INGEGNERIA DEL SOFTWARE

Docenti

Andrea Polini

Andrea Morichetta

Studente

Giorgio Della Roscia

Università degli studi di Camerino

Informatica per la Comunicazione Digitale (L-31)

Scuola di Scienze e Tecnologie

A.A. 2024/2025

Indice

1 PROCESSI DI SVILUPPO **1**
- 1.1 Prospettive del processo . . . 2
- 1.2 Ciclo di vita . . . 2
  - 1.2.1 Modello a cascata . . . 2
  - 1.2.2 Modello evolutivo . . . 3
  - 1.2.3 Modello iterativo . . . 3
    - 1.2.3.1 UP . . . 3
      - 1.2.3.1.1 Iterazioni . . . 4
      - 1.2.3.1.2 Struttura dell'UP . . . 4
- 1.3 Requisiti . . . 5
  - 1.3.1 Specifica dei requisiti . . . 5
  - 1.3.2 Processo di ingegneria dei requisiti . . . 6
    - 1.3.2.1 Elicitazione . . . 7
      - 1.3.2.1.1 Interviste . . . 8
      - 1.3.2.1.2 Workshop . . . 8
      - 1.3.2.1.3 KJ method . . . 9
    - 1.3.2.2 Formato di definizione dei requisiti . . . 9
  - 1.3.3 Requisiti qualitativi . . . 10
    - 1.3.3.1 Metriche di prodotto . . . 10
      - 1.3.3.1.1 Correttezza funzionale . . . 10
      - 1.3.3.1.2 Affidabilità . . . 11
      - 1.3.3.1.3 Robustezza . . . 11
      - 1.3.3.1.4 Efficienza . . . 11
      - 1.3.3.1.5 Usabilità . . . 12
      - 1.3.3.1.6 Verificabilità . . . 12
      - 1.3.3.1.7 Manutenibilità . . . 12
      - 1.3.3.1.8 Riusabilità . . . 12
      - 1.3.3.1.9 Portabilità . . . 12
      - 1.3.3.1.10 Comprensibilità . . . 12
      - 1.3.3.1.11 Interoperabilità . . . 12
    - 1.3.3.2 Metriche del processo di sviluppo . . . 12
      - 1.3.3.2.1 Produttività . . . 12
      - 1.3.3.2.2 Timeliness . . . 13
      - 1.3.3.2.3 Visibilità . . . 13
- 1.4 GRASP . . . 13
  - 1.4.1 Creator . . . 13
  - 1.4.2 Information expert . . . 13
  - 1.4.3 Low coupling . . . 13
  - 1.4.4 High cohesion . . . 14
  - 1.4.5 Controller . . . 14
  - 1.4.6 Pure fabrication . . . 14
  - 1.4.7 Polymorfism . . . 14
  - 1.4.8 Indirection . . . 14
  - 1.4.9 Protected variants . . . 15
- 1.5 Verifica e validazione . . . 15

1.5.1 Testing 15

2 UML **18**
2.1 Use case diagram 19
2.1.1 Attori 21
2.1.1.1 Generalizzazione degli attori 21
2.2 Class diagram 22
2.2.1 Classi 22
2.2.1.1 Classi e stato del progetto 23
2.2.2 Relazioni 24
2.2.2.1 Associazioni 24
2.2.2.2 Dipendenze 25
2.2.2.3 Generalizzazioni 25
2.2.2.3.1 Processi di modellazione 25
2.2.2.3.2 Insieme di generalizzazione 26
2.2.2.3.3 Polimorfismo 26
2.2.2.4 Progettazione 27
2.3 Interaction diagram 27
2.3.1 Sequence diagram 27
2.4 Petri nets 28
2.4.1 Contesa dei token 29
2.4.2 Estensione delle reti di Petri 30
2.4.3 Diagrammi di attività 30
2.5 Component diagram 31
2.6 State chart diagram 32

3 OOD **35**
3.1 OOP 36
3.1.1 Relazioni tra oggetti 36
3.2 Software qualitativo 36
3.2.1 Isolamento dei cambiamenti 37
3.2.1.1 Composizione rispetto ereditarietà 37
3.2.2 Principi di OOD 38
3.3 Design pattern 39
3.3.1 Pattern creazionali 40
3.3.1.1 Factory method 40
3.3.1.2 Abstract factory 41
3.3.1.3 Builder 42
3.3.1.4 Prototype 43
3.3.1.5 Singleton 44
3.3.2 Pattern strutturali 45
3.3.2.1 Adapter 45
3.3.2.2 Bridge 46
3.3.2.3 Composite 47
3.3.2.4 Decorator 48
3.3.2.5 Facade 49
3.3.2.6 Flyweight 50
3.3.2.7 Proxy 51
3.3.3 Pattern comportamentali 52
3.3.3.1 Chain of responsibility 52
3.3.3.2 Command 53
3.3.3.3 Iterator 54
3.3.3.4 Mediator 55
3.3.3.5 Memento 56
3.3.3.6 Observer 57
3.3.3.7 State 58

3.3.3.8 Strategy 59
3.3.3.9 Template method 60
3.3.3.10 Visitor 61

Sitografia

Bibliografia

Figure

1.1 Itinerario sviluppo software . . . 1
1.2 Diagramma modello a cascata . . . 2
1.3 Diagramma modello iterativo . . . 3
1.4 Fasi dell'Unified Process . . . 4
1.5 Tipologie di requisiti . . . 5
1.6 Classificazione dei requisiti qualitativi . . . 10

2.1 Gerarchia diagrammi UML . . . 18
2.2 4+1 views of software development . . . 19
2.3 Rappresentazione use case diagram . . . 19
2.4 Success and exception scenarios . . . 20
2.5 Attore . . . 21
2.6 Rappresentazione UML di un oggetto . . . 22
2.7 Rappresentazione UML di una classe . . . 22
2.8 Reciprocità generalizzazione e specializzazione . . . 25
2.9 Coerenza orizzontale delle sottoclassi . . . 25
2.10 Ereditarietà a diamante . . . 26
2.11 Tipologie di messaggi dell'integration diagram . . . 27
2.12 SSD e SD a confronto . . . 27
2.13 Notazione grafica delle reti di Petri . . . 28
2.14 Circuito con nodi di controllo finali . . . 31

3.1 Diagramma UML di una classe . . . 35
3.2 Generalizzazione classi e interfacce . . . 35
3.3 Relazioni tra oggetti . . . 36
3.4 Introduzione di astrazioni . . . 37
3.5 Logistica con factory method . . . 40
3.6 Struttura del factory method . . . 40
3.7 Arredamento con abstract factory . . . 41
3.8 Struttura dell'abstract factory . . . 41
3.9 Costruzione case con builder . . . 42
3.10 Struttura del builder . . . 42
3.11 Configurazione cloni con prototype . . . 43
3.12 Struttura del prototype . . . 43
3.13 Costruzione case con singleton . . . 44
3.14 Struttura del singleton . . . 44
3.15 Conversione XML in JSON con adapter . . . 45
3.16 Struttura dell'adapter . . . 45
3.17 Colorare le forme con bridge . . . 46
3.18 Struttura del bridge . . . 46
3.19 Calcolare il prezzo totale con composite . . . 47
3.20 Struttura del composite . . . 47
3.21 Ricezione notifiche con decorator . . . 48
3.22 Struttura del decorator . . . 48
3.23 Centro assistenza con facade . . . 49
3.24 Struttura del facade . . . 49
3.25 Sviluppo videogioco con flyweight . . . 50

3.26 Struttura del flyweight 50
3.27 Delegazione lavoro con proxy 51
3.28 Struttura del proxy 51
3.29 Gestione ecommerce con chain of responsibility 52
3.30 Struttura del chain of responsibility 52
3.31 Definizione bottoni con command 53
3.32 Struttura del command 53
3.33 Scorrere un albero con iterator 54
3.34 Struttura dell'iterator 54
3.35 Finestre di dialogo con mediator 55
3.36 Struttura del mediator 55
3.37 Undo nei text editor con memento 56
3.38 Struttura del memento 56
3.39 Newsletter mirata con observer 57
3.40 Struttura del observer 57
3.41 Pubblicazione documento con state 58
3.42 Struttura del state 58
3.43 Calcolo percorso con strategy 59
3.44 Struttura del strategy 59
3.45 Analisi dati con template method 60
3.46 Struttura del template method 60
3.47 Esportazione mappa con visitor 61
3.48 Struttura del visitor 61

Tabelle

1.1 Modalità di specifica dei requisiti 5
1.2 Requisiti stabili e volatili 7
1.3 Formato VOLERE© 9

2.1 Livelli di accesso dei modificatori 23
2.2 Indicatori di molteplicità 24
2.3 Indicatori di navigabilità 24
2.4 Comparazione ereditarietà e composizione 26
2.5 Notazione grafica nodi di controllo 31

3.1 Categorizzazione di design pattern 39

Codici

1.1 Libreria matematica interi 10
1.2 Libreria matematica decimali 10
1.3 Libreria matematica interi 11
1.4 Libreria matematica decimali 11
1.5 Libreria matematica interi 11
1.6 Libreria matematica decimali 11
1.7 Linux pipe 12

Approfondimenti

Definizione (software) . . . 1
Definizione (ingegneria del software) . . . 1
Nota bene (waterfall sequence) . . . 2
Nota bene (terminamento dei task) . . . 3
Definizione (incremento) . . . 4
Definizione (fase) . . . 4
Definizione (milestone) . . . 4
Nota bene (UP ≠ waterfall) . . . 4
Definizione (requisito) . . . 5
Nota bene (multiuso delle tecniche di specifica dei requisiti) . . . 5
Esempio (ambiguità lessicale) . . . 6
Esempio (ambiguità sintattica) . . . 6
Esempio (ambiguità semantica) . . . 6
Esempio (ambiguità pragmatica) . . . 6
Principio (Paredo) . . . 7
Nota bene (meccanismo di generalizzazione) . . . 7
Principio (MoSCoW) . . . 9
Definizione (use case) . . . 9
Definizione (qualità) . . . 10
Definizione (metrica) . . . 10
Esempio (correttezza funzionale) . . . 10
Esempio (affidabilità) . . . 11
Esempio (robustezza) . . . 11
Esempio (JVM) . . . 12
Esempio (Linux pipe) . . . 12
Nota bene (GRASP controller ≠ MVC controller) . . . 14
Definizione (testing) . . . 15
Nota bene (testing ≠ debugging) . . . 15
Definizione (rivalidazione) . . . 15
Problema (dell'oracolo) . . . 16

Definizione (oggetto) . . . 18
Definizione (architettura) . . . 19
Definizione (caso d'uso) . . . 19
Nota bene (uso dei sinonimi) . . . 20
Nota bene (sequenza come dialogo) . . . 20
Definizione (condizioni) . . . 20
Definizione (pre-condizioni) . . . 20
Definizione (post-condizioni) . . . 20
Nota bene (attore ≠ entità) . . . 21
Definizione (oggetto) . . . 22
Definizione (identità) . . . 22
Definizione (classe) . . . 22
Definizione (dipendenza) . . . 23
Definizione (collegamento) . . . 24
Definizione (associazione) . . . 24

Definizione (dipendenza) 25
Definizione (insieme di generalizzazione) 26
Definizione (polimorfismo) 26
Problema (diamante) 26
Principio (Liskov substitution) 26
Definizione (piazza di input) 28
Definizione (piazza di output) 28
Definizione (scatto della transizione) 28
Esercizio (somma di interi) 29
Definizione (deadlock) 29
Esempio (contesa dei token) 29
Esercizio (meccanismo di fairness) 29
Esempio (differenza fra nodo finale dell'attività e del flusso) 31
Definizione (componente) 31
Definizione (stato) 32
Definizione (evento) 32
Definizione (transizione) 32
Esempio (SM di una lampadina) 33

Principio (open-closed) 38
Principio (dependency inversion) 38
Principio (Liskov substitution) 38
Principio (single responsibility) 38
Principio (interface segregation) 38
Definizione (design pattern) 39
Esempio (factory method) 40
Esempio (abstract factory) 41
Esempio (builder) 42
Esempio (prototype) 43
Esempio (singleton) 44
Esempio (adapter) 45
Esempio (bridge) 46
Esempio (composite) 47
Esempio (decorator) 48
Esempio (facade) 49
Esempio (flyweight) 50
Esempio (proxy) 51
Esempio (chain of responsibility) 52
Esempio (command) 53
Esempio (iterator) 54
Esempio (mediator) 55
Esempio (memento) 56
Esempio (observer) 57
Esempio (state) 58
Esempio (strategy) 59
Esempio (template method) 60
Esempio (visitor) 61

Sezione 1

PROCESSI DI SVILUPPO

> **Definizione** (software)**.** Programma che specifica le istruzioni che un calcolatore dovrà eseguire al fine di raggiungere uno scopo.

In questo contesto il termine identifica anche tutti i documenti che lo descrivono e che sono stati messi a punto durante le varie fasi della produzione del sistema.

> **Definizione** (ingegneria del software)**.** Disciplina che applica un approccio sistematico, disciplinato e quantificabile a sviluppo, supporto e mantenimento del software.

Il software si evolve e la sua dimensione e complessità con lui.

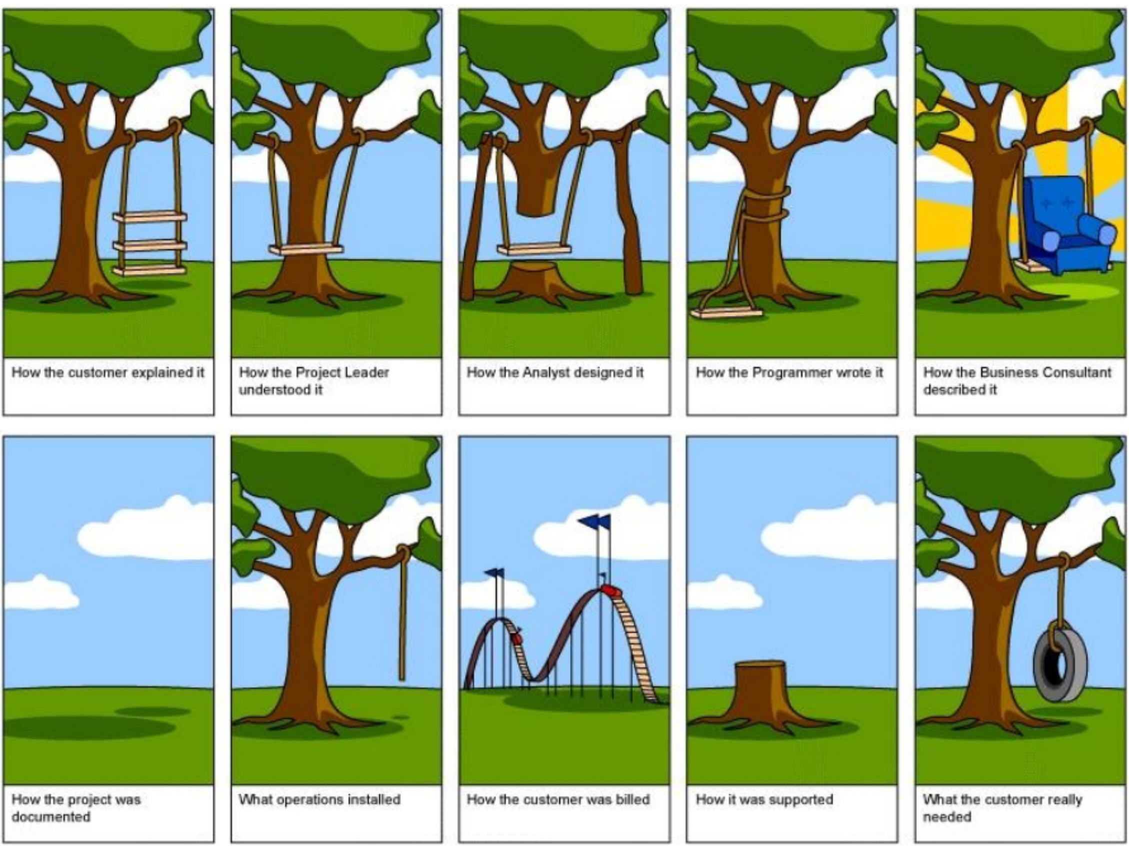


Figura 1.1: Itinerario sviluppo software

Date queste complicazioni, spesso il rilascio del software avviene con ritardo rispetto alle scadenze

prestabilite e la sua qualità è discutibile.

1.1 Prospettive del processo

Ogni proposta software deve tenere conto delle **4P**:

a. **P**roject, attività relative alla produzione;
b. **P**rocess, organizazzione delle attività;
c. **P**roduct, codebase e artefatti relativi;
d. **P**eople, stakeholder.

Il processo di sviluppo può essere osservato sotto diversi punti di vista:

a. **activity perspective**, definizione di cosa deve essere fatto;
b. **workflow perspective**, sequenza ed organizzazione delle attività;
c. **data-flow perspective**, flusso delle informazioni fra le varie attività;
d. **role/action perspective**, assegnazione dei compiti.

1.2 Ciclo di vita

Gli stati dell'entità, dal concepimento alla dismissione, vengono descritti secondo:

- comunicazione, definire i requisiti;
- pianficazione, valutare rischi e costi;
- modellazione, progettare le funzionalità;
- costruzione, implementare le scelte e verificarle;
- sviluppo, dispiegare nell'ambiente di produzione.

La successiva attività di manutenzione può essere: correttiva, adattiva o perfettiva.

1.2.1 Modello a cascata

Prende il nome dalla rappresentazione con diagramma di Gantt che va a creare una simil cascata.

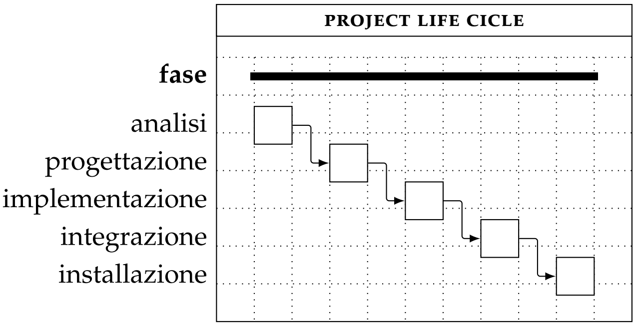


Figura 1.2: Diagramma modello a cascata

Nota bene (waterfall sequence)**.** Ogni elemento una volta completato non viene reiterato!

La prematura decisione dei requisiti rende il processo statico rispetto le successive richieste di adattamento; perciò si tratta di un modello applicabile a progetti con richieste precise.

i. Le tempistiche vengono rispettate, a volte andando a sacrificare il perfezionamento delle singole attività.

ii. La visibilità alta è data dalla comprensione dello stato del progetto in base allo stato di completamento.

1.2.2 Modello evolutivo

Lo sviluppo si incentra sulla derivazione di un prototipo da cui poi partire per incrementi successivi. A volte si costruiscono anche dei *throwaway prototypes*. Il sistema tenderà ad avere un'organizazzione poco strutturata dati i continui cambiamenti, ciò porterà a problematiche rilevanti in fase di mantenimento.

i. Le tempistiche sono indefinite data la durata dei requisiti pari a tutto il processo di sviluppo.

ii. La visibilità è scarsa poiché non viene seguita una direzione prestabilita.

1.2.3 Modello iterativo

Intermezzo fra waterfall ed evolutivo. Si incentra su rilasci incrementali che forniscono sistemi funzionanti ed ogni iterazione corrisponde ad un mini workflow.

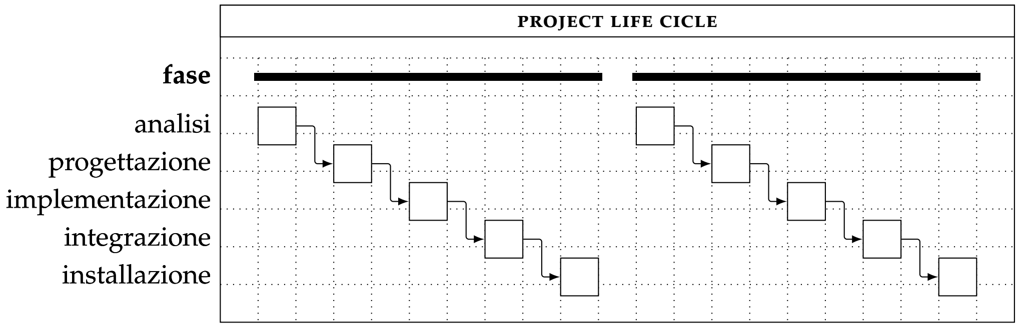


Figura 1.3: Diagramma modello iterativo

Ad gni rilascio viene distribuito un sistema funzionante con nuove funzionalità introdotte.

Nota bene (terminamento dei task)**.** Ogni iterazione avviata deve essere terminata.

Questo modello è efficiente su un team di lavoro ristretto e riduce il rischio di fallimento complessivo.

i. Le tempistiche sono instabili, ma ben osservate.

ii. La visibilità è alta grazie alla ciclicità delle operazioni.

