

Progettazione e Analisi Object Oriented Appunti

Andrea Mansi UniUD

II° Semestre 2021/2022

Contents

1	Introduzione al concetto di Object Oriented	3
1.1	Tipi di dato astratto	3
1.2	Scambio messaggi	3
1.3	Ereditarietà	4
1.4	Polimorfismo	5
1.5	Object Oriented Design e Analysis	6
2	UML	7
2.1	Diagrammi delle classi	7
2.2	Diagrammi dei package	8
2.3	Diagramma degli oggetti	8
2.4	Diagramma dei componenti	9
2.5	Diagramma di dispiegamento	10
2.6	Diagramma dei casi d'uso	10
2.7	Diagrammi di sequenza	11
2.8	Diagrammi di collaborazione	11
2.9	Diagrammi degli stati	12
2.10	Diagramma delle attività	13
2.11	Timing Diagrams	13
2.12	Considerazioni sull'uso di UML	14
3	UML per il progetto OO: diagrammi di classe	16
3.1	Relazioni	17
4	UML di qualità: linee guida, elementi di stile e colore	23

5 Principi di Progetto Object Oriented	29
5.1 Incapsulamento	29
5.2 Dipendenza	30
5.3 Domini	31
5.4 Ingombro	32
5.5 Legge di Demeter	33
5.6 Coesione	34
5.7 Spazio degli stati	35
5.8 Transizioni e comportamento	36
5.9 Asserzioni	36
5.10 Progetto per contratto	37
5.11 Conformità di tipo	38
5.12 Comportamento chiuso	38
5.13 Genericità	39
5.14 Anelli di operazioni	39
5.15 Tipologie di stati e transizioni di un'interfaccia	40
5.16 S.O.L.I.D.	41
5.17 Frameworks	44
6 Design patterns	45
6.1 8 Problemi evitabili con i pattern	116
6.2 Cenni ai Pattern Architetturali	118
7 Refactoring	120
7.1 Composizione di Metodi	121
7.2 Spostamenti fra Oggetti	124
7.3 Organizzazione dei Dati	127
7.4 Espressioni Condizionali	132
7.5 Invocazione di Metodi	137
7.6 Gestione della Generalizzazione	141
7.7 Bad Smells	143
7.8 Big Refactorings	146
7.9 JUnit	148
7.10 Profiling	148
8 eXtreme Programming	149
8.1 Agile Methods	151
9 Analisi Orientata agli Oggetti - OOA	154
9.1 Pattern di Analisi	156
10 Sistemi Multi Agente	160

1 Introduzione al concetto di Object Oriented

Iniziamo definendo il concetto di Object Oriented, analizzando i principali concetti cardine del suddetto paradigma.

1.1 Tipi di dato astratto

Un TDA è un nuovo tipo definito dal programmatore ed è composto dai suoi valori e dalle sue operazioni. Viene chiamato "astratto" perché è una sorta di interfaccia che astraee/separa dall'implementazione.

Con scomposizione per TDA si intende raggruppare codice in un nuovo tipo; individuando i tipi di dato che semplificherebbero la scrittura del programma. Si definiscono quindi nuovi tipi (interfaccia e implementazione) per poi utilizzarli astraendo dalla loro implementazione. Si contrappone alla scomposizione funzionale/procedurale, che consiste nel raggruppare istruzioni in una funzione che però non è un approccio che si adatta ai cambiamenti e al riuso.

Esempio di TDA: punto, cerchio, segmento, etc. (in un programma di grafica).

La definizione di un TDA è un esempio del concetto di **incapsulamento**: si definiscono nuovi tipi "incapsulando" (racchiudendo) in modo unitario, tutto all'interno di un solo concetto/entità/"capsula"; tutto questo rendendo inaccessibile l'implementazione. Senza i TDA si incapsula all'interno delle funzioni/procedure.

Questo porta al fenomeno dell'**information hiding**, ovvero l'occultamento delle informazioni relative all'implementazione sottostante, rendendo visibile solo l'interfaccia; ovvero le funzioni che permettono di manipolare i dati del TDA. L'information hiding porta a notevoli vantaggi: modificabilità migliore (evita l'effetto domino); semplicità d'uso, coerenza, robustezza, implementazione nascosta e riuso.

Con i TDA, oltre che alla separazione tra interfaccia e implementazione (information-hiding) si ha elevata modificabilità, usabilità, robustezza, riuso ("stampo per variabili").

