggetti subscriber di cambiamenti di stato o agli eventi di un oggetto publisher si definisce un'interfaccia Subscriber o listener da implementare negli oggetti subscriber

↳ Il publisher registra dinamicamente i subscriber interessati e li notifica tramite l'interfaccia → i subscriber reagiscono in modo indipendente agli eventi



Il client è responsabile della creazione e aggiornamento di publisher e subscriber

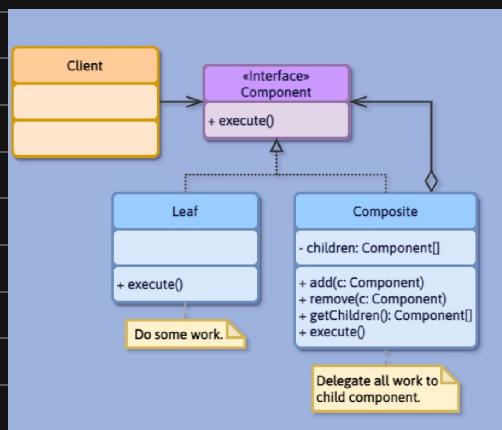
L'accoppiamento è debole in quanto i publisher conoscono i subscriber tramite interfaccia e i subscriber possono cancellarsi quando vogliono dal publisher  
o registrare

Observer si basa su Polimorfismo e fornisce Protected variations

→ i subscriber possono esser modificati liberamente

## COMPOSITE ~ SCOPO: STRUTTURALE

Quando si vuole trattare un gruppo o una struttura composta di oggetti come fossero un unico oggetto (atomico). Si definiscono classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia.



\* **COMPONENT** descrive le operazioni comuni a gli elementi del composto

\* Il **COMPOSITE** delega il lavoro alle foglie, raccoglie e processa i risultati intermedi e infine restituisce al client

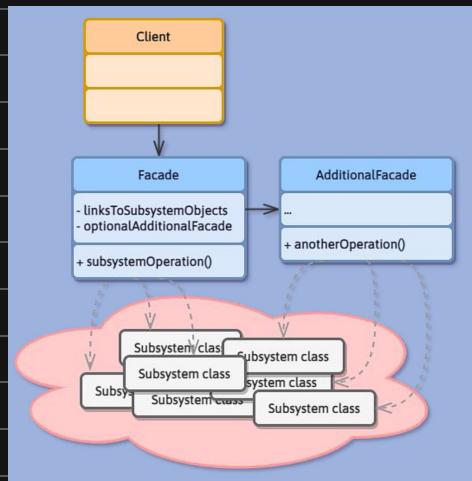
↳ La delega (o meglio, comunica) tramite l'interfaccia

Si usa con **STRATEGY**, si basa sul **POLIMORFISMO** e fornisce **PROTECTED VA**

## FAÇADE

Si vuole un'interfaccia comune e unificata per un insieme grande di implementazioni o interfacce (praticamente un sottosistema)

↳ L'oggetto **FAÇADE** (facciata) copre il sottosistema e funge da punto di contatto singolo, è incaricato della collaborazione tra i componenti



~ Solitamente un **SINGLETON**  
✓ **PROTECTED VARIATIONS** (implementazioni protette)  
✓ L'oggetto **INDIRECTION**  
sostiene **LOW COUPLING**

esempio: Facade controller

## PROGETTARE LA VISIBILITÀ

La visibilità è la capacità di un oggetto di vedere o aver un riferimento ad un altro oggetto  
↓

In caso di comunicazione  $A \rightarrow B$ , A deve avere il riferimento a B

si ottiene in quattro modi:

- **PER ATTRIBUTO**: B è attributo di A

- **PER PARAMETRO**: B è un parametro di un metodo di A

- **LOCALE**: B è un oggetto locale di un metodo di A

- **GLOBALE**: B è visibile globalmente

Le visibilità si classificano in PERMANENTI e TEMPORANEE

## DALLA PROGETTAZIONE ALLA IMPLEMENTAZIONE

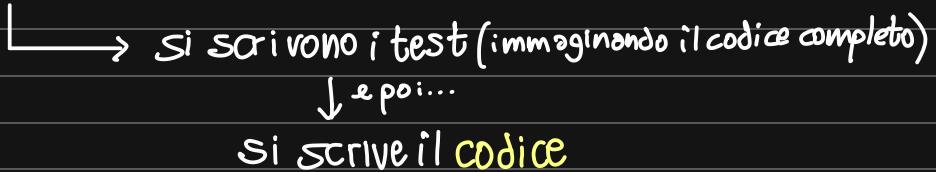
Si usano gli elaborati della progettazione come input per la generazione del codice OO (quindi classi, interfacce, variabili, metodi, costr.)