1.2.3.1 UP

L'*Unified Process* è standardizzato per lo sviluppo di software e si associa all'uso di UML. Le quattro caratteristiche primarie riguardano l'essere:

1. **guidato dai casi d'uso**, le scelte intraprese vertono verso la minimizzazione del rischio di fallimento del progetto;
2. **guidato dal rischio**, si tende ad individuare i potenziali rischi in una fase iniziale per mitigarli aumentando così la percezione del contesto;
3. **incentrato sull'architettura**, tecnica per scoprire e documentare i requisiti tramite artefatti;
4. **iterativo incrementale**, sviluppo delle caratteristiche attraverso i diagrammi.

UP è un processo generico di sviluppo che va adattato per ogni progetto intrapreso. RUP (*Rational UP*) è un'istanza di UP ben definita, ma da adattare.

1.2.3.1.1 Iterazioni

Si hanno come dei mini progetti che includono le seguenti attività:

a. pianificazione;
b. requisiti;
c. progettazione;
d. implementazione;
e. integrazione e test;
f. validazione e rilascio.

Per favorire l'agilità, le iterazioni sono timeboxed: non prolungabili. Il flusso di lavoro nelle iterazioni non è diviso equamente: i workflow possono avere differente enfasi in base allo stato di avanzamento. Ogni iterazione genera un output, o baseline. Tale insieme di artefatti saranno il punto di partenza dell'iterazione successiva.

> **Definizione** (incremento)**.** Differenza tra le baseline generate a due iterazioni successive.

1.2.3.1.2 Struttura dell'UP

> **Definizione** (fase)**.** Periodo di tempo che viene usato per il raggiungimento di obiettivi.

> **Definizione** (milestone)**.** Insieme di obiettivi di una fase.

Il ciclo di vita dell'UP è composto da quattro fasi.

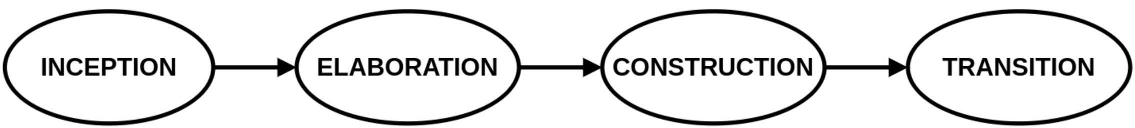


Figura 1.4: Fasi dell'Unified Process

> **Nota bene** (UP ≠ waterfall)**.** I flussi di lavoro sono distribuiti all'incirca sulla diagonale; ciò non implica però che si tratta di un modello a cascata!

1. **Avvio**
   Studio iniziale sulla fattibilità del progetto, dove si valuta la raggiungibilità tecnica, che si chiude con l'accettazione o il rifiuto.
2. **Elaborazione**
   Risoluzione dei rischi e piano dei casi d'uso da implementare in fase di costruzione. Termine estremo di accettazione o rifiuto del progetto.
3. **Costruzione**
   Soddisfacimento delle condizioni, che vedono un software stabile e qualitativo, e verifica dei costi rispetto la pianificazione. Si ha dunque l'accordo tra le parti per il rilascio.
4. **Transizione**
   Ultimato il beta testing si procede all'acceptance testing dove vengono coinvolti gli utenti finali per provare il sistema. Conclusione con il rilascio finale.

1.3 Requisiti

La parte più critica di ogni lavoro è a monte di ogni progettazione, si tratta infatti di definire assieme al cliente le richieste da soddisfare ed i vincoli da rispettare.

> **Definizione** (requisito)**.** Condizione o capacità da soddisfare per l'utente per risolvere un problema o raggiungere un obiettivo.

Il nostro interesse è relativo ai SIS (*Software Intensive Systems*), nei quali è evidente la difficoltà di una corretta comprensione del problema al fine di produrre un sistema adeguato:

- cliente non ha approvato i requisiti;
- visione e scopo non chiaramente definiti;
- clienti troppo impegnati per essere coinvolti;
- team di sviluppo ed utenti non interagiscono;
- sviluppatori incontrano ambiguità durante lo sviluppo;
- ambito del progetto aumenta al progredire dello sviluppo;
- requisiti dichiarati unitariamente come critici senza prioritizzazione;
- comunicazione focalizzata su rappresentazioni anziché funzionalità.

I requisiti vengono distinti a seconda del punto d'interesse durante l'analisi.

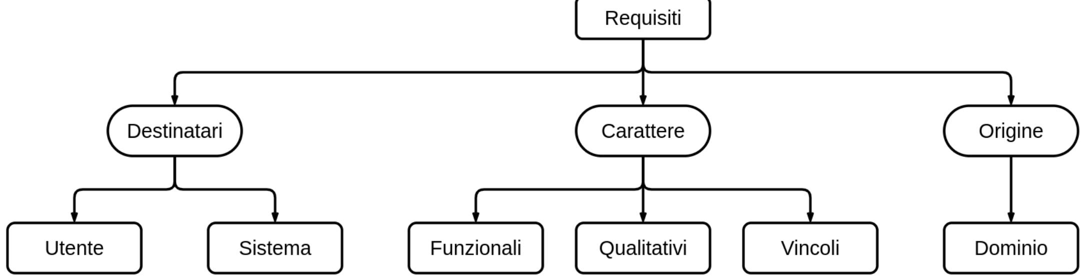


Figura 1.5: Tipologie di requisiti

I requisiti utente devono avere un alto livello d'astrazione per garantirne la comprensibilità; cosa agli antipodi nei requisiti di sistema, indubbiamente più tecnici e pragmatici.

1.3.1 Specifica dei requisiti

I requisiti possono essere riportati con diverse tecniche:

Tabella 1.1: Modalità di specifica dei requisiti

MANIERA	STRUTTURA
informale	linguaggio naturale
semiformale	notazione grafica
formale	modelli matematici

> **Nota bene** (multiuso delle tecniche di specifica dei requisiti)**.** Adoperare una tecnica non implica escludere le altre!

L'informalità risulta spesso ambigua per vari fattori:

a. **lessicale**, esistenza di termini polisemici;

> **Esempio** (ambiguità lessicale). Ecco a seguire una frase fraintendibile per via del suo lessico.
>
> *«Al di fuori di un cane, un libro è il miglior amico dell'uomo; da dentro è troppo difficile leggere»*
>
> «Al di fuori» viene inteso letteralmente per *fuori da*, anziché *escludendo*.

b. **sintattico**, frasi con più di un albero sintattico;

> **Esempio** (ambiguità sintattica). Ecco a seguire una frase fraintendibile per via della sua sintattica.
>
> *«Franco ha visto Luca in giardino con il canocchiale»*
>
> Franco ha utilizzato un canocchiale per vedere Luca oppure era Luca che aveva un canocchiale?

c. **semantico**, interpretazione multipla della frase nella logica dei predicati;

> **Esempio** (ambiguità semantica). Ecco a seguire una frase fraintendibile per via della sua semantica.
>
> *«Tutti i linguisti preferiscono una teoria»*
>
> La teoria preferita è la stessa, oppure ognuno ha la propria?

d. **pragmatico**, interpretazione dipendente dal contesto.

> **Esempio** (ambiguità pragmatica). Ecco a seguire una frase fraintendibile per via della sua pragmaticità.
>
> *«L'entrata è sulla destra»*
>
> Si parla rispetto al contesto in cui ci si trova.

1.3.2 Processo di ingegneria dei requisiti

Si individuano quattro parti:

a. **studio di fattibilità**, ci si chiede se convenga o meno continuare a sviluppare;

b. **richiesta ed analisi dei requisiti**, vengono individuati gli stakeholder a cui viene richiesto il problema (e non la soluzione!);

c. **validazione**, si cerca di rimuovere possibili problemi nella specifica dei requisiti. Le verifiche che possono essere effettuate sono:
   - **validità**, la specifica coincide con quanto necessario;
   - **consistenza**, i requisiti non si contraddicono;
   - **completezza**, i requisiti specificano tutte le funzionalità;
   - **concretezza**, coerenza fra la richiesta e le tecnologie, i costi e le scadenze;

- **verificabilità**, i requisiti devono poter essere soddisfatti ed accertati.

Oltre queste cinque tecniche si possono effettuare delle revisioni, prototipazioni e casi di test;

d. **gestione**, attività che si occupa di far emergere, permettere e gestire modifiche ai requisiti. Vengono usati strumenti appositi come una wiki o un DB.

> **Principio** (Paredo). Circa il 20% delle cause provoca l'80% degli effetti.

Applicato al nostro contesto, ciò tramuta nella definizione di due categorie di requisiti.

Tabella 1.2: Requisiti stabili e volatili

	stabili	**volatili**
QUANTITÀ	80%	20%
MODIFICHE	20%	80%

Le informazioni si immagazzinano in matrici di tracciabilità:

- **sorgente**, attore × requisito;
- **design**, sottosistema × requisito;
- **relazione con altri requisiti**, requisito × requisito.

1.3.2.1 Elicitazione

La seconda delle quattro fasi precedentemente descritte richiede un maggiore studio in comparazione alle altre. Dato che la comunicazione è diretta si incappa in problemi relativi al linguaggio naturale ed ai processi mentali: rimozione, distorsione, generalizzazione. . .

> **Nota bene** (meccanismo di generalizzazione). L'essere umano tende a raggruppare, tralasciando le ridotte eccezioni, utilizzando termini del tipo:
>
> tutto – ogni – sempre   niente – nessuno – mai

La scoperta dei requisiti avviene per tre istanze:

- **punto di vista**
  - **diretto**: chi interagisce direttamente con il sistema;
  - **indiretto**: chi non intergisce con il sistema ma è interessato al suo comportamento;
  - **di dominio**: attori esperti del dominio applicativo.
- **ruolo**: persone, sistemi esterni e sensori;
- **interfaccia per interagire col sistema**: HMI (*Human-Machine Interface*), software (API) e protocolli, e hardware;

Gli attori vengono classificati in:

* **primari**, utilizzano direttamente il sistema;
* **finali**, l'uso del sistema da parte di altri realizza i loro obiettivi;
* **di supporto**, offrono un servizio al sistema;
* **fuori scena**, sono interessati al comportamento del caso d'uso.

Per la scoperta dei requisiti si sfruttano delle tecniche specifiche, ognuna delle quali ha degli aspetti da definire: preparazione, esecuzione, follow-up, benefici, complessità e fattori di successo. Le più celeri da anticipare sono:

- **focus group**, per approfondire con un piccolo team tematiche del sistema poco chiari;
- **osservazione**, si scovano celatamente i processi dei potenziali utilizzatori del sistema nello svolgimento delle regolari mansioni;
- **questionari**, strutturamento di una lista di domande distribuita agli stakeholder per comprendere concetti specifici;
- **perspective-based reading**, identificazione e studio di documenti di rilievo per lo sviluppo.

Si adoperano inoltre delle tecniche accessorie:

- **brainstorming**, emersione di idee innovative;
- **KJ method**, brainstorming in contesti con partecipanti eterogenei;
- **prototyping**, concretizzazione pragmatica di alcuni aspetti del sistema;
- **mind mapping**, costruzione di mappe concettuali;
- **elicitation checklist**, lista di tipici aspetti in un particolare contesto applicativo.

Continuiamo infine ad esporre le ultime due tecniche principali tralasciate precedentemente.

1.3.2.1.1 Interviste

Metodo dispendioso ma verticale su un attore fondamentale. Ne esistono tre tipi:

- **standardizzate**, domande chiuse;
- **esplorative**, domande aperte;
- **destrutturate**, discussione guidata dall'intervistato.

Si divide in tre fasi:

1. la **preparazione** riguarda l'adocchiamento di un obiettivo, la selezione e l'invito dei partecipanti, la scelta del luogo e la definizione delle domande.
2. l'**esecuzione** si frammenta invece in:
   a. **apertura**, con relativa introduzione di obiettivi e motivazioni;
   b. **conduzione**, attenzione alla comunicazione non verbale ed alle pause;
   c. **chiusura**, breve sommario dei punti salienti e ringraziamenti.
3. infine si fa una **rielaborazione**, si identificano i concetti tralasciati e si comunicano i risultati agli intervistati.

Le interviste sono tanto efficaci quanto energivore.

1.3.2.1.2 Workshop

Consiste in un gruppo di stakeholder che sviluppano i requisiti di sistema, il che può portare a risultati eccellenti. La fase di preparazione delinea:

- luogo;
- partecipanti;
- obiettivi;
- minute-taker;
- moderatore;
- risultati attesi.

Le fasi sono anche in questo caso tre:

1. in **apertura** si descrivono gli obiettivi, le tecniche adottate, l'agenda e le regole d'interazione;
2. la **conduzione** viene gestita dal moderatore e trascritta dal minute-taker;
3. la **chiusura** è stabilita dal moderatore ed avviene la raccolta dei risultati.

Nei giorni successivi, il minute-taker sistema gli appunti e li inoltra ai partecipanti che possono richiedere modifiche. In generale, lo sforzo richiesto è alto, così come i costi necessari per organizzarlo. I fattori critici per il successo sono: partecipanti, interesse, moderazione e luogo.

1.3.2.1.3 KJ method

Tecnica accessoria che permette di far emergere requisiti a partire da un gruppo di persone allo stesso tempo: riflessione individuale e lavoro di gruppo. Anche qua si hanno tre fasi:

1. **apertura**, presentazione degli obiettivi;
2. **conduzione**, strutturata in tre parti:
   a. riflessione individuale su post-it;
   b. presentazione e discussione di idee;
   c. raggruppamento e sintesi.
3. **chiusura**, rielaborazione dei risultati.

1.3.2.2 Formato di definizione dei requisiti

La composizione è basata su: ID, nome, descrizione e sorgente.

> **Principio** (MoSCoW). I requisiti dovrebbero essere descritti attivamente adoperando verbi come quelli racchiusi nel termine MoSCoW (*Must-o-Should-Could-o-Would*).

Tabella 1.3: Formato VOLERE©

NOME	DESCRIZIONE
ID	identificativo
tipo	tipologia
evento	CU di riferimento
motivazione	breve descrizione
sorgente	chi l'ha richiesto
criterio valutazione	come valutare il soddisfacimento
soddisfazione cliente	grado soddisfazione se completato
insoddisfazione cliente	grado insoddisfazione se non completato
conflitti	divergenze con altri requisiti
priorità	importanza
materiale di supporto	documentazione
storia	creazione, modifiche

La scoperta dei requisiti coinvolge gli scenari.

> **Definizione** (use case). Tecnica di descrizione di scenari vincolata a precisione massima.

Tutto ciò confluisce in un documento dei requisiti (nel nostro caso Visual Paradigm); esso contiene ciò che gli sviluppatori devono implementare.

1.3.3 Requisiti qualitativi

Definizione (qualità)**.** Caratteristica, proprietà o condizione di un'entità che serve a determinare la natura e a distinguerla da altre istanze simili.

Dunque lo stesso software realizzati con standard diversi si differenzia. Questi possono essere classificati a seconda della percezione degli utenti, qualità esterne, o degli sviluppatori, interne. I requisiti qualitativi sono classificati in:

- **di prodotto**, relativi all'erogazione del servizio;
- **organizzativi**, per lo sviluppo interno all'azienda;
- **esterni**, basati sul contesto generale di implementazione.

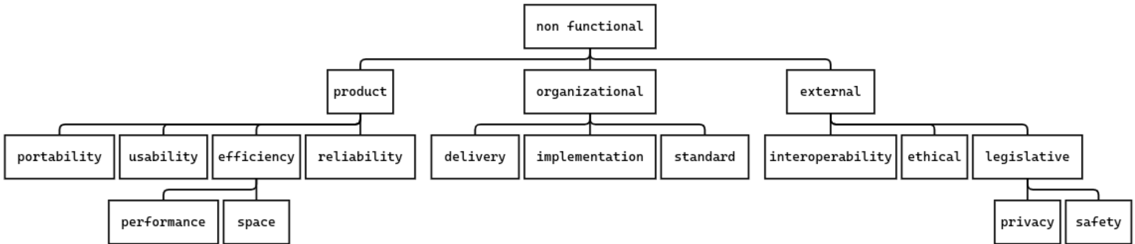


Figura 1.6: Classificazione dei requisiti qualitativi

Definizione (metrica)**.** Fornisce un'unità di misura che riferirà un insieme su cui è definita una relazione d'ordine. In campo software serve a definire se esso soddisfi una proprietà.

Spesso le qualità sono in contrasto fra loro; è dunque compito dell'ingegnere pesare le scelte da compiere. Non sono inoltre composizionali, manca quindi la possibilità di applicare modularità.

1.3.3.1 Metriche di prodotto

1.3.3.1.1 Correttezza funzionale

Un software corretto si comporta in accordo a quanto definito nella specifica del sistema.

Esempio (correttezza funzionale)**.** Viene definita una libreria di supporto a manipolazioni matematiche su numeri naturali.

```
public class MathOperationsInt {
    public int cube(int x) {
        return x * x * x;
    }
    public int abs(int x) {
        if (x > 0) return x;
        return -x;
    }
}
```

Codice 1.1: Libreria matematica interi

```
public class MathOperationsDouble {
    public double cube(double x) {
        return x * x * x;
    }
    public double abs(double x) {
        if (x > 0) return x;
        return x;
    }
}
```

Codice 1.2: Libreria matematica decimali

Il programmatore ha sbagliato nel definire la funzione che calcola il valore assoluto per numeri con virgola restituendo in output lo stesso input negativo passato. Pur evidenziato il problema, esso non si pone in quanto il dominio è quello dei naturali: perciò non si avrà mai a che fare con numeri minori di zero.

1.3.3.1.2 Affidabilità

Un sistema affidabile consente all'utilizzatore di confidare nel suo comportamento.

Esempio (affidabilità)**.** Viene definita una libreria di supporto a manipolazioni matematiche su numeri reali maggiori di -1.

```java
public class MathOperationsInt {
    public int cube(int x) {
        return x * x * x;
    }
    public int abs(int x) {
        if (x > 0) return x;
        return -x;
    }
}
```

Codice 1.3: Libreria matematica interi

```java
public class MathOperationsDouble {
    public double cube(double x) {
        return x * x * x;
    }
    public double abs(double x) {
        if (x > 0) return x;
        return x;
    }
}
```

Codice 1.4: Libreria matematica decimali

In questo caso nella stragrande maggioranza dei casi il sistema svolgono le richieste correttamente, ma nelle minime possibilità in cui il dato inserito sia proprio un decimale fra -1 e 0 la funzione restituisce un valore assoluto negativo, dunque errato.

In questo caso le metriche si applicano di un dominio reale:

- MTBF (*Mean Time Between Failures*);
- ANF (*Average Number of Failures*).

Il processo di misura tipicamente si basa sullo storico.

1.3.3.1.3 Robustezza

Controlla il comportamento ragionevole del software in circostanze non previste.

Esempio (robustezza)**.** Viene definita una libreria di supporto a manipolazioni matematiche sui naturali.

```java
public class MathOperationsInt {
    public int cube(int x) {
        return x * x * x;
    }
    public int abs(int x) {
        if (x > 0) return x;
        return -x;
    }
}
```

Codice 1.5: Libreria matematica interi

```java
public class MathOperationsDouble {
    public double cube(double x) {
        return x * x * x;
    }
    public double abs(double x) {
        if (x > 0) return x;
        return x;
    }
}
```

Codice 1.6: Libreria matematica decimali

Cosa accade se alla classe definita per i decimali passassi dei valori interi? E invece viceversa? Nel primo caso non ci sarebbero problemi, mentre nel secondo si incapperebbe in un errore non gestito tramite eccezione.

1.3.3.1.4 Efficienza

Viene svolto un compito sfruttando poche risorse, siano esse fisiche (memoria, rete o energia) o temporali: prestazioni. Queste sono di due tipologie:

a. **latenza**, velocità con cui il sistema risponde agli stimoli;

b. **throughput**, numero di stimoli che il sistema riesce a gestire nell'unità di tempo.

Ogni studio si può riferire a diversi casi: ottimo, medio e pessimo.

1.3.3.1.5 Usabilità
Semplicità d'utilizzo sperimentata dagli utenti target; presenta senza dubbio fattori soggettivi, ma vengono adoperate strutture standardizzate e valutati diversi aspetti:

- numero di errori commessi;
- numero di richieste di consultazione;
- tempo medio per raggiungere un obiettivo.

1.3.3.1.6 Verificabilità
Valutare la correttezza del software rilasciato tramite pre e post condizioni, o ancora meccanismi di eccezione. Il testing si divide in:

* **controllabilità**, fornire input al sistema al fine di manipolarlo verso un certo stato;
* **osservabilità**, controllare come il sistema si comporta durante la fase di elaborazione.

1.3.3.1.7 Manutenibilità
Modificare il prodotto dopo il rilascio per eventuali ritocchi che possono essere:

* correttivi;
* adattivi;
* perfettivi.

1.3.3.1.8 Riusabilità
Riferita alle singole componenti e non ad interi sistemi. Si considera l'usabilità del prodotto in un contesto differente da quello attuale.

1.3.3.1.9 Portabilità
Capacità di un sistema di poter esser trasferito da una piattaforma all'altra senza richiedere modifiche significative.

> **Esempio** (JVM). Il linguaggio di programmazione Java è molto versatile grazie alla JVM (*Java Virtual Machine*). Il codice viene infatti compilato in bytecode: un linguaggio macchina per la JVM. Successivamente sarà proprio quest'ultima ad interpretarlo seguendo le regole strutturali della macchina su cui sta eseguendo.

1.3.3.1.10 Comprensibilità
Semplicità di assimilazione del sistema software da parte di tecnici non creatori. È responsabilità dello sviluppatore aggiungere documentazione tale da facilitare l'approccio al proprio codice.