1.2 Scambio messaggi

Nel mondo Object Oriented, gli oggetti comunicano tramite scambio messaggi. Da esegui il metodo getX della classe Punto con argomento q...

```
1 double a = Punto.getX(q);  
... a manda il messaggio getX all'oggetto q.
```

```
1 double a = q.getX();
```

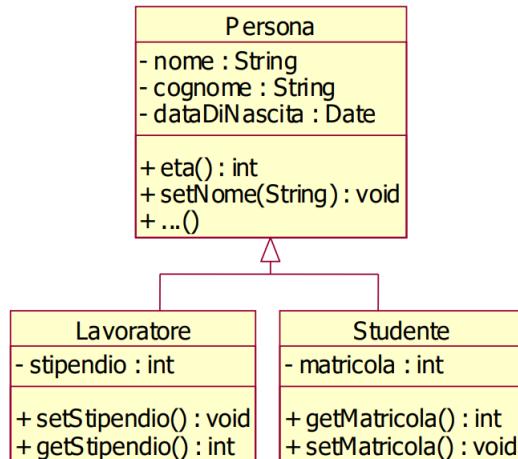
Nell'approccio TDA le variabili contengono solo lo stato, nell'approccio OO gli oggetti contengono stato e metodi associati.

TDA vs Object Oriented

Filosoficamente si parla di variabili passive contro oggetti attivi; in pratica, nel codice, si interagisce/parla con le istanze e non con le classi.

1.3 Ereditarietà

L'ereditarietà mira a risolvere il problema della duplicazione di codice; una classe specifica eredita attributi e metodi da una classe generica, eventualmente aggiungendone di altri o sovrascrivendoli. Ad esempio `Studente` eredita da `Persona`; perché gli studenti sono un sottoinsieme di `Persona`.



A tal proposito definiamo:

- **Overriding** (sovrascrittura): diversa implementazione dello stesso metodo con stesse firme;
- **Overloading** (sovraffunzione): diversa implementazione dello stesso metodo ma con firme diverse;

Esempio in JAVA:

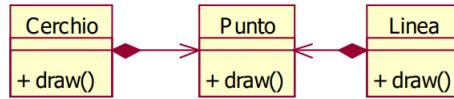
```
1 public class Processor {
2     public void process(int i, int j){
3         System.out.println("Processing two ints: %d, %d", i, j);
4     }
5
6     public void process(int[] ints){ // esempio di overloading
7         System.out.println("Processing array of ints");
8     }
9 }
10
11 // MathProcessor eredita da Processor
12 public class MathProcessor extends Processor {
13     @Override
14     public void process(int i, int j){ // esempio di overriding
15         System.out.println(i+j);
16     }
17 }
```

1.4 Polimorfismo

Nel contesto della programmazione orientata agli oggetti, con polimorfismo ci si riferisce al fatto che un'espressione il cui tipo sia descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B sottoclasse di A (polimorfismo per inclusione);

Esempio senza polimorfismo:

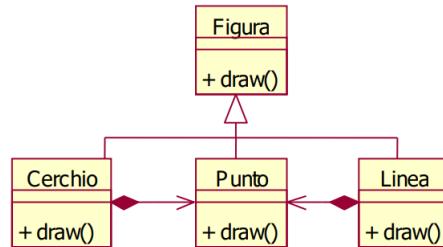
```
1 Punto[] punti;
2 Cerchio[] cerchi;
3 ...
4 if (x instanceof Punto)
5     punti[i] = x;
6 else if (x instanceof Cerchio)
7     cerchi[i] = x;
8 else if ...
9 for (int i = ...){
10     punti[i].draw();
11     cerchi[i].draw();
12 ...
13 }
```



Implementazione "sporca" e scomoda; non adatta al cambiamento (se aggiungo una forma devo modificare aggiungendo controlli e linee di codice, non è un approccio generico).

Esempio con polimorfismo:

```
1 Figura[] figure = new Figura[100];
2 figure[i] = new Punto();
3 figure[j] = new Cerchio();
4 for (int k = ...){
5     figure[k].draw();
6 }
```



È più comodo "parlare" alla classe base perché permette di generalizzare. Nel polimorfismo le variabili non contengono gli oggetti ma il riferimento agli oggetti (detto "maniglia"). Il polimorfismo si basa sullo scambio messaggi, ereditarietà e overriding.

1.5 Object Oriented Design e Analysis

Parliamo ora di Analisi orientata agli oggetti e del Design orientato agli oggetti; due attività che si svolgono prima dell’Object Oriented Programming.