- \* Si implementa dalla meno accoppiata (indipendente) alla più accoppiata
- \* Le eccezioni in UML si indicano sui messaggi o nelle operazioni
- \* L'interfaccia viene implementata direttamente su variabile d'istanza

## TESTING

### TTD

Lo sviluppo guidato dal test (TTD) è una best practice usabile in UP



- Scrivere prima i test aiuta a chiarire lo scopo del programma
- Rende chiara l'interfaccia e comportamento
- Verifica veloce e automatica
- I cambi son fatti con più consapevolezza

Tipi di test:

- \* di **UNITÀ**: testa singole classi e metodi per verificare a unità il sistema
- \* di **INTEGRAZIONE**: verifica la comunicazione tra varie parti
- \* di **SISTEMA**: per verificare che il collegamento complessivo del sistema funzioni
- \* di **ACCETTAZIONE**: per verificare a scatola nera, praticamente dal punto di vista dell'utente

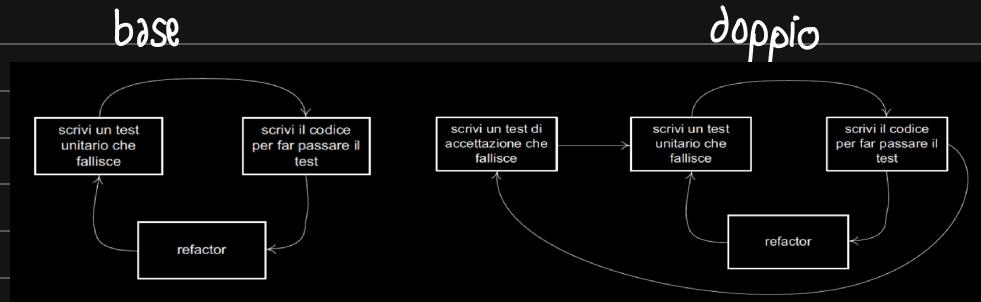


- \* Si crea l'oggetto da verificare e preparano altri oggetti/risorse
- \* Si eseguono le operazioni su di esso
- \* Si valutano i risultati ottenuti
- \* Si rilasciano eventuali altri test per pulire i test successivi

In generale, il ciclo funziona così:

- \* SCRIVERE UN TEST UNITARIO CHE FALLISCE per dimostrare che manca una funzione
- \* SCRIVERE CODICE SEMPLICE per validare i test
- \* RISCRIVERE o FARE REFACTOR per migliorare il codice e passare al prossimo test

Il ciclo può essere **base** o **doppio**:



Test di lavorazione brevi  
relativi ai test unitari, il ciclo più  
esterno è quello dei test di  
accettazione  
↳ questo potrebbe essere un  
intero test di un caso d'uso

## REFACTORING

Un metodo strutturato e disciplinato, utilizzato per riscrivere/ristrutturare del codice esistente **senza cambiare il comportamento esterno**



va avanti a piccoli passi e ripetendo i test ad ogni passo

- ✓ si migliora il codice dopo la scrittura
- ✓ predisponde il codice a nuove modifiche e cambiamenti

Inoltre, aiuta a eliminare codice DUPLICATO e migliorare la CHIAROZZA (e compatta codici lunghi!)

Si, ma refactoring quando? Diversi criteri:

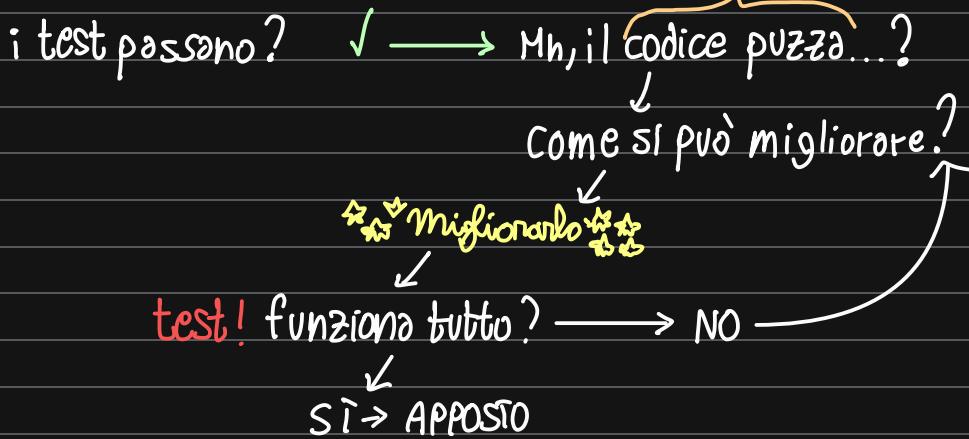
\* **Regola del tre**: se una porzione viene rivisitata almeno 3 volte nel programma

\* Quando si **AbbiUNBe** una funzionalità

\* Quando si corregge un **BVG**

\* Quando è in corso la **revisione** del codice

## PASSI DEL REFACTORING



**CODE SMELL**: caratteristiche del codice che denotano cattive pratiche di programmazione  
↓  
si suddividono in Categorie

\* **BLOATERS**: codici di dimensione sproporzionata

◦ Long Method: troppe linee di codice

per la spiegazione dei Refactor vai avanti!