1.3.3.1.11 Interoperabilità
La standardizzazione permette di facilitare la comunicazione fra più sistemi.

> **Esempio** (Linux pipe). Meccanismo che permette di collegare l'output di un comando all'input di un altro. Si tratta di una sorta di tubo, pipe: ciò che esce da una estremità entra direttamente nell'altra.
>
> ```
> ls -a | grep "txt"
> ```
> Codice 1.7: Linux pipe
>
> Il comando `ls -a` elenca tutti i file e le cartelle contenute nel percorso in cui viene eseguito e, tramite la pipe, si filtra con `grep "txt"` i risultati contenenti la stringa evidenziata nel nome.

1.3.3.2 Metriche del processo di sviluppo

1.3.3.2.1 Produttività
Velocità ed efficienza con cui viene rilasciato il prodotto. Aspetto fortemente legato al contesto, per questo lo stesso processo può variare in produttività a seconda di dove lo si consideri.

1.3.3.2.2 Timeliness
Capacità di un processo software di rispettare le scadenze preposte, senza quindi andare a violare le deadline. Una pianificazione che considera i rischi è solitamente più efficace nella gestione delle tempistiche.

1.3.3.2.3 Visibilità
Proprietà che permette di capire lo stato di avanzamento del processo. La documentazione gioca un ruolo centrale, ma anche il tipo di approccio strutturale intrapreso.

1.4 GRASP

I (*General Responsibility Assignment Software Patterns*) mirano ad assegnare una responsabilità ad ogni oggetto. Si ha quindi una struttura RDD (*Responsibility Driven Development*) che prevede di applicare iterativamente i seguenti passi:

i. identificazione responsabilità;

ii. associazione di ogni responsabilità alle classi;

iii. domandarsi come la classe può soddisfare la responsabilità.

1.4.1 Creator

- **Problema**: chi si occupa di creare una nuova istanza di una classe?
- **Soluzione**: B crea A se vale almeno una delle seguenti condizioni
  - B registra oggetti di tipo A
  - B utilizza strettamente oggetti di tipo A
  - B contiene o aggrega con una composizione di oggetti di tipo A
  - B possiede dati per l'inizializzazione di oggetti di tipo A che passa per la creazione
- **Pro**: accoppiamento basso.
- **Contro**: complessità che tende a favorire deleghe a classi supporto.

1.4.2 Information expert

- **Problema**: esiste un principio per assegnare le responsabilità agli oggetti?
- **Soluzione**: assegna la responsabilità alla classe esperta, che non richiede informazioni a terzi.
- **Pro**: incapsulamento e coesione.
- **Contro**: troppe responsabilità alla singola classe.

1.4.3 Low coupling

- **Problema**: come si sostiene una dipendenza bassa?
- **Soluzione**: assegna la responsabilità senza aumentare l'accoppiamento.
- **Pro**: manutenzione, comprensione, riuso.
- **Contro**: l'accoppiamento con elementi stabili non causa problemi.

1.4.4 High cohesion

- **Problema**: come mantenere gli oggetti gestibili sostenendo low coupling?
- **Soluzione**: assegna responsabilità in modo che la coesione resti alta.
- **Pro**: manutenzione, comprensione, riuso.
- **Contro**: accetta una violazione per le classi in relazione a determinate specializzazioni.

> *«La modularità è la proprietà di un sistema che è stato decomposto in un insieme di moduli coesi e debolmente accoppiati»*
>
> G. Booch

1.4.5 Controller

- **Problema**: qual è il primo oggetto oltre l'interfaccia che riceve e coordina un'operazione?
- **Soluzione**: assegna la responsabilità ad una classe che
  a. rappresenta il sistema complessivo
  b. rappresenta uno scenario di un caso d'uso
- **Pro**: riuso e disaccoppiamento con l'interfaccia.
- **Contro**: controller colmi.

> **Nota bene** (GRASP controller ≠ MVC controller)**.** Questo pattern non è la C del modello MVC! Bensì ricade maggiormente nella sezione del model.

1.4.6 Pure fabrication

- **Problema**: come valutare una soluzione idonea di Information Expert che però viola High Coesion e Low Coupling?
- **Soluzione**: assegna un insieme di funzionalità coese ad una classe creata per tale scopo.
- **Pro**: riuso.
- **Contro**: tendenza alla progettazione procedurale.

1.4.7 Polymorfism

- **Problema**: come gestire alternative tipizzate?
- **Soluzione**: operazioni polimorfe.
- **Pro**: flessibilità ed estensione.
- **Contro**: rischio abuso per previsione di necessità future.

1.4.8 Indirection

- **Problema**: come disaccoppiare degli oggetti in modo da sostenere una bassa dipendenza e mantenere alto il potenziale di riuso?
- **Soluzione**: oggetti che mediano con altre componenti o servizi.
- **Pro**: ridotto accoppiamento.
- **Contro**: codice meno comprensibile.

1.4.9 Protected variants

- **Problema**: come progettare oggetti e sistemi in modo tale che le variazioni non abbiano un impatto indesiderato su altri elementi?
- **Soluzione**: identifica i punti in cui sono previste instabilità ed assegna proprio lì delle responsabilità stabilizzanti.
- **Pro**: modifica, basso accoppiamento, bassi costi e client isolati.
- **Contro**: rischio generalizzazione e complessità.

Tale principio ne generalizza altri: interfacce, polimorfismo, lookup, progettazione riflessiva, LSP, legge di Demetra o *"don't talk to strangers"*.

1.5 Verifica e validazione

Gli artefatti del processo di sviluppo devono essere coerenti, occorre dunque che rappresentino la stessa informazione seppur a diversi livelli di astrazione. Per quanto riguarda la validazione, si confronta il sistema rispetto all'idea coinvolgendo l'utente finale. Lo scopo è quello di capire se il prodotto è ciò che effettivamente si aveva necessità.

> **Definizione** (testing)**.** Tecnica di verifica, se svolta dallo sviluppatore, o di validazione, se perpetuata dall'utente finale.

Queste due attività non mirano a definire la correttezza del sistema, bensì a rassicurare l'utente finale sull'utilizzo di esso in base alle criticità. La verifica avviene per

- **approcci statici**, analisi del sorgente senza esecuzione del codice;
- **approcci dinamici**, osservazione dell'esecuzione.

> **Nota bene** (testing ≠ debugging)**.** La pratica del debugging non è incluso negli approcci statici e dinamici. Il testing differisce dal debugging in quanto nel primo si ricercano possibili difetti, mentre nel secondo l'intento è quello di risolvere problemi già identificati.

1.5.1 Testing

Dopo che un bug viene riportato si farà debugging per risolverlo e, al termine, verrà rilasciata una nuova versione del software.

> **Definizione** (rivalidazione)**.** Minimo sottoinsieme di test che devono essere rieseguiti dopo una modifica del software per accertarne le garanzie al pari dell'intera suite di testing.

Il testing si può utilizzare per verificare caratteristiche:

- **funzionali**, analizzando la relazione tra un input fornito e un output atteso;
- **qualitative**, osservando le performance del sistema.

Si può inoltre effettuare il **fault-based testing**: una tecnica che infetta volontariamente il codice per provare l'efficacia dei test esistenti.

> *«Il testing non può dimostrare l'assenza di difetti, ma solo la loro presenza»*
>
> Dijkstra

I test si dividono in:

* **unit testing**, eseguito sulle componenti software in modo isolato tramite approcci white box;
* **integration testing**, verifica che i legami fra unità funzionino correttamente;
* **system testing**, controlla che i requisiti di sistema siano rispettati avendo comportamenti attesi;
* **validation testing**, curato rispetto le esigenze dell'utente finale;
* **alpha testing**, effettuato in un ambiente controllato di tipo sandbox;
* **beta testing**, praticato direttamente dagli utenti finali.

> **Problema** (dell'oracolo)**.** Non è possibile automatizzare quale output di un test case sia corretto.

L'idea è certamente quella di realizzare test quanto più esaustivi possibili.

Sezione 2

UML

Linguaggio standardizzato di modellazione e specifica che si basa sul paradigma object-oriented con aspetto grafico semi-formale: definito da una sintassi formale con una semantica dei diagrammi informale. Il sistema è costruito come insieme di entità interagibili tra loro; il quale ha due differenti caratteristiche:

i. **struttura statica**, elementi necessari a modellare il sistema e rispettive correlazioni;

ii. **comportamento dinamico**, ciclo di vita degli oggetti e specifica delle collaborazioni.

> **Definizione** (oggetto)**.** Entità a cui sono associati dei dati e che fornisce funzionalità per l'accesso e la manipolazione di essi.

La struttura di UML è formata da tre elementi fondamentali:

a. **costituenti fondamentali**
   - entità (strutturali, comportamentali, raggruppamentali e informative)
   - relazioni
     - dipendenza
     - associazione
     - aggregazione
     - composizione
     - implementazione
     - generalizzazione
   - diagrammi

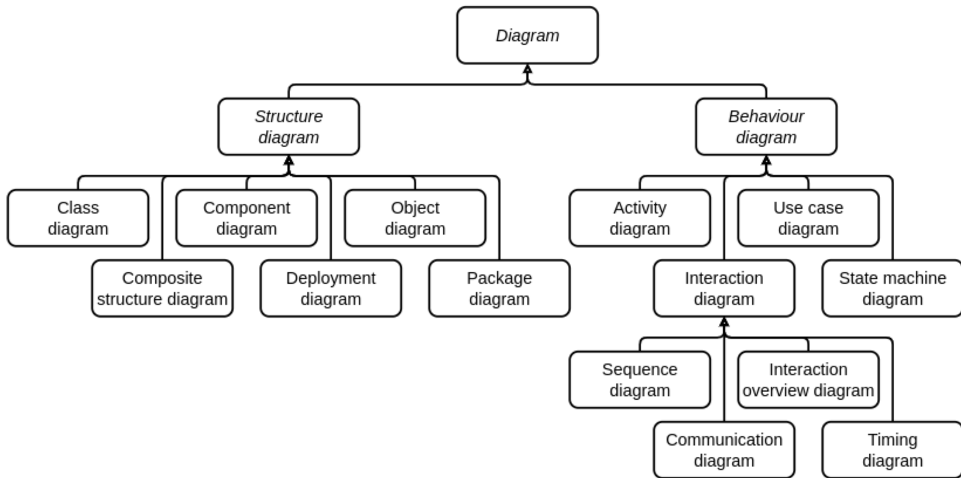


Figura 2.1: Gerarchia diagrammi UML

b. **meccanismi comuni**
tecniche comuni per raggiungere specifici obiettivi.
   - Specifiche: informazioni visive non grafiche ma testuali.
   - Ornamenti: versione arricchita dell'elemento baso per evidenziare informazioni.
   - Distinzioni comuni: differenza fra classificatore e istanza, e fra interfaccia e implementazione.
   - Meccanismi di estendibilità: vincoli, stereotipi ed etichette.

c. **architettura**
il modo in cui UML esprime l'architettura di un sistema.

> **Definizione** (architettura). Struttura organizzativa di un sistema, inclusa la sua scomposizione in parti, la loro connettività, l'interazione, i meccanismi ed i principi guida che insieme formano il progetto.

Si tratta di quattro prospettive, o viste, esprimibili con diversi diagrammi.

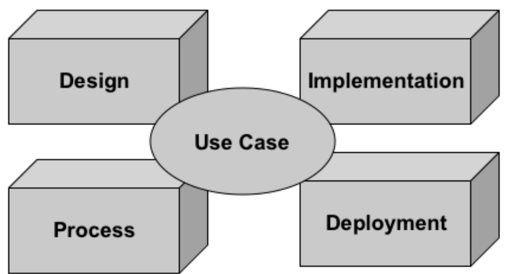


Figura 2.2: 4+1 views of software development

2.1 Use case diagram

> **Definizione** (caso d'uso). Descrizione di un insieme di scenari di utilizzo del sistema.

La procedura da seguire per stabilire i requisiti del progetto incappa innanzitutto nel chiedere agli attori che utilizzeranno il sistema finale come immaginano l'uso e l'interazione con esso:

a. analisi dei requisiti; b. identificazione degli attori; c. sviluppo possibili use cases.

Comprendere la granularità della descrizione del sistema è fondamentale.

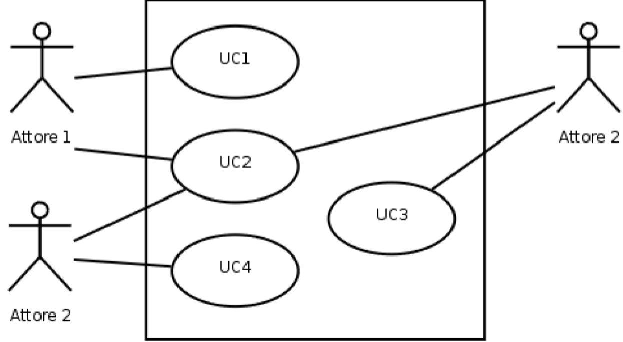


Figura 2.3: Rappresentazione use case diagram

Ogni caso d'uso non deve quindi essere troppo corposo, per assicurarsi di ciò esistono tre tecniche:

- **test del capo**, immaginarsi di dover dichiarare al datore di lavoro i compiti svolti;
- **test processo di business elementare**, ciò che si sta facendo deve permettere di raggiungere un determinato obiettivo;
- **test della dimensione**, limitare la descrizione dettagliata ad una decina di righe.

La descrizione dei del caso d'uso ha diversi formati:

* **breve**, frase descrittiva dell'obiettivo;
* **informale**, aggiunta di differenti scenari;
* **dettagliato**, introduzione di pre e post condizioni.

In fase di avvio gli use cases dettagliati saranno un 20%, giusto i fondamentali.

> **Nota bene** (uso dei sinonimi)**.** Nella descrizione dei requisiti è errato usufruire di sinonimi!

Più avanti, durante l'elaborazione, si arriva ad una descrizione precisa sull'80% di essi. Le sezioni imprescindibili del formato dettagliato sono:

- id;
- nome;
- attori;
- descrizione;
- pre-condizioni;
- post-condizioni;
- frequenza d'uso;
- sequenza principale;
- sequenza alternativa.

Le sequenze sono numerate ed ogni passo ha la struttura che segue:

`<numero>. <attore><azione>`

> **Nota bene** (sequenza come dialogo)**.** Il sistema reagisce, non agisce! Alle azioni corrispondono delle reazioni.

I casi d'uso ottenuti sono categorizzabili in due fazioni: *success* ed *exception scenario*. Il primo riguarda il percorso liscio, senza intoppi, mentre il secondo comprende tutte le digressioni non inizialmente desiderate dall'utente. Una volta entrato nella sequenza alternativa, l'attore effettua delle modifiche per poi tornare in uno stato di sequenza principale.

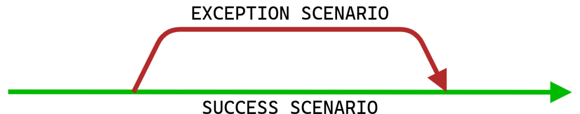


Figura 2.4: Success and exception scenarios

Esistono delle sequenze alternative accessibili in ogni instante, come l'azione di annullamento.

> **Definizione** (condizioni)**.** Frasi booleane formate da una composizione di preposizioni atomiche.
>
> > **Definizione** (pre-condizioni)**.** Condizioni senza dipendenze rispetto ai dati immessi nel sistema verificabili da quest'ultimo per rendere disponibile il caso d'uso.
>
> > **Definizione** (post-condizioni)**.** Condizioni attese successivamente allo svolgimento del caso d'uso.

2.1.1 Attori

Gli use cases vengono riportati dal punto di vista dell'attore che li compie. Per questo, nella descrizione formale, i verbi non saranno passivi ma attivi.

> **Nota bene** (attore ≠ entità). Gli attori non rappresentano vere e prorie entità, ma i ruoli che queste possono decidere di avere!

La stessa entità può rappresentare più ruoli. Gli attori sono rappresentati tramite stickman.

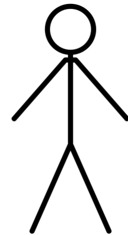


Figura 2.5: Attore

Per individuarli bisogna chiedersi dei quesiti che vanno a fornire una visione elevata antecedente allo sviluppo.

- Chi o cosa usa il sistema? Chi installa il sistema?
- Chi partecipa alle varie fasi del ciclo di vita del sistema?
- Chi ottiene informazioni dal sistema e a chi esso ne fornisce?
- Ci sono azioni che vengono eseguite a intervalli prestabiliti o a scadenza?

Quest'ultimo punto vede spesso presentarsi un attore inusuale: il tempo. Gli attori di un sistema possono essere a loro volta dei sistemi esterni che attivano o supportano gli use cases implementati.

2.1.1.1 Generalizzazione degli attori

Vi è la possibilità che molti attori agiscano su diversi sistemi. E succede inoltre con utenti con usi simili di sistemi, differenziati per pochi passi, siano rappresentati da più attori. Per risolvere queste complicanze si ha:

- **generalizzazione tra casi d'uso**;
  Un caso d'uso può essere definito come specializzazione di un altro. Vengono ereditate le caratteristiche e vi è data la possibilità di aggiungerne altre.
- **inclusione**;
  I casi d'uso condividono diversi passi che andrebbero riscritti. Per questo si pone un meccanismo che indica che il comportamento è definito altrove.
- **estensione**.
  Vi sono casi d'uso completi e funzionanti a cui è consentito aggiungere funzionalità senza le quali risulta comunque possibile svolgere le attività consolidate dalla specifica. Il caso d'uso deve dichiarare l'estensione!

Gli use cases possono essere definiti in ogni fase del processo unificato, tranne che nell'ultima.

2.2 Class diagram

Definizione (oggetto)**.** Entità discreta, con confini ben definiti, che incapsula stato e comportamento; un'istanza di una classe.

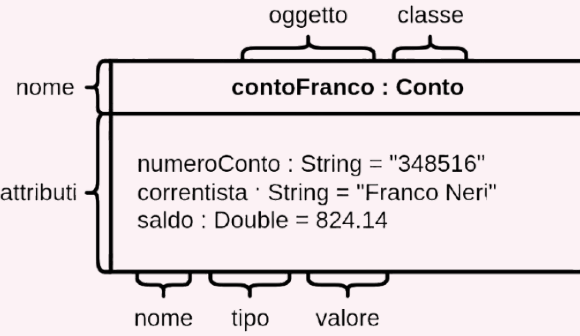


Figura 2.6: Rappresentazione UML di un oggetto

Ogni oggetto è singolare e viene identificato tramite una propria combinazione di aspetti.

Definizione (identità)**.** Riferimento ad un oggetto che lo rende univocamente distinguibile da altri suoi simili.

– **Stato**: valori degli attributi e connessione ad altri oggetti.

– **Comportamento**: azioni che vengono richieste all'oggetto e possono cambiarne lo stato.

2.2.1 Classi

Definizione (classe)**.** Descrittore di un insieme di oggetti che condividono gli stessi attributi, operazioni, metodi, relazioni e comportamento.

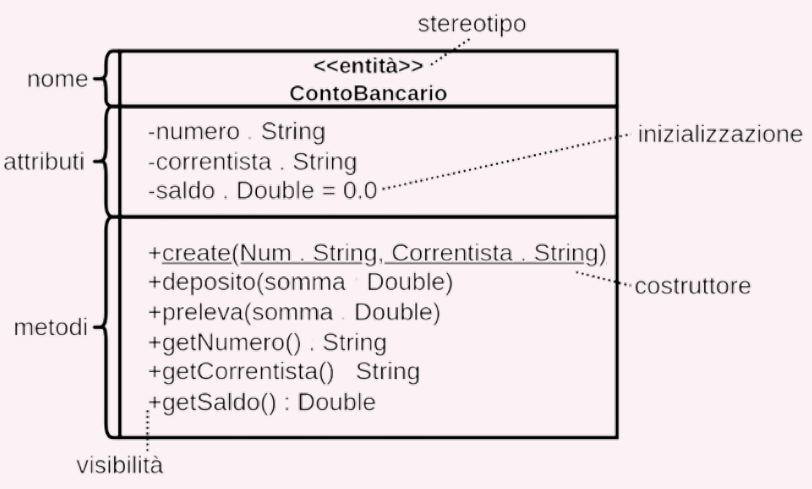


Figura 2.7: Rappresentazione UML di una classe

classe : oggetto = classificatore : istanza

Il modificatore di visibilità è applicato sia agli attributi che ai metodi.

Tabella 2.1: Livelli di accesso dei modificatori

SIGN	KEYWORD	CLASS	PACKAGE	SUBCLASS	WORLD
+	`public`	✓	✓	✓	✓
#	`protected`	✓	✓	✓	–
~	`package`	✓	✓	–	–
-	`private`	✓	–	–	–

Definizione (dipendenza)**.** Relazione tra due elementi in cui un cambiamento di un elemento può influenzare o fornire informazioni di cui l'altro elemento ha necessità.

2.2.1.1 Classi e stato del progetto

La distinzione relativa alle fasi di sviluppo è fondamentale per definire una distinzione di obiettivi. In quella iniziale di analisi si specifica cosa il sistema deve fare, mentre in una successiva fase di progettazione si stabilisce come deve farlo.