- OOA = studio del problema, modellazione del problema escluso il software.
- OOD = progetto della soluzione, com’è fatto il sistema software.

Il confine tra OOD e OOA è labile, sfuocato: per fare una buona analisi bisogna pensare agli strumenti utilizzati, anche se lo studio del problema in linea di massima è separato dall’implementazione (OOD/OOP); essendo ”concettuale”. Forse ancora più sfuocato nell’Object Oriented, essendoci la pretesa di rendere ”isomorfi” lo spazio del problema e lo spazio delle soluzioni.

2 UML

In questa sezione, vedremo i principali diagrammi dell'UML 1.5 (Unified Modelling Language), ovvero:

- Classi (class)
- Package
- Oggetti (object)
- Componenti (component)
- Dispiegamento (deployment)
- Casi d'uso (use case)
- Sequenza (sequence)
- Collaborazione (collaboration)
- Stati (statechart)
- Attività (activity)

L'UML è un linguaggio di modellazione e di specifica basato sul paradigma object oriented. Viene utilizzato principalmente per le fasi di analisi e di progetto.

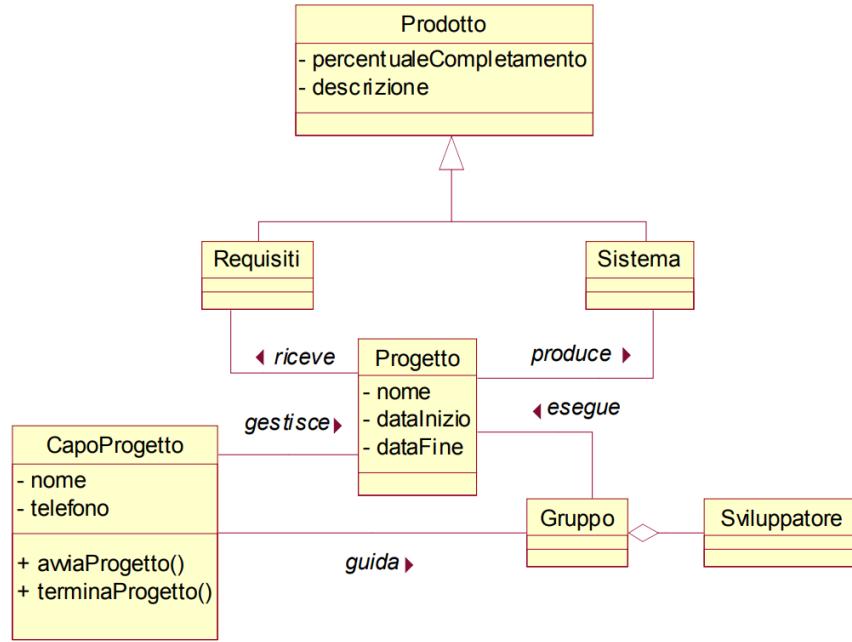
2.1 Diagrammi delle classi

I diagrammi tra classi si concentrano nel rappresentare relazioni fra classi (interfacce e package), evidenziando relazioni di:

- Dipendenza generica
- Associazione
- Aggregazione e composizione (tutto/parte)
- Generalizzazione/specializzazione (eredità)

Un esempio è il seguente:

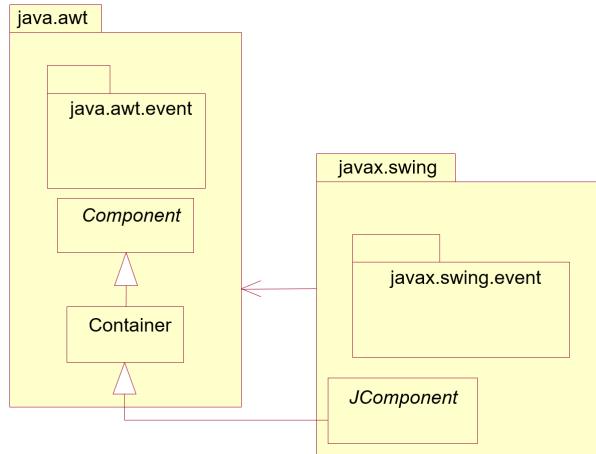
- Ogni progetto ha un nome, data di inizio e fine (attributi);
- Ogni progetto ha associati un capo progetto i cui attributi sono nome e telefono, e un gruppo di lavoro;
- Ciascun capo gestisce (avvia e termina) il progetto a cui è associato e guida il gruppo di lavoro associato al progetto;
- Il progetto riceve in input i requisiti e produce un sistema;
- Ogni gruppo è composto da più sviluppatori.