- sottocasi:
- ~ posso ridurre la lunghezza senza problemi → *Extract method*
  - ~ ci sono dei cicli
  - ~ problemi con var locali →
    - Replace temp with query
    - Introduce param objects
  - ~ altrimenti → Replace Method with Obj → *Preserve Whole Obj*
  - ~ Se c'erano condizionali: *Decompose conditionals*

- Large Class: una classe ha troppi metodi
  - ~ si può separare in più classi → *Extract Class*
  - ~ si può implementare parte del comportamento in altri modi o è poco usato → *Extract Subclass*
  - ~ uno o più metodi si possono spostare coerentemente in altre classi → *Move method*
- Long parameter List: un metodo ha troppi parametri
  - ~ se alcuni parametri sono i return di altri metodi di altro oggetto → *Replace Parameter With Method call*
  - ~ se ci sono più parametri relativi ad un oggetto → *Preserve Whole Object*
  - ~ se i parametri non sono correlati → *Introduce Parameter Object*

## \* OO ABUSERS: implementazione errate o deboli della OO

- Switch statement: c'è uno switch molto lungo (o tanti if-else in cascata)

- ~ Si può sfruttare il polimorfismo → Replace cond. with Poli
  - ~ si può isolare lo switch → Extract, Move Method

- Refused Bequest: la sottoclasse non usa tutti i metodi; le proprietà ereditate, quindi la gerarchia non è corretta

- ~ se l'eredità non ha molto senso → Replace Inheritance with Delegation

- ~ l'eredità va bene, si possono togliere metodi dalla super → Extract superclass

## \* CHANGE Preventers: parti di codice altamente accoppiate che impediscono i rapidi cambiamenti

- Shotgun surgery: servono cambi piccoli ma in tante classi diverse

- ~ se si può riorganizzare il comportamento → Move Method raggruppando gli elementi in un'altra classe → Move Field

## \* DISPENSABLES: codice inutile da rimuovere/spostare altrove

- Duplicated Code: lo stesso codice appare in molti luoghi

- ~ ha senso rendere il codice un → Extract Method metodo chiamabile

◦ Data class: una classe che è solo un contenitore per i dati

~ si possono integrare nella Data Class metodi per manipolare dati → Move method

◦ Comments: troppi commenti

~ il commento...

... spiega un'espressione lunga → Extract variable

... spiega una porzione di codice → Extract method

... afferma regole necessarie → Introduce al funzionamento assertion

~ il metodo è già estratto ma molto → Rename Method  
commentato

\* COUPLERS: hanno eccessivi accoppiamenti o troppe deleghe

◦ Feature envy: quando un metodo accede di più ai dati di un altro che ai propri

~ il metodo deve star chiaramente altrove → Move

~ se solo una parte accede ai dati → Extract

~ il metodo usa tante funzioni di tante classi, quindi → Move, extract  
si può "smembrare" e suddividere

# REFACTORING: Caso e soluzioni

## x Extract

- **Variable**: uno o più variabili vengono assegnate ad un'istruzione complessa

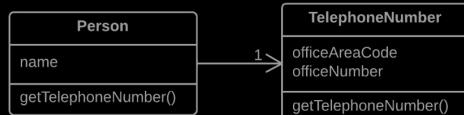
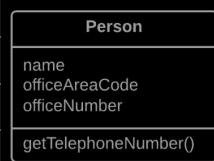
ESEMPIO: un if con tante condizioni lunghe

`if (xy==xx and xy==yy and ...)`  $\Rightarrow$    
 `bool x = xy==xx`  
`bool y ...`  
`if (x and y and ...)`

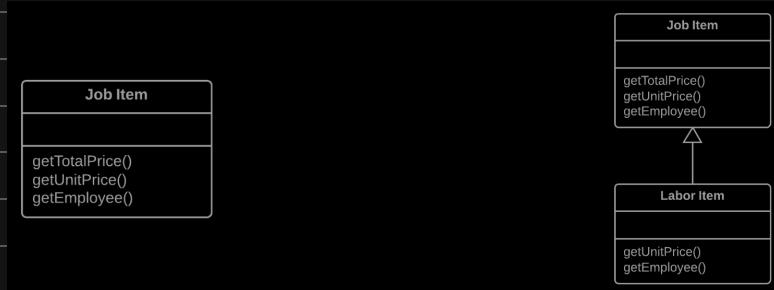
- **Method**: si crea un nuovo metodo  
estraendo codice da uno già esistente