* **Fase di analisi**
  Vengono nominate le classi con termini concreti ma non tecnici, definiti gli attributi senza tipazione e decise le operazioni in modo tale che si rispetti l'SRP. L'individuazione delle classi avviene un miscuglio di varie tecniche:

  ◦ **nome-verbo**, procede similmente all'analisi grammaticale recuperando i termini nella documentazione.
  ◦ **classi CRC**, le *Class Responsibility Collaborators* si sviluppano sul ragionamento di committenti ed analisti.
  ◦ **elenco categorie concettuali**, si costruiscono degli insiemi ideali dove ripartire le classi ideate.

  Una buona classe di analisi deve:

  - rispettare l'SRP;
  - aderire al vincolo sul nome;
  - avere interdipendenza minima;
  - essere riconducibile ad un concetto della realtà d'interesse.

  La struttura ottenuta dovrebbe evitare l'isolatezza, la presenza di troppe classi semplici o poche complesse, e alberi di ereditarietà profondi.

* **Fase di progettazione**
  Si aggiungono i modificatori di visibilità, i parametri ed i valori di inizializzazione.

2.2.2 Relazioni

> **Definizione** (collegamento)**.** Connessione semantica dinamica tra due oggetti che consente loro di scambiare messaggi.

Il diagramma degli oggetti mostra un'istantanea del sistema in un dato momento con i rispettivi collegamenti.

2.2.2.1 Associazioni

> **Definizione** (associazione)**.** Si ha la seguente relazione
>
> $$\text{collegamenti} : \text{oggetti} = \text{associazioni} : \text{classi}$$
>
> Se esiste quindi un collegamento tra oggetti, deve esistere un'associazione fra le classi corrispondenti.
>
> $$\longrightarrow$$

Ogni associazione dichiara le seguenti informazioni:

* **nome** esplicativo;
* **ruoli**;
* **molteplicità**, cardinalità del collegamento;

  Tabella 2.2: Indicatori di molteplicità

MIN	MAX	SIGN
0	1	0..1
1	1	1
0	∞	0..*
1	∞	1..*
2	5	2..5

  Tale specifica va già espressa nelle classi di analisi.
* **navigabilità** indicata da una freccia.
  Non va dichiarata nel modello concettuale. Viene mostrata secondo tre modalità:

  Tabella 2.3: Indicatori di navigabilità

FRECCIA	VERSO
$\longrightarrow$	monodirezionale
$\longleftrightarrow$	bidirezionale
——	indefinita

  Nelle relazioni uno-a-uno spesso una classe è ancillare all'altra, dunque, se non aggiunge informazioni proprie, è consigliato rimuoverla ed aggregarle.

Il legame associativo fra due classi può riportare informazioni contenute in una classe associazione.

2.2.2.2 Dipendenze

> **Definizione** (dipendenza)**.** Indica una relazione tra due o più elementi del modello, dove un cambiamento ad uno di essi può influenzare o fornire informazioni necessarie all'altro.
>
> ----------->

Si definiscono con frecce tratteggiate e vanno rappresentate non nel modello concettuale ma in quello progettuale. Possono evidenziare diversi tipi di rapporto:

- **uso**, elemento che ne utilizza un altro per fornire un servizio o un'informazione;

  keywords: usa, chiama, parametro, invia, istanzia

- **astrazione**, un elemento rappresenta un'astrazione di un altro;

  keywords: traccia, sostituisce, raffina, deriva da

- **permesso**, un elemento ha il permesso di accedere o modificarne un altro;

  keywords: accede, importa, permette

2.2.2.3 Generalizzazioni

Se la classe A è generalizzata dalla B, quest'ultima è definita una specializzazione della prima.

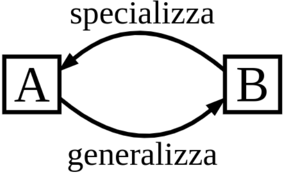


Figura 2.8: Reciprocità generalizzazione e specializzazione

Questo legame è strettissimo, si tratta infatti del verbo essere: *isA*. Viene anche definito come un meccanismo di riuso, la sottoclasse può infatti ridefinire (`override`) quanto stabilito nella superclasse. Le dinamiche riutilizzate sono: attributi, operazioni, relazioni e vincoli. Le classi ereditarie devono esser poste allo stesso livello d'astrazione, dunque essere comparabili.

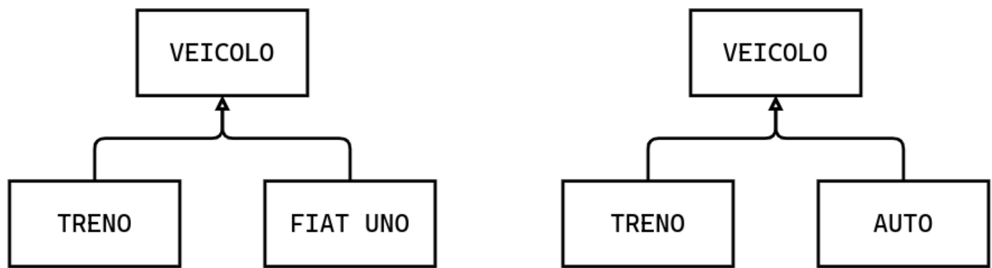


Figura 2.9: Coerenza orizzontale delle sottoclassi

2.2.2.3.1 Processi di modellazione

Il MOF *Meta-Object Facility* definisce regole per creare l'UML, il quale ne definisce altre per generare lo user model che a sua volta specifica gli elementi del sistema poi concretizzati tramite le istanze.

```
Meta-Meta-Model:  MOF
└─Meta-Model:  UML
   └─Model:  user model
      └─Instances:  instanze runtime
```

2.2.2.3.2 Insieme di generalizzazione

Definizione (insieme di generalizzazione)**.** Nello specializzare le classi si viene a creare automaticamente un albero di ereditarietà. L'insieme di tutti i suoi nodi è detto di generalizzazione.

Esso ha due caratteristiche primarie: completezza e disgiunzione, che vengono specchiate da incompletezza e sovrapposizione.

a. **Completezza**: ogni elemento figlio ha un solo padre.

b. **Disgiunzione**: assenza di oggetti duplicati in diverse classi.

2.2.2.3.3 Polimorfismo

Definizione (polimorfismo)**.** Capacità di assumere differenti comportamenti dipendentemente al contesto.

Esiste il problema della classe base fragile che ci suggerisce di non ereditare da classi non costruite personalmente.

Problema (diamante)**.** Si ha una gerarchia di classi dove una figlia eredita da due differenti padri che a loro volta ereditano da una stessa entità comune.

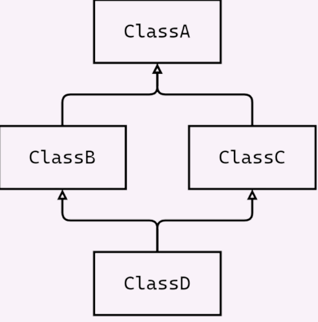


Figura 2.10: Ereditarietà a diamante

Se `ClassD` eredita lo stesso attributo dalle due sue generalizzazioni il compilatore non riuscirebbe a sapere quale utilizzare rendendo il codice non deterministico.

Principio (Liskov substitution)**.** In un programma è sempre possibile sostituire le istanze di una classe con le istanze di una sua sottoclasse senza alterare il comportamento del programma!

Spesso è preferibile fare a meno della generalizzazione preferendo piuttosto la composizione. Questo è dato dal legame più debole, e quindi pratico, stabilito fra le classi.

Tabella 2.4: Comparazione ereditarietà e composizione

EREDITARIETÀ	COMPOSIZIONE
white-box	black-box
dipendenze	libera
supporto	runtime

2.2.2.4 Progettazione

Viene intrapreso il passaggio dal modello di progetto dettagliato all'attività d'implementazione. Le classi di progettazione contengono infatti tutte le informazioni necessarie, verranno quindi definite mature, per poterle sviluppare nel codice. Inoltre, le relazioni nella fase di analisi non sono spesso supportate dai linguaggi di programmazione.

- **Aggregazione**: relazione del tipo tutto/parte dove quest'ultima mostra una sua indipendenza e può essere porzione di più entità di aggregazione.
- **Composizione**: relazione molto forte di tipo tutto/parte dove quest'ultima non può esistere se non come componente di una sola entità più complessa.

Entrambe sono transitive ed asimmetriche. In un diagramma di progetto la regola è:

$$\begin{aligned} 1-1 &\implies \text{composizione o attributo} \\ n-1 &\implies \text{aggregazione} \\ 1-n &\implies \text{uso di classi collezione} \end{aligned}$$

2.3 Interaction diagram

Occorre stabilire come si rapportano le classi per realizzare il comportamento definito nei casi d'uso. Per concretizzare questi ultimi si sfruttano i diagrammi di interazione, le cui due principali entità evidenziate sono:

- **linee di vita**, elemento di una classe che riporta nome, tipo e selettore;
- **messaggi**, tipo di interazione fra due linee risolvibile in una chiamata di un'operazione, creazione o distruzione di un'istanza, e invio di un segnale.

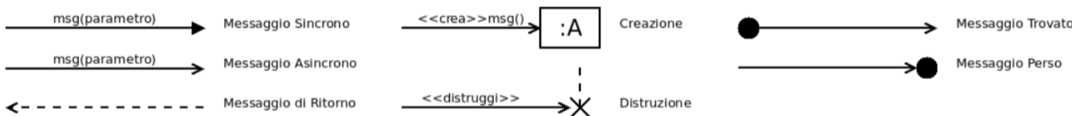


Figura 2.11: Tipologie di messaggi dell'integration diagram

2.3.1 Sequence diagram

I diagrammi di sequenza si dividono in SSD (*System Sequence Diagram*), cui ognuno è associato ad un caso d'uso, e quelli di dettaglio, che vanno ad approfondire il comportamento del sistema nell'elaborare le richieste dell'utente. Nei primi il sistema ha un'unica lifeline, nei secondi si frammenta nelle molteplici componenti presenti.

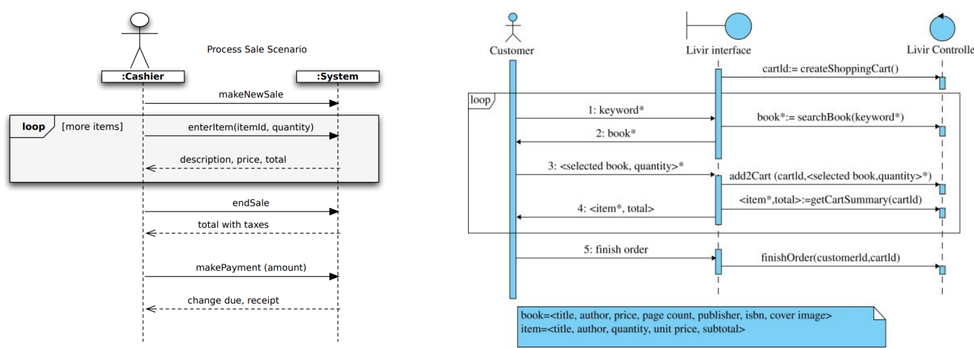


Figura 2.12: SSD e SD a confronto

È inoltre possibile rappresentare elementi complessi attraverso l'uso di frammenti combinati; questi sono composti da un operatore, almeno un operando e potrebbe avere delle condizioni di guardia:

- `par`, parallelismo;
- `loop`, ciclo (`do-while`);
- `break`, uscita operando;
- `neg`, iterazioni non valide;
- `opt`, condizione opzionale;
- `alt`, condizione alternativa;
- `critical`, esecuzione atomica;
- `ignore`, elenca messaggi omessi;
- `seq`, sequenzializzazione debole;
- `strict`, sequenzializzazione forte;
- `ref`, riferimento ad altro diagramma;
- `assert`, unico comportamento valido.

2.4 Petri nets

Le reti di petri sono un formalismo per la modellazione di sistemi concorrenti, asincroni, distribuiti, paralleli e stocastici. Si tratta di una tupla $< \mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{W}, \mathcal{M}_0 >$:

- $\mathcal{P}$, insieme finito di piazze;
- $\mathcal{T}$, insieme finito di transizioni;
- $\mathcal{F} \subseteq \{\mathcal{P} \times \mathcal{T}\} \cup \{\mathcal{T} \times \mathcal{P}\}$, relazione di flusso;
- $\mathcal{W} : \mathcal{F} \to \mathbb{N}^+$, funzione peso che associa un valore non nullo agli elementi di $\mathcal{F}$;
- $\mathcal{M}_0 : \mathcal{P} \to \mathbb{N}$, marcatura iniziale che indica lo stato di partenza.

Deve poi valere $\mathcal{P} \cup \mathcal{T} \neq \emptyset$ e $\mathcal{P} \cap \mathcal{T} = \emptyset$.

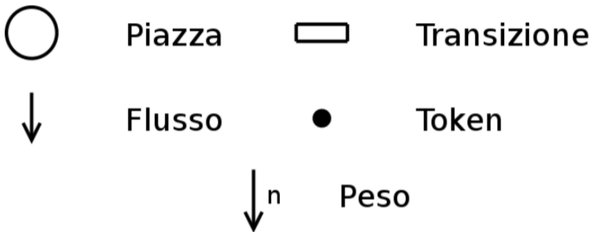


Figura 2.13: Notazione grafica delle reti di Petri

Definizione (piazza di input). Piazza collegata ad un flusso entrante ad una transizione.

Definizione (piazza di output). Piazza collegata ad un flusso uscente da una transizione.

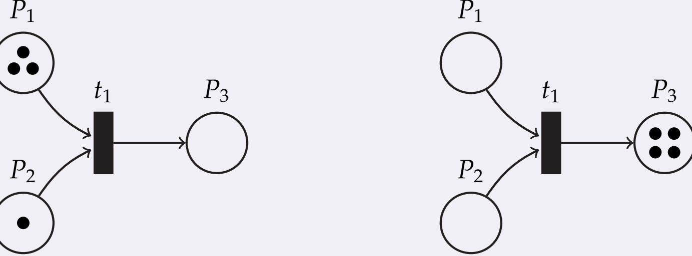
Le Petri nets servono a capire i meccanismi di scatto, ossia come procede il comportamento.

Definizione (scatto della transizione). Una transizione è abilitata, e può dunque "scattare", se, e solo se, tutte le piazze collegate ad un qualsiasi flusso di input alla transizione contengono un numero di token maggiore o uguale al peso dell'elemento flusso che le collega alla transizione.

Quando una transizione "spara" ogni piazza collegata ad un flusso uscente dalla transizione riceverà un numero di token pari al peso del corrispondente flusso. Le transizioni con nessun flusso entrante sono sempre abilitate e una transizione può avere più flussi entranti ed uscenti.

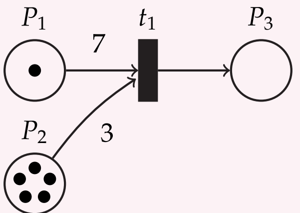
Esercizio (somma di interi)**.** Si voglia realizzare una somma fra due numeri interi n ed m tramite reti di Petri.

La rete risultante è della forma che segue.



In P_1 e P_2 ci saranno i numeri da sommare, sotto forma di token, ed in P_3 il totale.

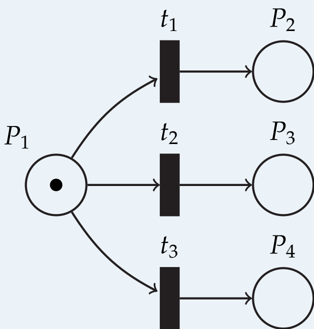
Definizione (deadlock)**.** Rete di Petri con nessuna transizione abilitata; perciò la sua marcatura rappresenta lo stato finale della rete.



2.4.1 Contesa dei token

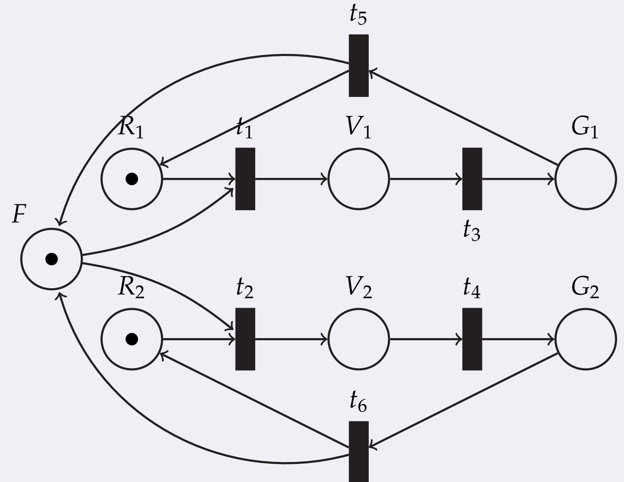
A volte può accadere che due o più transizioni si contendano un singolo token rendendo il processo non deterministico.

Esempio (contesa dei token)**.** Data una rete come segue sarà impossibile sapere a priori se la transizione attivata sarà t_1, t_2 o t_3.



Nel formalismo non vi sono infatti politiche di risoluzione dei conflitti, ma devono invece essere implementate manualmente.

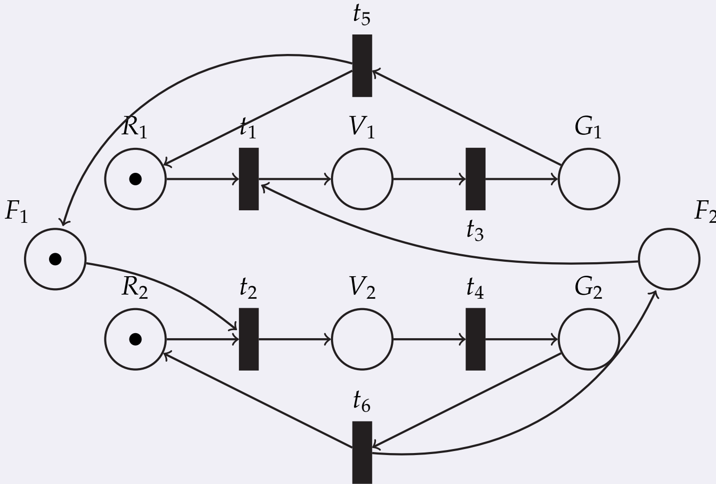
Esercizio (meccanismo di fairness)**.** Si ha una strada con un incrocio dove sono presenti due semafori.



Così facendo il verde potrebbe rimanere ciclicamente su uno dei due semafori per periodi prolungati, mantenendo gli automobilisti dell'altra strada fermi per troppo tempo.

. . . .

Per risolvere ciò si inserisce un artefatto che consente la distribuzione paritaria del permesso di avanzamento.



In questo modo una volta che un ciclo di uno dei due semafori sarà terminato, inizierà quello dell'altro e viceversa all'infinito.

2.4.2 Estensione delle reti di Petri

Per modellare sistemi complessi è stato necessario ampliare le funzionalità di questo formalismo:

- **CPN** (*Coloured Petri Nets*), associazione di un colore ai token fornendo tipizzazione;
- **politiche di priorità**, transazioni con associato un valore per eliminare il non determinismo;
- **temporizzazione**, ogni transazione ha un valore di tempo minimo e massimo associato entro il quale se viene abilitata dovrà scattare;
- **SPN** (*Stochastic Petri Nets*), reti temporizzate con associate funzioni di distribuzione.

2.4.3 Diagrammi di attività

La semantica è descritta dal formalismo delle reti di Petri. In campo di UP si sfrutta nella sezione di analisi e progettazione, ed anche nella modellazione di BP (*Business Process*). L'attività è composta

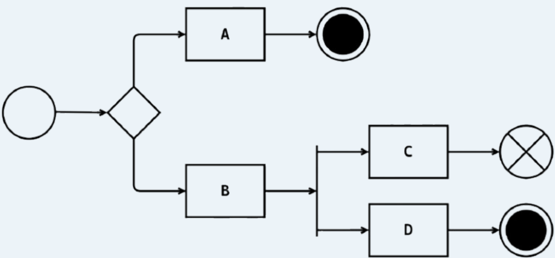
da archi e nodi; i primi possono essere di controllo o di oggetti, mentre i secondo si dividono in:

- **azione**, devono essere validate delle precondizioni per far partire l'attività e che i token verranno emessi solo se le postcondizioni vengono valutate positivamente;
- **controllo**, gestiscono il flusso dei token;

Tabella 2.5: Notazione grafica nodi di controllo

NOME	SIMBOLO
iniziale	
finale dell'attività	
finale del flusso	
decisione	

NOME	SIMBOLO
fusione	
biforcazione	
ricongiunzione	

> **Esempio** (differenza fra nodo finale dell'attività e del flusso)**.** Viene di seguito mostrata una rete con molteplici scenari di terminazione possibili.
>
> 
>
>
> Figura 2.14: Circuito con nodi di controllo finali
>
> i. A e si arriva al termine.
> ii. B, flusso parallelo fra C e D, termina prima C, si arriva al termine e troncamento esecuzione ramo D.
> iii. B, flusso parallelo fra C e D, termina prima D, si arriva al termine e continuazione esecuzione ramo C.

- **oggetto**, fungono da buffer di dati ordinati (per default FIFO) ed è possibile specificarne la dimensione.

2.5 Component diagram

Nell'approcciare grandi progetti occorre scomporli in sottosistemi collaborativi. Tale cooperazione è definita attraverso interfacce specificanti un contratto che le rispettive implementazioni si impegnano a rispettare.