2.2 Diagrammi dei package

Può essere visto come un'estensione del diagramma delle classi in cui si utilizza il simbolo di package per raggruppare classi. Sono utili per una visione più astratta (divide et impera).

Mostrano il posizionamento delle classi e le dipendenze tra package.

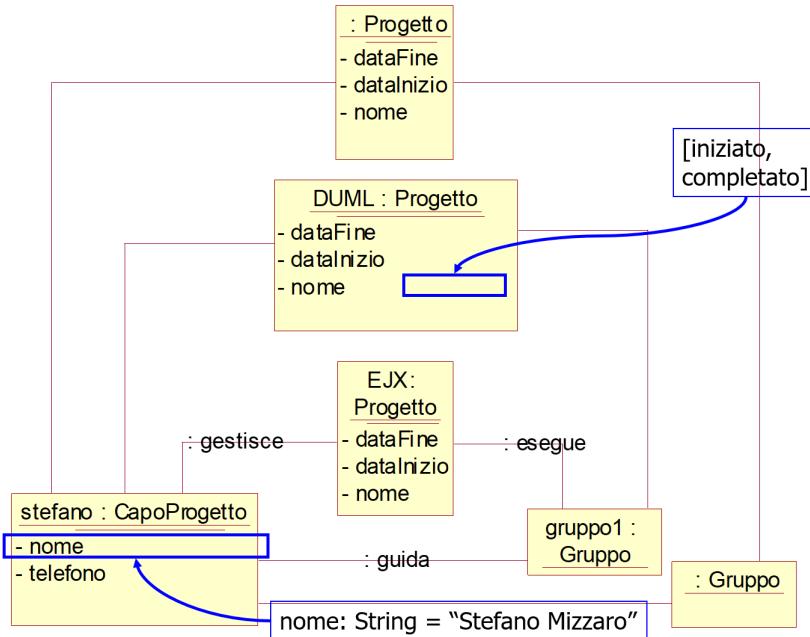


2.3 Diagramma degli oggetti

Gli oggetti rappresentati sono istanze; il diagramma è simile a quello delle classi ma abbiamo:

- Le istanze al posto delle classi (nomeIstanza: nomeClasse);
- Link al posto di associazioni;
- Valori e tipi degli attributi, stato dell'istanza.

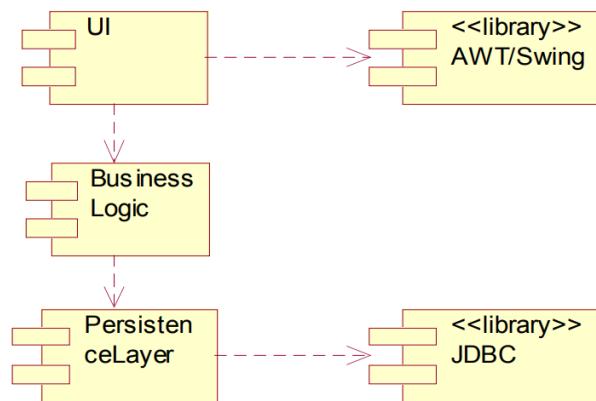
Un diagramma degli oggetti può essere visto come un’istantanea della struttura del sistema a un certo istante dell’esecuzione.



2.4 Diagramma dei componenti

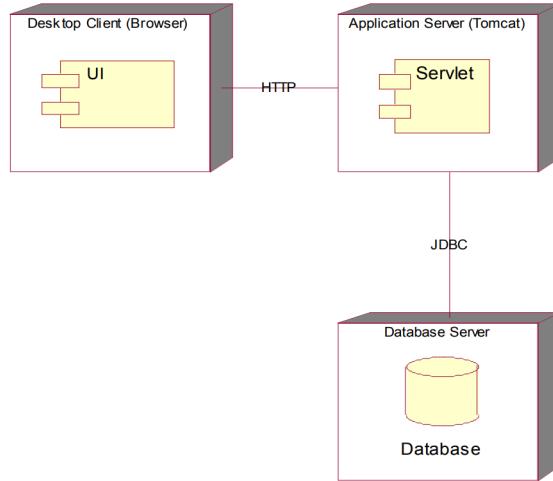
Con componente si intende un ”pezzo fisico” del sistema, parte del sistema che esiste durante l’esecuzione, unità fisica di codice, ad esempio: eseguibile .exe, libreria .dll, file compilato, tabella di database, file etc.