`~() {`  
  `istr1`  
  `istr2`  
  `...`  
`}`               $\Rightarrow$       `~() {`  
  `:`  
  `3`  
  `~() {`  
  `istr1`  
  `istr2`  
`3`

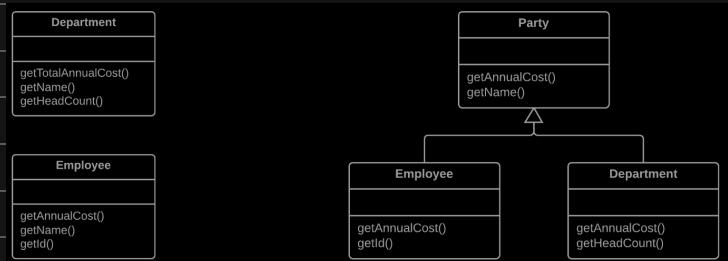
- **Class**: si estrae una parte della logica o del comportamento dalla classe per metterlo in una nuova



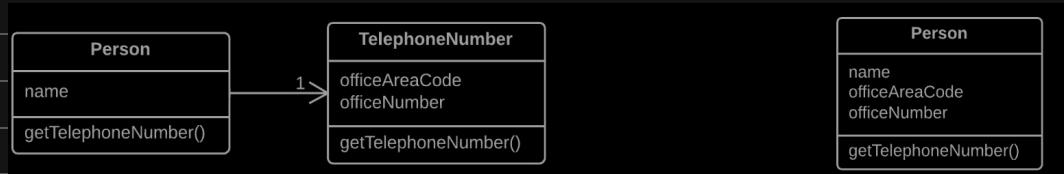
- **Subclass**: si estraе parte del comportamento e campi usati; raramente o possono esser implementati diversamente



- **Superclass**: si estraгono le parti comuni a più classi e si raggruppano in una nuova superclasse



✗ **Inline Class**: si incorpora una classe quasi vuota in una che la sfrutta



✗ **Move method**: spostamento metodo da una classe all'altra  
✗ **Rename method**: rinomina metodo

x **Preserve Whole Object**: si passa direttamente il metodo come parametro al posto dei suoi attributi

**Prima**

```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

**Dopo**

```
boolean withinPlan = plan.withinRange(daysTempRange);
```

x **Replace Temp with Query**: Si rimpiazza una variabile temporanea con una chiamata a un metodo

**Prima**

```
double calculateTotal() {
    double basePrice = quantity * itemPrice;
    if (basePrice > 1000) {
        return basePrice * 0.95;
    }
    else {
        return basePrice * 0.98;
    }
}
```

**Dopo**

```
double calculateTotal() {
    if (basePrice() > 1000) {
        return basePrice() * 0.95;
    }
    else {
        return basePrice() * 0.98;
    }
}
double basePrice() {
    return quantity * itemPrice;
}
```

→ variabile temporanea

← metodo che rispecchia la variabile

x **Replace Parameter with method call**: si toglie un parametro dal metodo A derivante da una call di B e lo sostituisce con chiamata a B da A

**Prima**

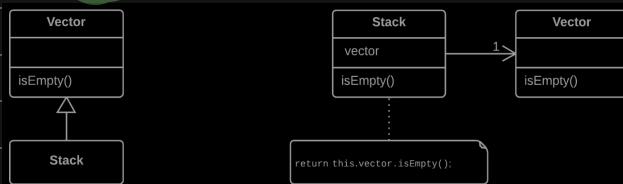
```
int basePrice = quantity * itemPrice;
double seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice = discountedPrice(basePrice, seasonDiscount,
fees);
```

**Dopo**

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

Replace Method with Method Object: si crea un nuovo oggetto e si rimpiazza un metodo con un nuovo metodo appartenente all'oggetto

Replace Inherit with Delegation: si sostituisce l'eredità con una relazione di delega



Replace conditional with Polymorphism: si rimpiazza un costrutto condizionale dividendo i casi in sottoclassi diverse

```

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");
    }
}

Dopo

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();
  
```

Decompose Conditional: si scompongono le parti complicate del blocco condizionale in metodi separati

× Introduce Param Obj : dei parametri vengono raggruppati in un oggetto

amount(start:date, end:date) ⇒ amount(Daterange)

× Introduce assertion: si introduce un'asserzione per esprimere la necessità di una condizione al proseguimento

#### Prima

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

#### Dopo

```
double getExpenseLimit() {  
    Assert.isTrue(expenseLimit != NULL_EXPENSE || primaryProject !=  
    null);  
  
    return (expenseLimit != NULL_EXPENSE) ?  
        expenseLimit:  
        primaryProject.getMemberExpenseLimit();  
}
```