> **Definizione** (componente)**.** Modulo indipendente di un sistema fatto per essere prelevato e riutilizzato altrove. Esiste la possibilità di esploderlo per mostrarne gli elementi.

Si realizza dunque un insieme di funzionalità coese con metodo black box senza assunzioni da poter poi rimplementare in base alla necessità.

2.6 State chart diagram

Vengono rappresentate le differenti fasi che un oggetto può passare durante il suo ciclo di vita. Il tutto si basa su SM (*State Machine*): uno strumento per la modellazione del comportamento dinamico dei classificatori. Ogni SM è composta da: stati, eventi e transizioni.

* **Stati**: si mantiene una traccia non univoca di ciò che è avvenuto fin'ora.

  > **Definizione** (stato)**.** Condizione della vita di un oggetto durante la quale esso soddisfa una condizione, esegue un'attività o attende un evento.
  >
  > - **Rappresentazione**: rettangolo con angoli arrotondati.

  Gli aspetti che lo determinano sono: valori attributi, relazioni fra oggetti ed attività in esecuzione. Esistono diverse tipologie di stati:

  - **stati giunzione**, usati per rappresentare controlli;
  - **stati composti**, contengono altri stati annidati. Ogni transizione dello stato contenitore viene ereditata dagli stati contenuti;
  - **stati con memoria**, mantengono una memoria in ingresso e uscita dallo stato.

* **Eventi**: accadimento che può causare un cambiamento di stato.

  > **Definizione** (evento)**.** Specifica di un'occorrenza di interesse che ha una collocazione spaziale e temporale.
  >
  > - **Rappresentazione**: etichetta applicata alle transizioni tra stati.

  Le tipologie di eventi sono:

  - **di chiamata**;
  - **di segnale**, mettono in comunicazione diverse SM;
  - **di variazione**, risorsa che varia stato;
  - **temporale**, riguardanti scadenza e durata.

* **Transizioni**: passaggio da uno stato all'altro.

  > **Definizione** (transizione)**.** Comportamento dinamico del sistema che mostra come esso reagisce agli eventi.
  >
  > - **Rappresentazione**: frecce che collegano due stati.

I sistemi si categorizzano in:

- **reattivi**, reagiscono ad uno stimolo;
- **proattivi**, iniziano un'azione.

Le SM possono essere associate a:

· **comportamento**, rappresentano l'evoluzione di una classe concreta;

· **protocollo**, descrive i metodi di un'interfaccia. Per la chiamata delle operazioni si hanno:

  - condizioni
  - risultati attesi
  - ordine

  Non contiene aspetti relativi alle specifiche implementazioni della classe.

Se si ha un'interfaccia descritta da una SM del protocollo e implementata da una classe descritta da una SM del comportamento, queste due SM devono essere coerenti!

> **Esempio** (SM di una lampadina)**.** Una SM descrive una lampadina. Fisicamente, essa può essere impostata su tre diverse condizioni: spenta, accesa e soffusa. La SM considera solamente le prime due, ma ciò non vuol dire che non venga rispettata la coerenza. Chiaramente il contrario non sarebbe ugualmente valido.

Di norma, se la SM del protocollo contiene meno informazioni di quella del comportamento, allora la coerenza viene comunque rispettata.

Sezione 3

OOD

Prima di approfondire l'*Object Oriented Design* occorre rinfrescare alcuni concetti basilari. Si ha innanzitutto il concetto di classe e i relativi componenti.

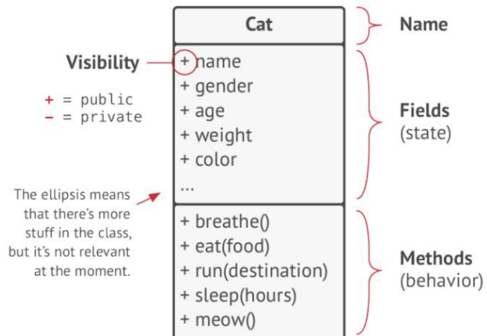


Figura 3.1: Diagramma UML di una classe

Molto comune è la gerarchizzazione delle entità presenti nello schema.

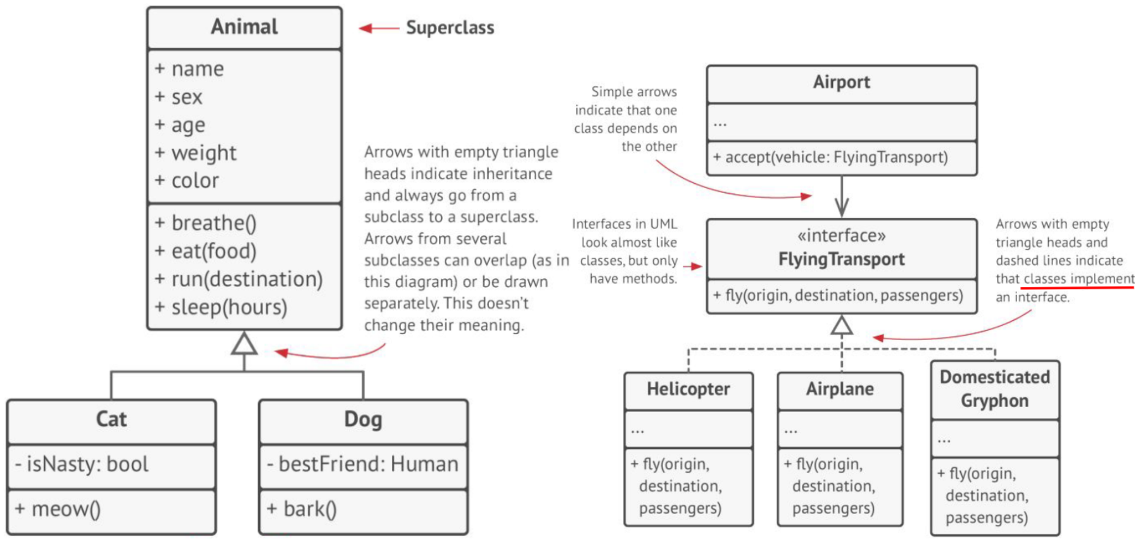


Figura 3.2: Generalizzazione classi e interfacce

In questo caso venono mostrate due generalizzazioni, la prima coinvolge una classe padre, o superclasse, e due classi figlie, o sottoclassi; mentre la seconda include un'interfaccia che si interpone fra fra la superclasse e la sottoclasse fornendo uno strato di maggiore modularità. Servono a definire i contratti che le classi che la implementano devono rispettare.

3.1 OOP

L'*Object Oriented Programming* ha dei cardini strutturali:

- **astrazione**
  Permette di semplificare il codice andando a rappresentare gli aspetti fondamentale di un comportamento. Nell'approcciare un problema il primo passo riguarda l'astrazione di quest'ultimo.
- **incapsulamento**
  Vengono nascosti i dettagli dell'oggetto e permesso l'accesso agli attributi solo tramite dei metodi appositi, ciò garantisce maggiore sicurezza.
- **ereditarietà**
  Consente di estendere una classe fornendo alla sottoclasse i suoi metodi, questa potrà quindi riutilizzare degli elementi della superclasse ed è capace inoltre di adottarne di nuovi.
- **polimorfismo**
  Oggetti di classi differenti possono esser trattati in modo uniforme se condividono uno stesso metodo o interfaccia.

3.1.1 Relazioni tra oggetti

Le istanze delle classi possono essere associate secondo varie tecniche.

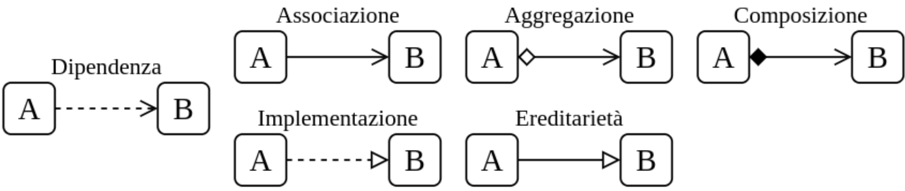


Figura 3.3: Relazioni tra oggetti

- **Dipendenza**: A può essere influenzata da modifiche di B.
- **Associazione**: A conosce B e dipende da essa.
- **Aggregazione**: A conosce B, ne dipende ed ne è composta.
- **Composizione**: A conosce B, ne dipende, ne è composta e ne gestisce il ciclo di vita.
- **Implementazione**: A definisce metodi dichiarati dall'interfaccia B e dipende da essa.
- **Ereditarietà**: A eredita l'interfaccia e l'implementazione di B ma può estenderla.

3.2 Software qualitativo

Il codice deve essere flessibile ma stabile. Esistono dei principi universali di design da rispettare:

- **semplicità**, mantenendo chiarezza e comprensibilità;
- **flessibilità**, avendo una struttura adattabile a cambiamenti futuri;
- **modularità**, suddividendo in componenti indipendenti e riutilizzabili;
- **manutenibilità**, correggendo e migliorando nel tempo.

Oltre ciò, è bene creare software di qualità per permetterne il riutilizzo; andando quindi a progettare con ottica di estensibilità. Vero è che maggiore è il codice scritto, più alte sono le possibilità di introdurre errori.

3.2.1 Isolamento dei cambiamenti

Occorre andare a definire una separazione dei concetti presenti all'interno del codice per minimizzare l'impatto di futuri cambiamenti nel sistema. Per questo, ogni programma va basato sulle astrazioni e non sulle implementazioni.

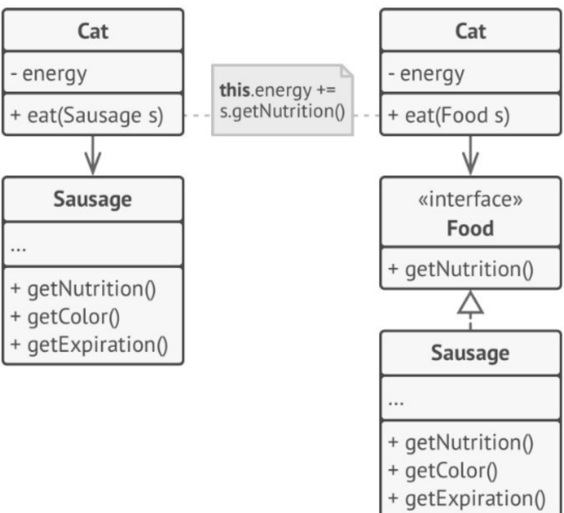


Figura 3.4: Introduzione di astrazioni

Per applicare questo principio bisogna:

1. identificare i metodi di cui un oggetto necessita dall'altro;
2. descrivere questi metodi in un'interfaccia o classe astratta;
3. la classe dipendente deve implementare tali metodi;
4. la sottoclasse deve dipendere dall'astrazione, non dalla classe concreta.

Inizialmente il codice si complica, ma è facile vederne le facilitazioni verso un'estensione futura.

3.2.1.1 Composizione rispetto ereditarietà

L'ereditarietà ha delle problematiche:

- **interfaccia fissa**, occorre implementare anche i metodi non necessari;
- **compatibilità del comportamento**, il comportamento sovrascritto dalla sottoclasse deve essere compatibile con quello della superclasse;
- **rottura dell'incapsulamento**, le sottoclassi accedono ai dettagli interni alla superclasse;
- **accoppiamento stretto**, modifiche alla superclasse possono intaccare il funzionamento della sottoclasse;
- **ereditarietà parallela**, generazione di gerarchie di classi complesse.

Per questo è preferibile favorire la composizione: un miglior incapsulamento e flessibilità garantiti dalla relazione *has a* anziché *is a*.

3.2.2 Principi di OOD

I principi SOLID[1] sono stati introdotti per rendere il software più manutenibile. Il termine SOLID è un acronimo contenente cinque differenti principi al suo interno.

> **Principio** (open-closed)**.** Le componenti software sono aperte alle estensioni, ma chiuse alle modifiche!
>
> * Un modulo è **aperto** se permette le estensioni.
> * Un modulo è **chiuso** se è disponibile per l'uso di altre componenti senza la necessità di conoscere l'esatta implementazione.

Ciò spinge all'utilizzo del polimorfismo andando ad estendere la classe base e non intaccandone il comportamento originario.

> **Principio** (dependency inversion)**.** Le dipendenze devono sempre essere verso le astrazioni, mai verso le concretizzazioni! Le interfacce non dipendono dalle classi concrete, entrambe devono infatti dipendere dalle astrazioni.

Vengono disaccoppiati i moduli di alto e basso livello intermediandoli con un'interfaccia.

> **Principio** (Liskov substitution)**.** È sempre possibile sostituire le istanze di una classe con quelle di una sua sottoclasse senza alterare il comportamento del programma!

Alcuni suggerimenti per la corretta applicazione del LSP:

- la sottoclasse deve preservare le invarianti dalla superclasse;
- la sottoclasse non deve rafforzare le condizioni imposte dalla superclasse;
- la sottoclasse non deve indebolire le condizioni imposte dalla superclasse;
- la sottoclasse non deve modificare i valori dei campi privati dalla superclasse;
- la sottoclasse non può lanciare eccezioni più specifiche di quelle lanciate dalla superclasse;
- i tipi dei parametri della sottoclasse devono essere uguali o più astratti di quelli della superclasse;
- il tipo di ritorno di un metodo di una sottoclasse deve essere uguale o un sottotipo di quello della superclasse.

> **Principio** (single responsibility)**.** Ogni classe deve avere una singola responsabilità! Molti la possono usare, ma solo un'istituzione potrà modificarla.

La responsabilità in questione dovrà essere interamente incapsulata al suo interno ed accessibile solamente tramite metodi setters e getters.

> **Principio** (interface segregation)**.** Meglio avere diverse interfacce specifiche al posto di una singola interfaccia generale! Nessuna classe dovrebbe essere forzata ad essere dipendente di metodi che non vengono utilizzati.

Si punta ad avere meno interfacce meglio implementate.

[1] Mar02.

3.3 Design pattern

Rendere il codice più comprensibile e chiaro aumentandone la media qualitativa è l'obiettivo che ogni sviluppatore dovrebbe seguire.

> **Definizione** (design pattern)**.** Soluzione collaudata che permette di risolvere un problema ricorrente.

Ciò si rifersisce a tutte le discipline, non solo all'ingegneria del software. S'intende dunque un modello flessibile non una soluzione rigida.

Tabella 3.1: Categorizzazione di design pattern

PURPOSE		
creational	**structural**	**behavioral**
factory method	adapter	chain of responsibility
abstract factory	bridge	command
builder	composite	iterator
prototype	decorator	mediator
singleton	facade	memento
	flyweight	observer
	proxy	state
		strategy
		template method
		visitor

La divisione basata sul dominio spartisce situazioni statiche da dinamiche. Le prime favoriscono l'ereditarietà, le seconde la composizione. Ogni DP (*Design Pattern*[2]) individua un problema ricorrente e ne fornisce una soluzione.

[2] Gur14i.

3.3.1 Pattern creazionali

Astraggono il meccanismo di creazione delle istanze andando a rendere il sistema indipendente dai suoi oggetti. Semplificano la composizione di classi e la favoriscono rispetto all'uso dell'ereditarietà.

3.3.1.1 Factory method

Fornisce un'interfaccia per la creazione di oggetti in una superclasse, ma permette alle sottoclassi che la estendono di modificare il tipo dei prodotti generati.

> **Esempio** (factory method)**.** Immaginiamo di avere una ditta di logistica su terra. Le attività vanno alla grande e vogliamo estenderci sul trasporto via mare. Ciò può essere fatto con semplicità solamente se si aveva predisposto un DP factory method[a].
>
> Figura 3.5: Logistica con factory method
>
> L'astrazione ***Logistics*** fornirà un metodo `createTransport()` riadattabile dalle concretizzazioni che la implementano.
>
> [a] Gur14k.

La generalizzazione sta quindi nel disporre un'interfaccia che, tramite le sue implementazioni, genera prodotti specifici.

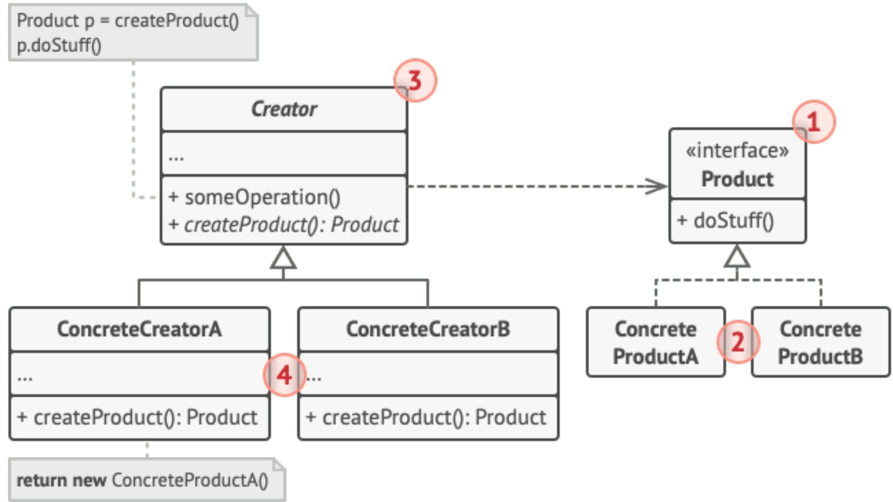


Figura 3.6: Struttura del factory method

Le sottoclassi dell'entità ***Creator*** generano dei prodotti che vengono indicizzati nell'interfaccia `Product` e nelle sue implementazioni. Ciò è utile per oggetti non definiti a priori o, come visto, nel caso in cui si intenda estendere i componenti in futuro.

3.3.1.2 Abstract factory

Definisce un'interfaccia per la creazione di famiglie di oggetti correlati senza specificare quali siano le rispettive concretizzazioni.

> **Esempio** (abstract factory)**.** Immaginiamo di dover riadderare casa, per selezionare i mobili con gusto bisogna innanzitutto scegliere uno stile e mantenerlo con coerenza. Ciò viene facilitato da un DP abstract factory[a].
>
> Figura 3.7: Arredamento con abstract factory
>
> L'interfaccia ***FornitureFactory*** definisce metodi implementati da fabbriche di famiglie caratterizzate.
>
> [a] Gur14a.

Diversi gruppi di oggetti omogenei vengono dunque generati implementando la stessa interfaccia.

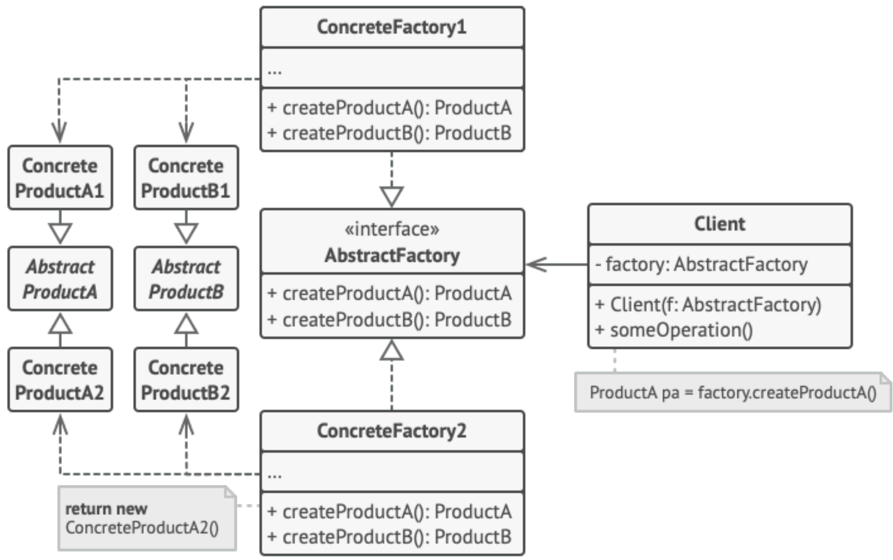


Figura 3.8: Struttura dell'abstract factory

Sarà dunque il `Client` a selezionare la fabbrica di costruzione del prodotto garantendo coerenza ed evitando incongruenze. L'abstract factory va utilizzato proprio quando si deve lavorare con famiglie di prodotti correlati senza però dover dipendere da classi concrete.

3.3.1.3 Builder

Separa la costruzione di un oggetto complesso dalla sua rappresentazione in modo che lo stesso processo di costruzione possa generare rappresentazioni differenti.

Esempio (builder)**.** Immaginiamo di dirigere una ditta edile. Le richieste riguardo le case possono essere numerose: con giardino, piscina, numero stanze da letto e bagni, o ancora con garage. Per sviluppare un software flessibile è possibile usare un DP builder[a].

Figura 3.9: Costruzione case con builder

La casa viene costruita pezzo per pezzo secondo le necessità specificate.

[a] Gur14d.

Questo garantisce riusabilità, ma soprattutto flessibilità.