Rappresenta un modello dell’implementazione del sistema, evidenziando dipendenze tra i componenti.



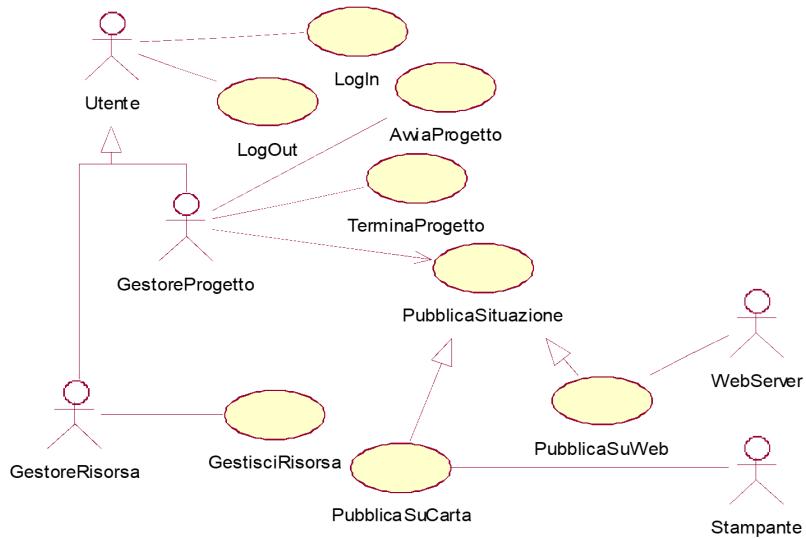
2.5 Diagramma di dispiegamento

Mostrano dove risiedono fisicamente i vari componenti del sistema, ad esempio in un sistema distribuito o un sistema client/server. Contengono nodi e comunicazioni, componenti e dipendenze.



2.6 Diagramma dei casi d'uso

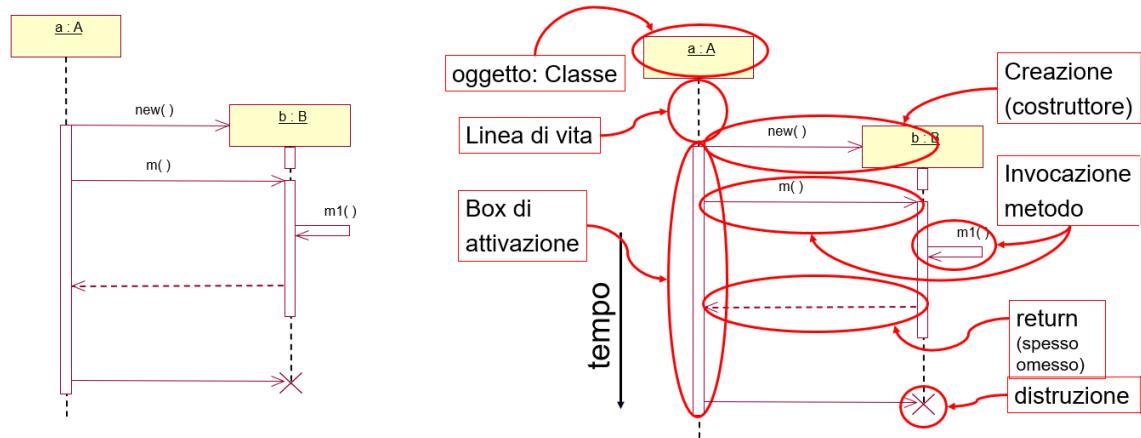
I diagrammi dei casi d'uso sono più "dinamici", catturano i requisiti (analisi). Un caso d'uso è un documento che descrive una specifica funzionalità fornita dal sistema. Le componenti presenti sono gli attori (ruoli, non solo umani), casi d'uso e associazioni. Le relazioni coinvolgono attori e casi d'uso.



C'è da puntualizzare che i diagrammi dei casi d'uso sono "pericolosi". Inducono a scomposizione funzionale (non TDA/OO). Non bastano i diagrammi ma è necessario fornire anche un documento di descrizione testuale del caso d'uso.