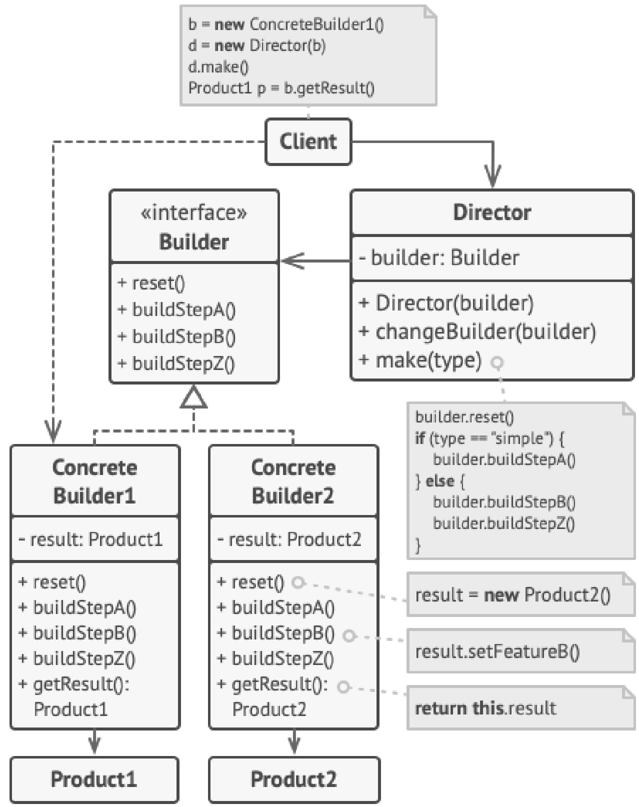


Figura 3.10: Struttura del builder

Un direttore assegna costruttori al completamento del prodotto, esiste una separazione dei ruoli.

3.3.1.4 Prototype

Consente la copia di oggetti esistenti senza rendere il codice dipendente alle loro classi.

Esempio (prototype). Immaginiamo di voler costruire un aereoplano uguale ad uno che si ha già. Per fare ciò non è plausibile iniziare a copiare passo passo tutte le componenti; viene in aiuto il DP prototype[a].

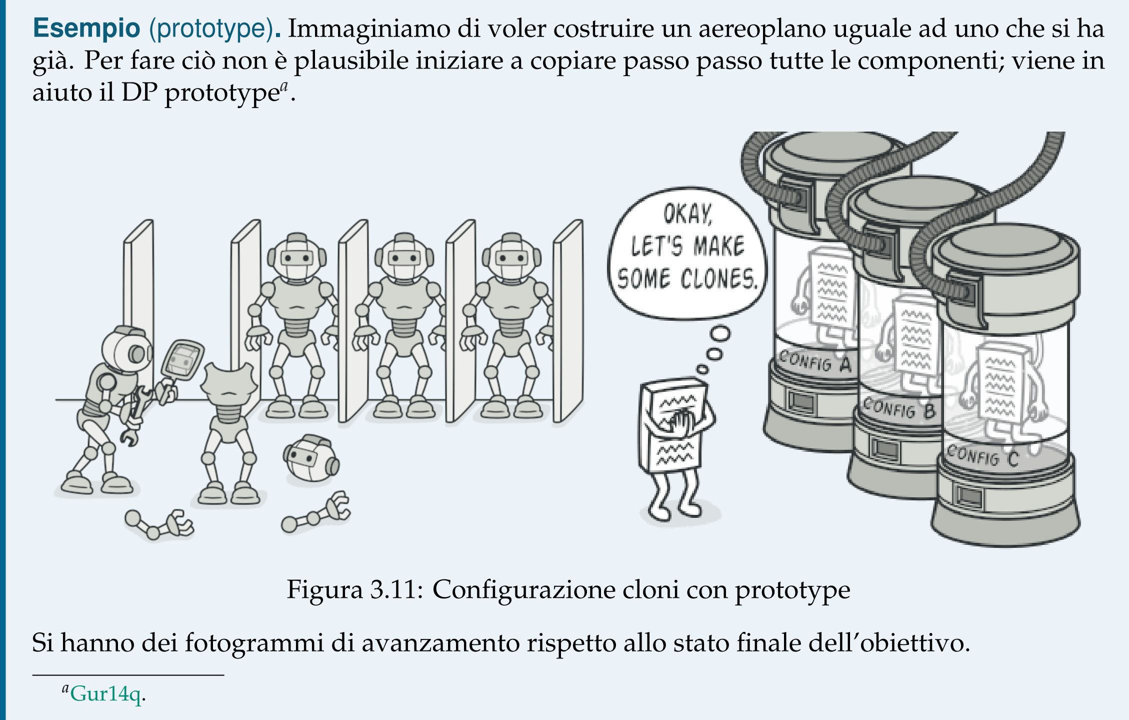


Figura 3.11: Configurazione cloni con prototype

Si hanno dei fotogrammi di avanzamento rispetto allo stato finale dell'obiettivo.

[a] Gur14q.

In particolare, ne è consigliato l'utilizzo quando si vuole semplificare la creazione di oggetti complessi sfruttando copie di prototipi già pronti.

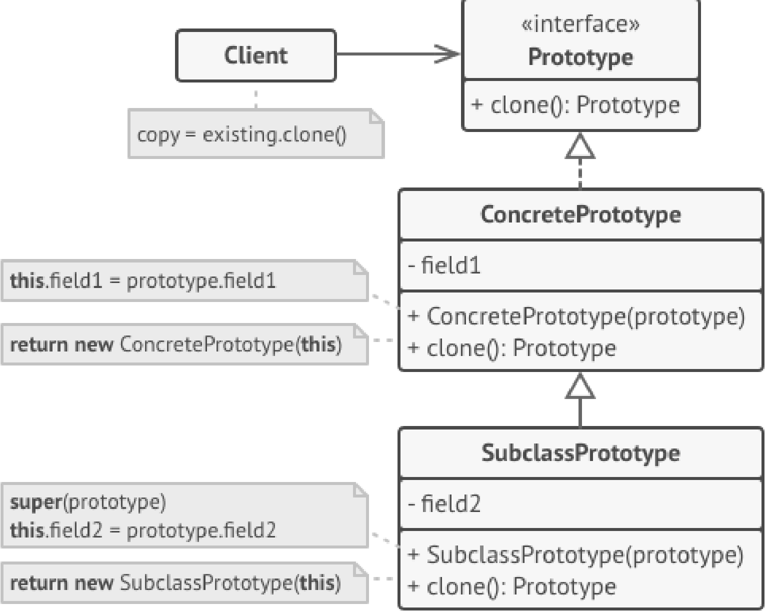


Figura 3.12: Struttura del prototype

Il numero di sottoclassi che differiscono solamente in stato d'iniziazione degli oggetti viene ridotto.

3.3.1.5 Singleton

Si assicura che una classe abbia una, e una sola, istanza consentendone al contempo un accesso globale. Ciò viola naturalmente l'SRP!

Esempio (singleton)**.** Immaginiamo di voler creare un nuovo oggetto a partire da uno con cui si è già interagito. Con altri DP risulterebbe impossibile ottenere tale comportamento, ma il singleton[a] lo permette andando a restituire un secondo accesso, o vista, a quello già presente.

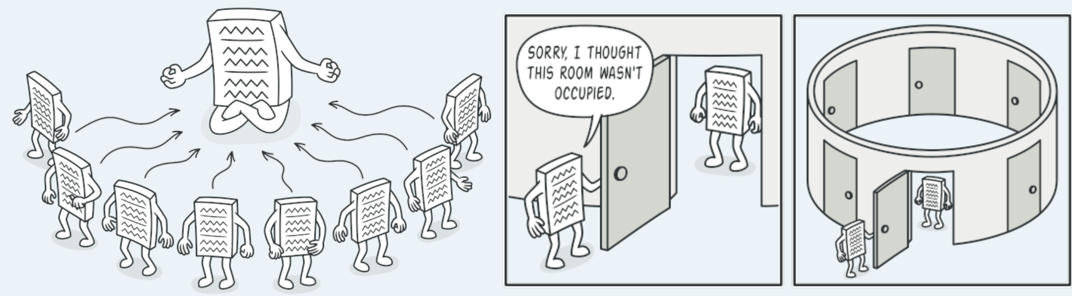


Figura 3.13: Costruzione case con singleton

Si rende il costruttore privato per prevenire l'uso dell'operatore `new` da parte di altri oggetti sulla classe. Inoltre, viene creato un metodo statico, che si finge da costruttore andandolo poi a richiamare creando un oggetto statico.

[a] Gur14s.

Il controllo sulle variabili globali è certamente ristretto: nulla oltre la classe singleton può rimpiazzare un'istanza presente.

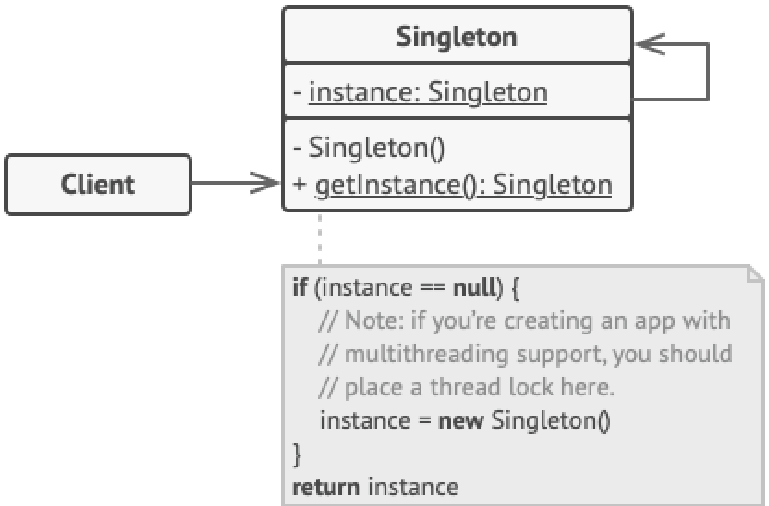


Figura 3.14: Struttura del singleton

L'accesso alla singola istanza è globale e viene fornito dal metodo `getInstance()`.

3.3.2 Pattern strutturali

Viene gestita la separazione fra interfacce e relativa implementazione.

3.3.2.1 Adapter

Permette la collaborazione di oggetti con interfacce incompatibili.

Esempio (adapter)**.** Immaginiamo di sviluppare un applicativo che consente di monitorare il mercato delle azioni. Esso scaricherà i dati in formato XML e poi li adatterà in dei grafi comprensibili all'utente. La libreria responsabile di ciò accetta però solo il formato JSON! Per ovviare si può sfruttare il DP adapter[a] che convertirà le interfacce per farle interagire.

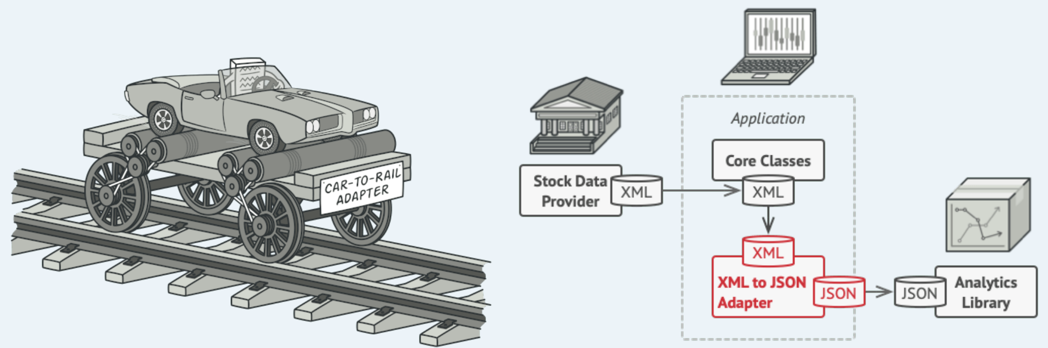


Figura 3.15: Conversione XML in JSON con adapter

Si mantiene dunque il sistema consolidato in XML, andando però a convertire i dati in JSON per la sola visualizzazione.

[a] Gur14b.

Si usa la composizione di oggetti andando ad implementare l'interfaccia di uno per renderla adattabile al secondo.

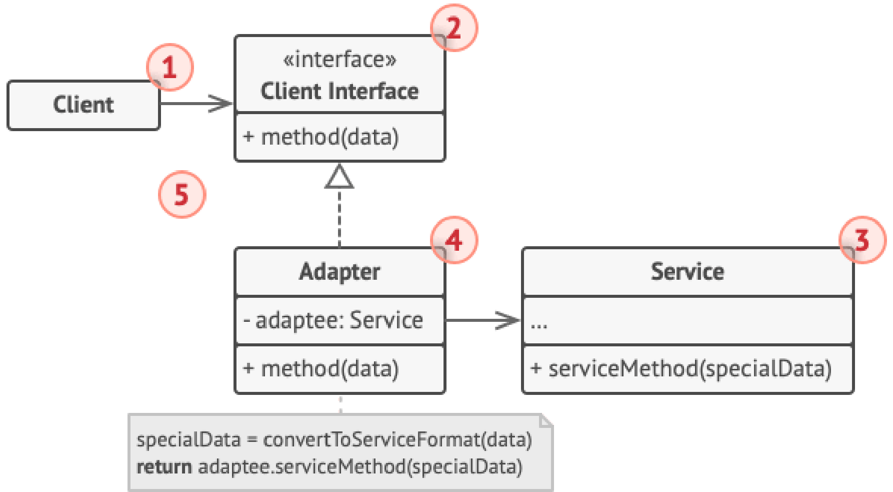


Figura 3.16: Struttura dell'adapter

Ciò può avvenire anche con le classi. Questa tecnica rispetta il SRP e l'OCP.

3.3.2.2 Bridge

Consente di separare una classe corposa, o un insieme di classi, in due gerarchie correlate: astrazioni e implementazioni.

> **Esempio** (bridge). Immaginiamo di voler estendere una classe `Shape` per introdurre i colori. Per fare ciò dovremmo inserire tutte le combinazioni. Invece, grazie al DP bridge[a], è possibile separare le tematiche forma e colore.
>
> Figura 3.17: Colorare le forme con bridge
>
> Così facendo si hanno due contesti ben divisi, ma che possono entrare in relazione.
>
> [a] Gur14c.

Un'altra idea di utilizzo sta nella separazione backend e frontend.

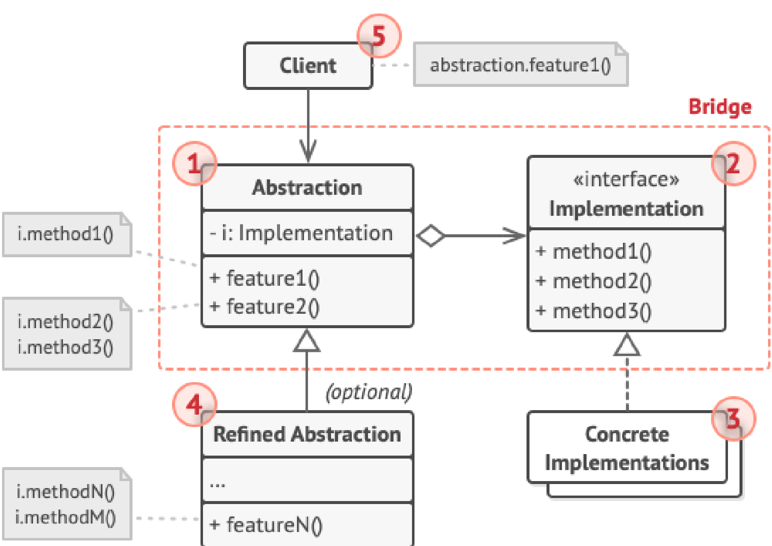
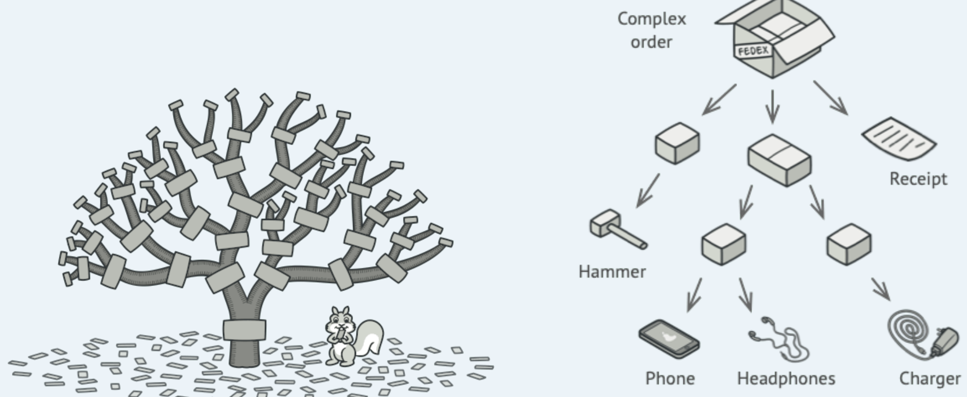


Figura 3.18: Struttura del bridge

Questo DP consente di evitare classi monolitiche che, per forza di cose, non rispetterebbero il SRP. Infine, le applicazioni create saranno indipendenti dalla piattaforma.

3.3.2.3 Composite

Riesce a comporre oggetti in strutture ad albero per poi lavorare con esse come fossero entità proprie.

> **Esempio** (composite). Considerariamo di dover calcolare il prezzo totale di un ordine contenente più prodotti già inscatolato. Essi saranno impacchettati singolarmente per poi essere riuniti in una singola confezione più grande. Per riuscire ad ottenere il dato desiderato si sfrutta il DP composite[a].
>
> 
>
>
> Figura 3.19: Calcolare il prezzo totale con composite
>
> Ciò è benefico poiché sarà indifferente sapere se l'oggetto osservato è un prodotto o un contenitore.
>
> [a] Gur14g.

Ottimo per strutturare formazioni simili ad alberi. E viene rispetta l'OCP.

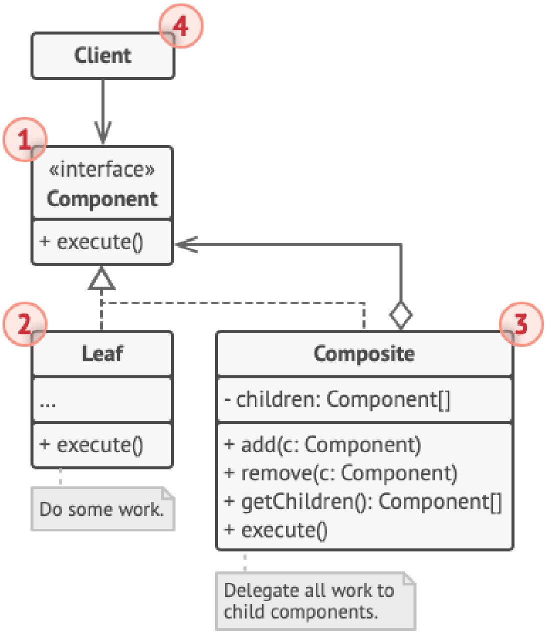


Figura 3.20: Struttura del composite

La generalizzazione potrebbe risultare in un accomunamento di entità troppo differenti.

3.3.2.4 Decorator

Annette nuovi comportamenti ad oggetti piazzandoli dentro a contenitori appositi.

> **Esempio** (decorator). Analizziamo lo sviluppo di una libreria per la ricezione di notifiche da diversi apparati. Per impedire la formazione di numerosissime combinazioni possibili si adopera il DP decorator[a] che le riunisce sotto un'unica avvertenza.
>
> Figura 3.21: Ricezione notifiche con decorator
>
> Si riuniscono quindi diverse notifiche in un insieme di decoratori.
>
> [a] Gur14h.

Questo DP è comodo quando si vogliono aggiungere comportamenti agli oggetti. Viene infatti strutturata una logica a strati componibili.

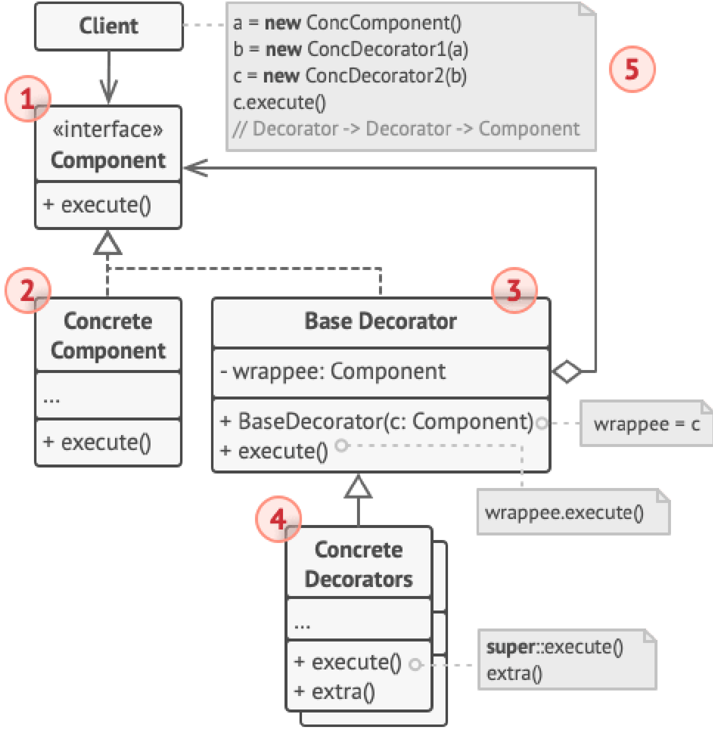
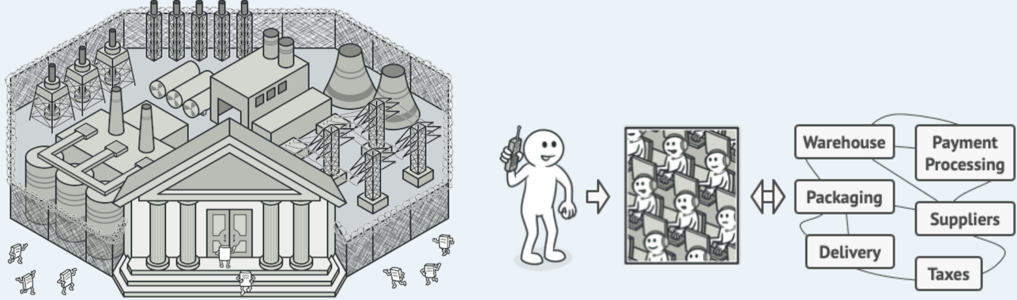


Figura 3.22: Struttura del decorator

Tale tecnica è sostitutiva dell'ereditarietà laddove l'applicazione di quest'ultima sia strana. Infine, le classi `final` non sono estendibili se non con questo DP.

3.3.2.5 Facade

Fornisce un'interfaccia semplificata che consente l'accesso ad una libreria, un framework o insiemi di classi complesse.

> **Esempio** (facade). Se l'implementazione richiede l'intenterazione con insiemi di oggetti che appartengono ad una libreria sofisticata risulta arduo andare a tener traccia di tutte le inizializzazioni e dipendenze. Per far ciò viene in soccorso il DP facade[a] che costruisce una classe la quale fornisce una semplice ed intuitiva interfaccia al sistema complesso sottostante.
>
> 
>
>
> Figura 3.23: Centro assistenza con facade
>
> Si incorre certamente in delle limitazioni, ma spesso le funzionalità principali sono anche le uniche utilizzate.
>
> [a] Gur14j.

L'ideale sta nell'adoperarlo quando si vuole integrare una libreria, ma si necessita solamente di alcune delle sue funzionalità.

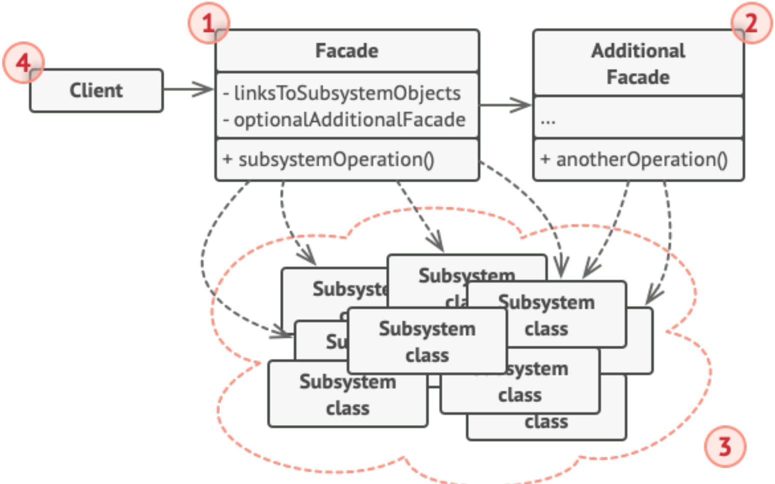


Figura 3.24: Struttura del facade

Risulta immediata la separazione del codice fra sottosistema complicato e interfaccia intuitiva.

3.3.2.6 Flyweight

Permette di adagiare più oggetti nello spazio disponibile di RAM condividendo porzioni comuni.

Esempio (flyweight)**.** Si voglia realizzare un videogioco, per la precisione uno sparatutto con dettagli alti. Ciò risulta sicuramente energivoro e, occupando totalmente la RAM, il gioco andrà in crash. Viene dunque adoperato il DP flyweight[a] che raggruppa elementi ripetuti in memoria.

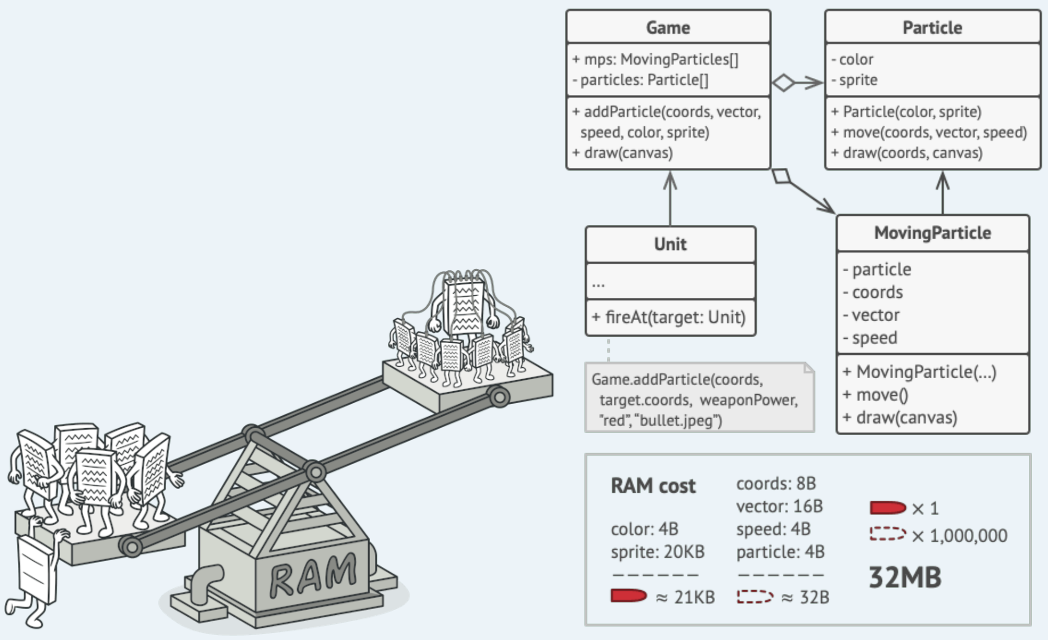


Figura 3.25: Sviluppo videogioco con flyweight

Tali dati restano costanti per ogni oggetto, poiché accessibili in sola lettura, e sono detti intrinsechi. Quelli modificabili da terzi sono invece detti estrinsechi. Sono proprio questi ultimi che dovrebbero esser salvati esternamente.

[a] Gur14l.

Dato che tale oggetto è utilizzabile in contesti differenti, bisogna accertarsi che il suo stato non possa essere modificato. Va infatti inizializzato tramite costruttore e poi non più toccato.

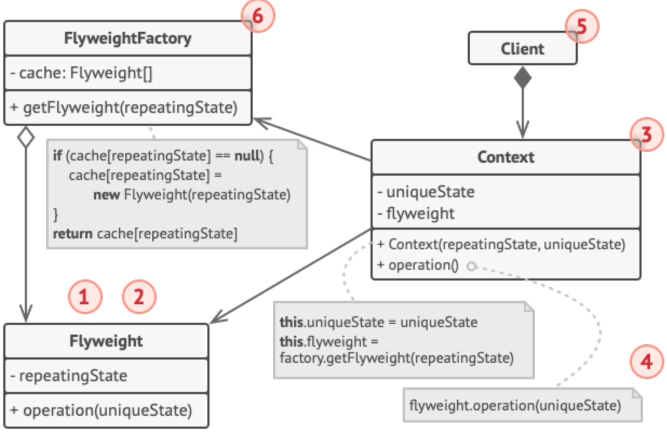


Figura 3.26: Struttura del flyweight

Lo spazio in memoria viene risparmiato, andando però ad appesantire la CPU con ulteriori cicli.

3.3.2.7 Proxy

Imposta un sostituto per un oggetto monitorandone gli accessi.

> **Esempio** (proxy). Sia dato un oggetto molto oneroso in termini di utilizzo delle risorse che viene però occasionalmente utilizzato. Grazie al DP proxy[a] si realizza una copia fasulla con la stessa interfaccia per ricevere tutte le comunicazioni originariamente indirizzate all'oggetto in questione.
>
> Figura 3.27: Delegazione lavoro con proxy
>
> Il vantaggio sta nell'ottenuta possibilità di eseguire aspetti prima o dopo della logica primaria della classe.
>
> [a] Gur14r.

La classe proxy può venire tranquillamente passata come quella base senza destare sospetti. Ciò garantisce l'OCP ritardando lievemente i tempi di risposta.

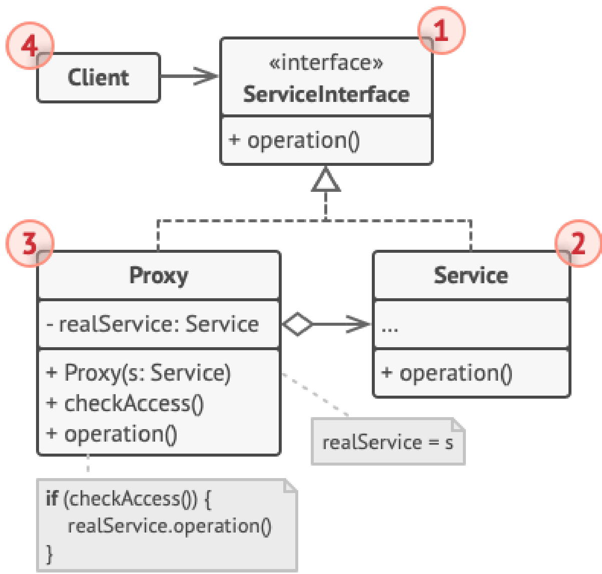


Figura 3.28: Struttura del proxy

Infine, come strato intermedio riesce a filtrare gli accessi all'oggetto.

3.3.3 Pattern comportamentali

Regolano la comunicazione efficace e l'assegnazione delle responsabilità fra gli oggetti.

3.3.3.1 Chain of responsibility

Fa passare una richiesta in una catena di gestori specializzati che possono decidere di processarla oppure inviarla al successivo.

Esempio (chain of responsibility). Volendo sviluppare un ecommerce si intende restringere l'accesso al sistema per l'invio di ordini ai soli autenticati. Inoltre, gli utenti ammistrativi devono aver accesso a tutti gli ordini. Questi controlli devono essere impostati sequenzialmente tramite il DP chain of responsibility[a].

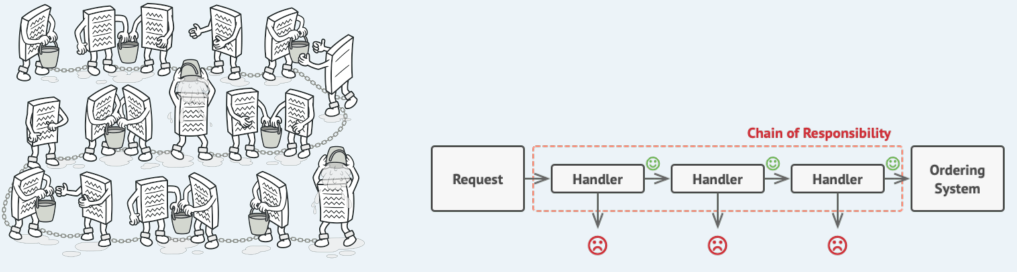


Figura 3.29: Gestione ecommerce con chain of responsibility

Si mira ad isolare i comportamenti in blocchi separati che possono decidere di inoltrare la richiesta al successivo o terminare il passaggio. Dunque, se l'accesso tramite credenziali non va a buon fine l'handler apposito cessa l'esecuzione di tutta la catena.

[a] Gur14e.

Spesso si adopera un approccio differente nel quale l'handler sviluppa la richiesta se ne è capace, altrimenti passa il compito ad un suo superiore.

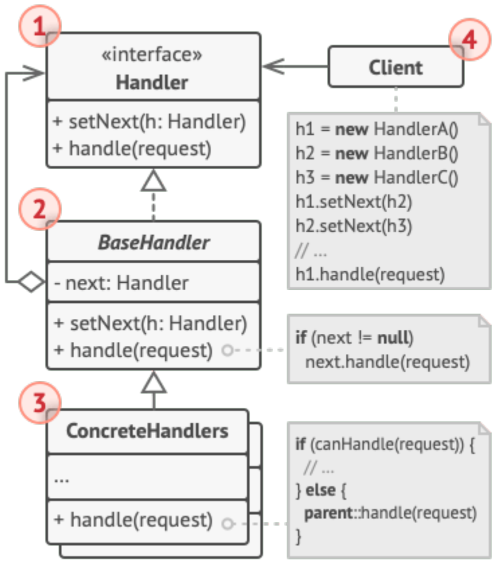
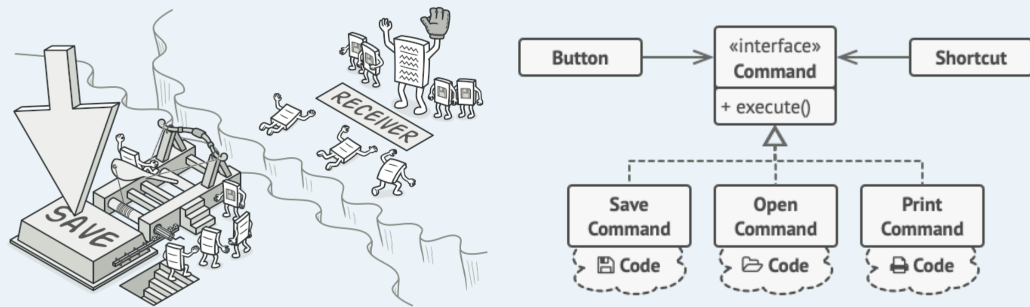


Figura 3.30: Struttura del chain of responsibility

Gli handler della catena possono essere aggiunti, rimossi e scambiati a piacimento. Il controllo tramite gestori rispetta l'SRP e l'OCP.

3.3.3.2 Command

Trasforma la richiesta in un oggetto stand-alone contenente tutte le dovute informazioni e ciò permette di passarla come argomento.

> **Esempio** (command). Costruendo un nuovo editor di testo arriverà il punto in cui si deve realizzare una barra degli strumenti con molteplici bottoni. Questi avranno stile simile, ma ognuno una funzione diversa. Anziché creare molte sottoclassi specializzate si struttura un principio di stratificazione con il DP command[a]: la GUI è comune, mentre la business logic cambia di volta in volta.
>
> 
>
>
> Figura 3.31: Definizione bottoni con command
>
> Gli elementi accomunati da un comportamento simile vengono connessi dallo stesso comando, prevenendo così duplicazione di codice.
>
> [a] Gur14f.

Si implementano operazioni reversibili, viene mantenuta infatti una cronologia degli eventi che permette di ripercorrerla.

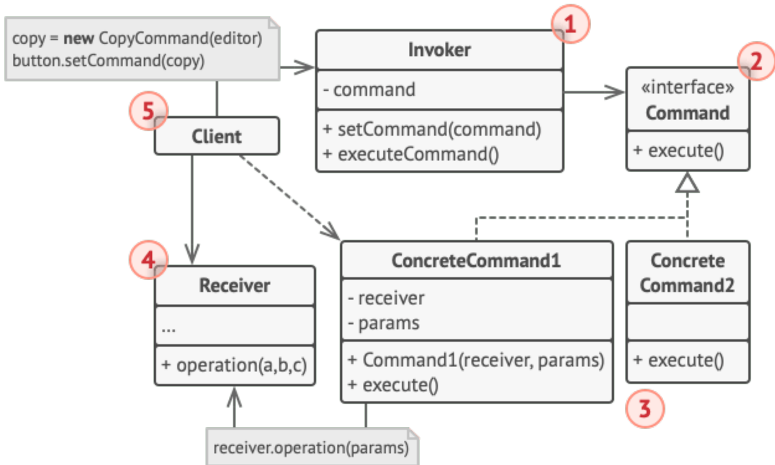


Figura 3.32: Struttura del command

Vengono rispettati l'SRP e l'OCP, e possono essere introdotte operazioni eseguite in differita.

3.3.3.3 Iterator

Consente di scorrere gli elementi di una collezione senza svelare la struttura effettiva: lista, pila, albero. . .

> **Esempio** (iterator). Volendo accedere agli oggetti contenuti in un albero si può procedere con almeno due tecniche: BFS (*Breadth First Search*) e DFS (*Depth First Search*). Si procede quindi estraendo il comportamento di una collezione in un oggetto separato tramite il DP iterator[a].
>
> Figura 3.33: Scorrere un albero con iterator
>
> Tutti gli iteratori implementano la stessa interfaccia, cosicché ogni collezione sia compatibile.
>
> [a] Gur14m.

Da applicare con collezioni complesse, potrebbe infatti essere eccessivo per strutture dati basilari.

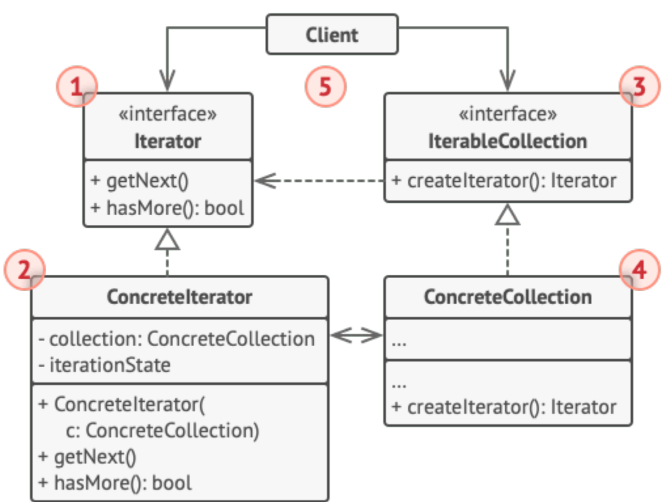


Figura 3.34: Struttura dell'iterator

Adoperando un oggetto apposito all'esaminazione di strutture dati si rispetta l'SRP e l'OCP. Inoltre, essendo un oggetto esterno, è possibile scorrere la collezione con molteplici iteratori al contempo.

3.3.3.4 Mediator

Mira a ridurre le dipendenze caotiche tra oggetti restringendo le comunicazioni forzandole a passare per uno strumento apposito.

Esempio (mediator). Supponiamo di avere una schermata per creare e modificare il profilo utente con le rispettive caselle di testo, bottoni e altri elementi. Alcune di questi sono in relazione, ad esempio selezionandone uno potrebbe apparirne un altro a comparsa. Implementando ciò direttamente si limita la riusabilità del codice, applicando invece il DP mediator[a] si rindirizzano tutte le comunicazioni all'oggetto di riferimento.

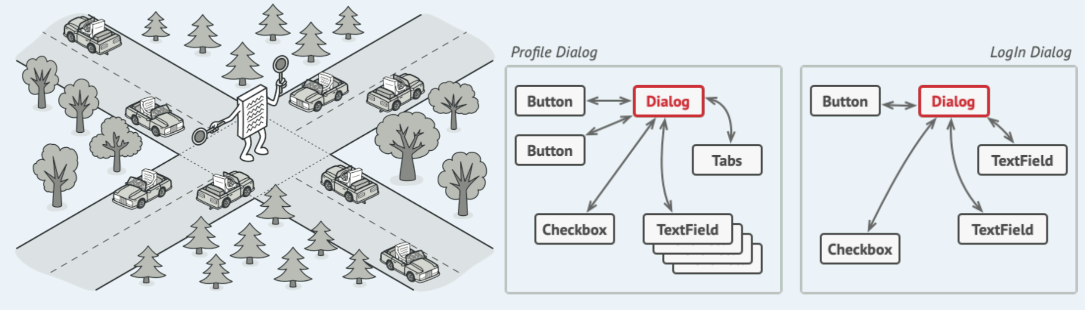


Figura 3.35: Finestre di dialogo con mediator

Così facendo, tutte le componenti dipengono solamente da un elemento.

[a] Gur14n.

Sarà poi facile stravolgere il modo in cui gli oggetti collaborano sostituendo la classe mediatrice.

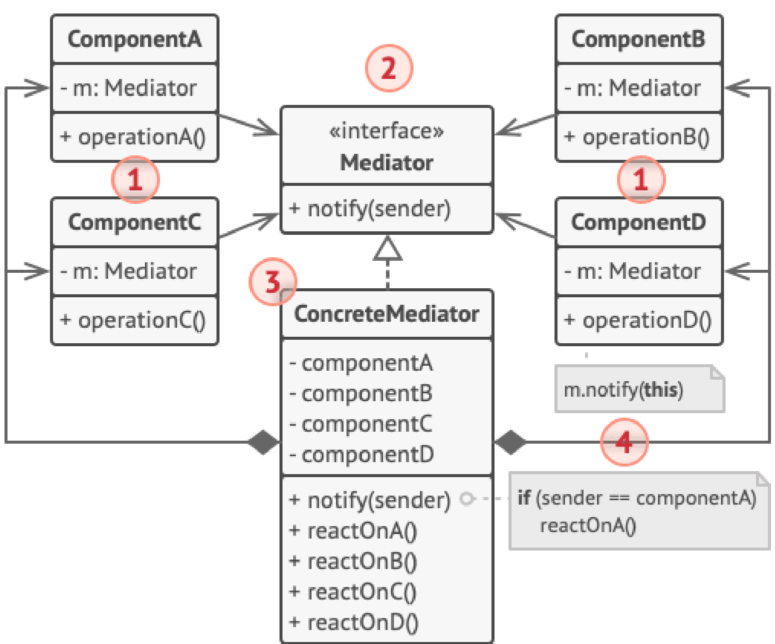


Figura 3.36: Struttura del mediator

Si ha l'SRP e l'OCP, col rischio però che la classe potrebbe divenire troppo centralizzata.

3.3.3.5 Memento

Consente di salvare e ripristinare lo stato di un oggetto senza rilevarne i dettagli implementativi.

> **Esempio (memento).** Realizzando un editor di testo, che permette fra le altre cose di formattare ed inserire immagini, l'utente si aspetterà di poter retrocedere le sue azioni: il comune `Ctrl+Z`. Si arriva però ad un bivio: esporre gli snapshot oppure renderli inaccessibili? Ciò si risolve col DP memento[a] che delega la creazione dei suddetti snapshot al proprietario originario.
>
> Figura 3.37: Undo nei text editor con memento
>
> I contenuti non sono accessibili agli esterni, tranne al produttore effettivo.
>
> [a] Gur14o.

Gli oggetti memorizzatori, memento, vengono contenuti in una classe definita *caretaker*. Quindi, ogni retrocessione ripristina l'elemento corretto effettuando un roll-back.

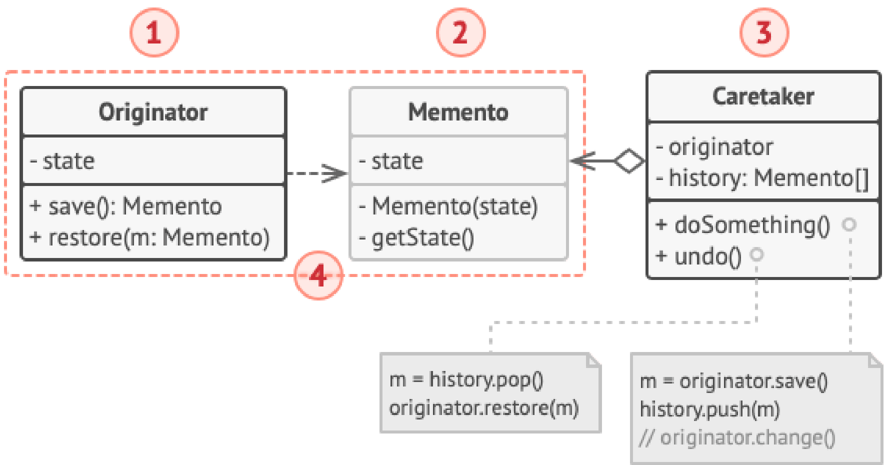


Figura 3.38: Struttura del memento

La vera rivoluzione sta nell'accedere agli snapshot senza violare l'incapsulamento.

3.3.3.6 Observer

Imposta un meccanismo per notificare più oggetti su un determinato evento avvenuto all'oggetto che tutti essi considerano.

Esempio (observer). Un negozio fisico, come uno virtuale, potrebbe desiderare di avvertire l'utente, interessato ad un nuovo prodotto, qualora esso arrivi in magazzino. È quindi una via di mezzo fra il non comunicare con la clientela e l'infastidire con posta indesiderata. Ciò si effettua con il DP observer[a] facendo direttamente iscrivere le persone all'evento futuro.

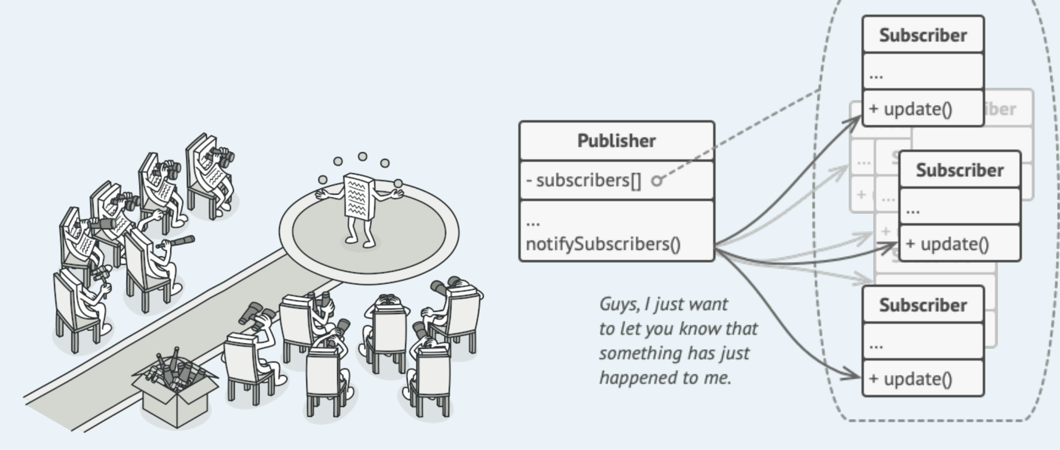


Figura 3.39: Newsletter mirata con observer

Il gestore avrà dunque una lista di iscritti a cui indirizzerà le relative informazioni.

[a] Gur14p.

L'utilizzo è pensato per oggetti che mirano le gesta di altri per un periodo limitato.

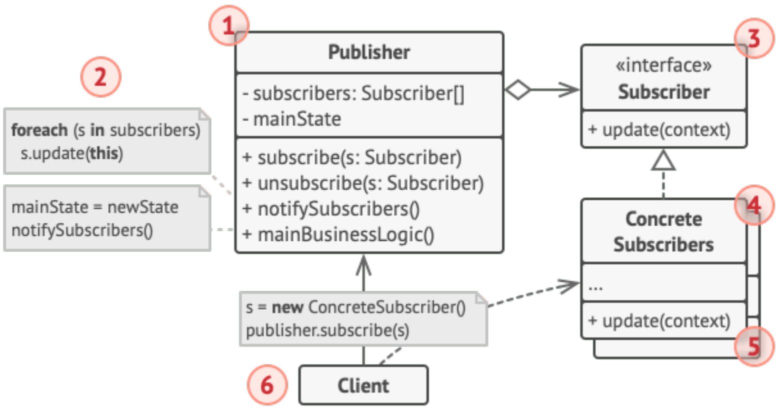
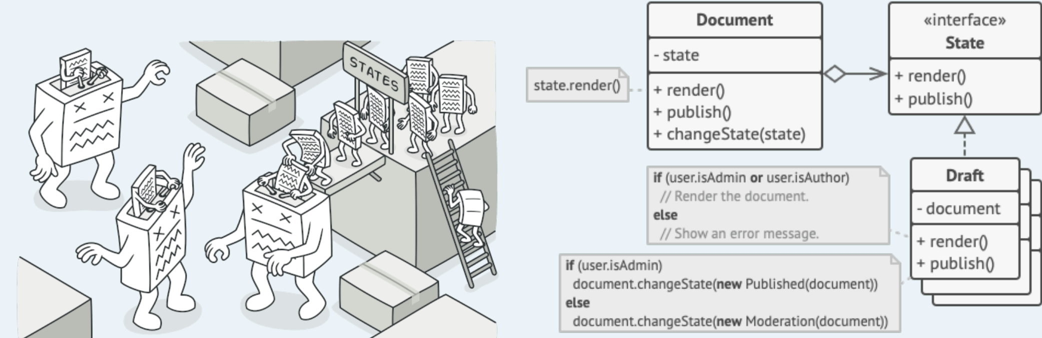


Figura 3.40: Struttura del observer

L'OCP viene rispettato ed è possibile strutturare una relazione fra oggetti a runtime; l'aspetto negativo sta nella randomicità dell'ordine di avvertenza.

3.3.3.7 State

Permette all'oggetto di modificare il suo comportamento qualora la sua condizione cambi.

> **Esempio** (state). Volendo gestire una classe `Document` per la pubblicazione di un testo, si ripetono delle azioni per contesti differenti: c'è innanzitutto il passaggio di approvazione della bozza e poi la rispettiva pubblicazione. Esiste un parallelismo con gli automi a stati finiti. Il DP state[a] suggerisce di creare entità indipendenti contenenti le informazioni relative allo stato in cui l'oggetto si trova. Quest'ultimo avrà perciò un riferimento alle prime.
>
> 
>
>
> Figura 3.41: Pubblicazione documento con state
>
> Il passaggio ad un altro stato muta dunque il riferimento all'oggetto rappresentante le nuove condizioni. Per garantire ciò gli oggetti devono relazionarsi tramite un'interfaccia comune.
>
> [a] Gur14t.

Questo sistema va adoperato qualora l'oggetto si comporta differentemente in relazione al suo stato.

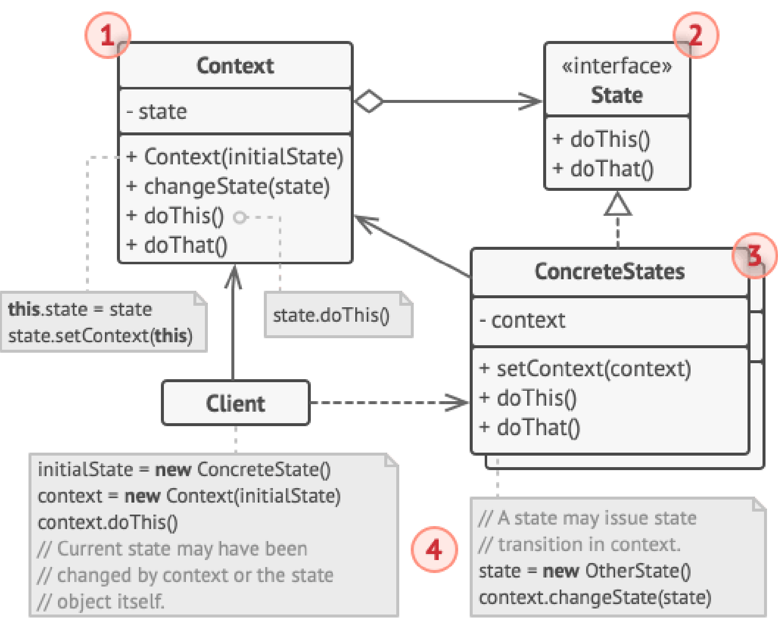


Figura 3.42: Struttura del state

L'SRP e l'OCP vengono rispettati, ed il codice semplificato.

3.3.3.8 Strategy

Definisce una famiglia di algoritmi divisi in classi e consente l'interscambiabilità dei rispettivi oggetti.

Esempio (strategy). Nello sviluppare un'app di navigazione è ovvio implementare la funzionalità per il calcolo automatico della strada più veloce. La problematica risiede sul fatto che la scelta varia per il tipo di mezzo utilizzato. Si risolve col DP strategy[a] che realizza varie classi specifiche dette appunto strategie.

Figura 3.43: Calcolo percorso con strategy

La classe principale non sceglie direttamente l'algoritmo adatto, così facendo la si mantiene indipendente dalle singole strategie concrete.

[a] Gur14u.

Le molteplici varianti dello stesso algoritmo sono un sintomo per il corretto utilizzo del pattern. Avviene inoltre una piacevole spartizione fra logica ed implementazione.

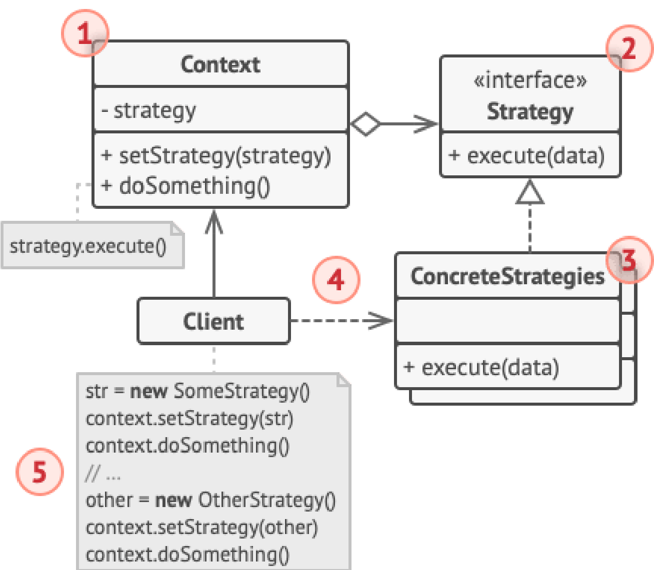


Figura 3.44: Struttura del strategy

L'OCP viene garantito, e permette la sostituzione dell'ereditarietà a favore della composizione.

3.3.3.9 Template method

Definisce lo scheletro di un algoritmo in una superclasse, ma permette alle rispettive sottoclassi di ereditare passi di esso senza cambiarne la struttura.

> **Esempio** (template method)**.** Un'applicazione per il data mining analizza i documenti di un'azienda. Questi sono in vari formati: PDF, DOC, CSV. . . Dunque il codice è stato esteso per accettarli tutti, ma ci si è resi conto che molte parti erano duplicate. La soluzione arriva col DP template method[a] che sollecita a spezzettare l'algoritmo in parti, potendo quindi poi andare a sostituire sezioni di esso a piacimento.
>
> Figura 3.45: Analisi dati con template method
>
> Si vanno quindi a rimplementare i metodi necessari specializzandoli relativamente alla sottoclasse.
>
> [a] Gur14v.

Da notare è che se nella superclasse si inseriscono metodi astratti, essi andranno ridefiniti in ogni sottoclasse.

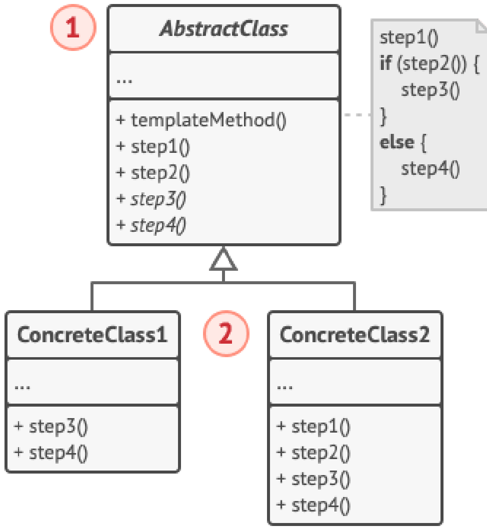
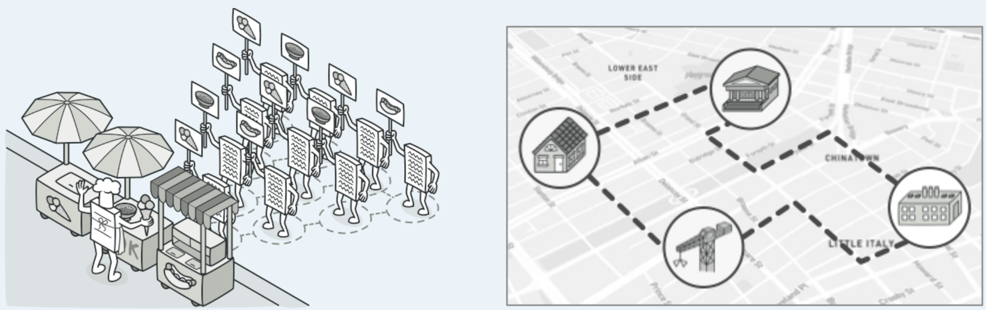


Figura 3.46: Struttura del template method

Il codice inizialmente duplicato viene rimosso e la versatilità dell'algoritmo spicca. Maggiori i passi di quest'ultimo, più arduo il mantenimento. Infine il LSP potrebbe esser violato.

3.3.3.10 Visitor

Separa gli algoritmi dagli oggetti su cui essi operano.

> **Esempio** (visitor). Si intende realizzare un'app che contenga una mappa geografica strutturata a grafo con molteplici livelli di granularità. Il tutto potrà poi essere esportato in XML. L'idea iniziale era quella di aggiungere un metodo di esportazione a ciascuna classe di nodo ed utilizzare la ricorsione per attraversare il grafo. Il DP visitor[a] permette di aggiungere nuove operazioni a una gerarchia di classi senza modificare quelle esistenti.
>
> 
>
>
> Figura 3.47: Esportazione mappa con visitor
>
> L'idea alla base è il *double dispatch*, ossia il delegare la scelta dell'operazione specifica agli oggetti che vengono visitati. Gli oggetti stessi conoscono il proprio tipo e possono selezionare il metodo appropriato.
>
> [a] Gur14w.

Viene adoperato quando occorre effettuare la stessa operazione su diversi elementi di una struttura di oggetti complessa.

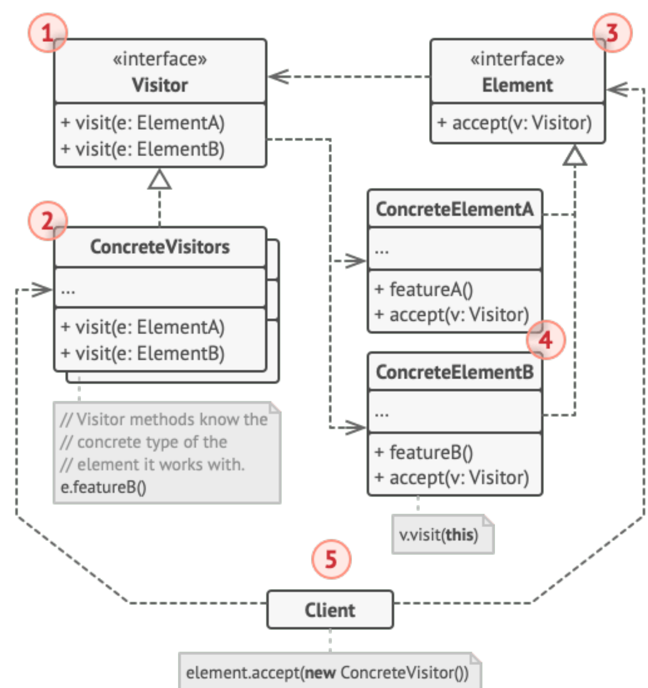


Figura 3.48: Struttura del visitor

Si rispettano l'OCP e l'SRP, ma occorre aggiornare ogni volta che una classe viene aggiunta o rimossa dalla gerarchia.

Sitografia

[Gur14a] Refactoring Guru. Abstract factory. 2014.
URL: https://refactoring.guru/design-patterns/abstract-factory.

[Gur14b] Refactoring Guru. Adapter. 2014.
URL: https://refactoring.guru/design-patterns/adapter.

[Gur14c] Refactoring Guru. Bridge. 2014.
URL: https://refactoring.guru/design-patterns/bridge.

[Gur14d] Refactoring Guru. Builder. 2014.
URL: https://refactoring.guru/design-patterns/builder.

[Gur14e] Refactoring Guru. Chain of responsibility. 2014.
URL: https://refactoring.guru/design-patterns/chain-of-responsibility.

[Gur14f] Refactoring Guru. Command. 2014.
URL: https://refactoring.guru/design-patterns/command.

[Gur14g] Refactoring Guru. Composite. 2014.
URL: https://refactoring.guru/design-patterns/composite.

[Gur14h] Refactoring Guru. Decorator. 2014.
URL: https://refactoring.guru/design-patterns/decorator.

[Gur14i] Refactoring Guru. Design patterns. 2014.
URL: https://refactoring.guru/design-patterns.

[Gur14j] Refactoring Guru. Facade. 2014.
URL: https://refactoring.guru/design-patterns/facade.

[Gur14k] Refactoring Guru. Factory method. 2014.
URL: https://refactoring.guru/design-patterns/factory-method.

[Gur14l] Refactoring Guru. Flyweight. 2014.
URL: https://refactoring.guru/design-patterns/flyweight.

[Gur14m] Refactoring Guru. Iterator. 2014.
URL: https://refactoring.guru/design-patterns/iterator.

[Gur14n] Refactoring Guru. Mediator. 2014.
URL: https://refactoring.guru/design-patterns/mediator.

[Gur14o] Refactoring Guru. Memento. 2014.
URL: https://refactoring.guru/design-patterns/memento.

[Gur14p] Refactoring Guru. Observer. 2014.
URL: https://refactoring.guru/design-patterns/observer.

[Gur14q] Refactoring Guru. Prototype. 2014.
URL: https://refactoring.guru/design-patterns/prototype.

[Gur14r] Refactoring Guru. Proxy. 2014.
URL: https://refactoring.guru/design-patterns/proxy.

[Gur14s] Refactoring Guru. Singleton. 2014.
URL: https://refactoring.guru/design-patterns/singleton.

[Gur14t] Refactoring Guru. State. 2014.
URL: https://refactoring.guru/design-patterns/state.

[Gur14u] Refactoring Guru. Strategy. 2014.
URL: https://refactoring.guru/design-patterns/strategy.

[Gur14v] Refactoring Guru. Template method. 2014.
URL: https://refactoring.guru/design-patterns/template-method.

[Gur14w] Refactoring Guru. Visitor. 2014.
URL: https://refactoring.guru/design-patterns/visitor.

Bibliografia

[AN06] Jim Arlow and Ila Neustadt. UML e Unified Process. 2nd ed. McGraw-Hill, 2006. ISBN: 9788838661440.

[CS14] Scott Chacon and Ben Straub. Pro Git. 2nd ed. Apress, 2014.

[Lar16] Craig Larman. Applicare UML e i pattern. Pearson, 2016. ISBN: 9788891904591.

[LLL18] Michael Lewrick, Patrick Link, and Larry Leifer. Manuale di design thinking. 1st ed. LSRW, 2018. ISBN: 9788868956738.

[Mar02] Robert Martin. Agile Software Development. 1st ed. Pearson, 2002. ISBN: 9780135974445.

[Poh10] Klaus Pohl. Requirements engineering. 1st ed. Springer Verlag, 2010. ISBN: 9783642125775.