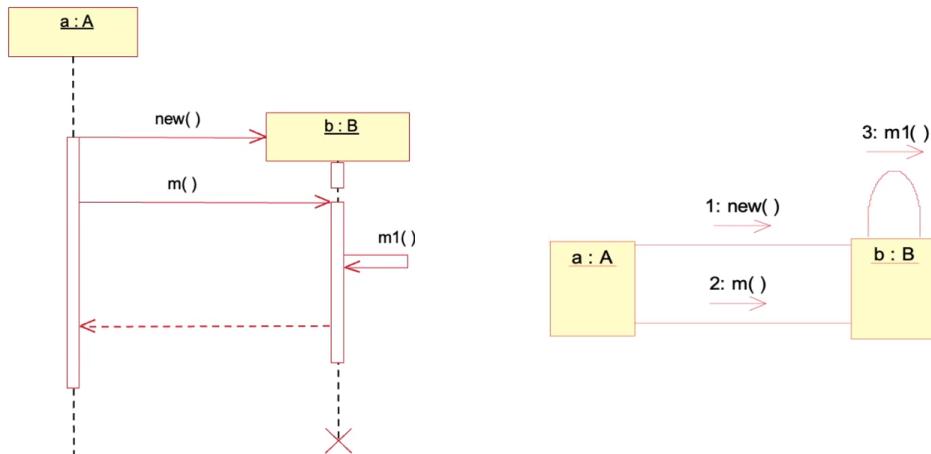
2.7 Diagrammi di sequenza

Sono diagrammi dinamici (classificati come di interazione) e descrivono cosa succede durante uno specifico flusso di esecuzione. Include gli oggetti e le azioni da essi compiuti, i messaggi scambiati e l'ordine di invocazione.



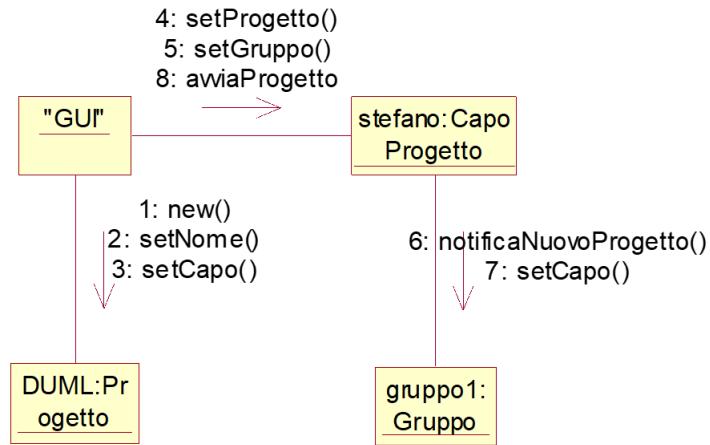
2.8 Diagrammi di collaborazione

Consistono in diagrammi (classificati come di interazione) degli oggetti più informazioni sulle invocazioni dei metodi. Contengono le stesse informazioni dei diagrammi di sequenza ma si predilige la disposizione degli oggetti piuttosto che l'ordine temporale.



Due diagrammi equivalenti, a sinistra di sequenza, a destra di collaborazione.

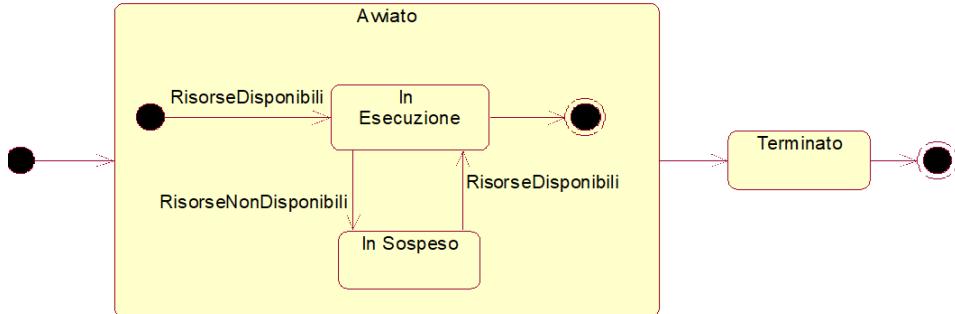
Un altro esempio:



2.9 Diagrammi degli stati

Visualizzano i vari stati di un'entità durante il suo ciclo di vita. Ogni entità è un oggetto/istanza, metodo, componente, sistema o sottosistema. Contiene stati iniziali e finali, transizioni, sottostati, eventi e guardie etc.

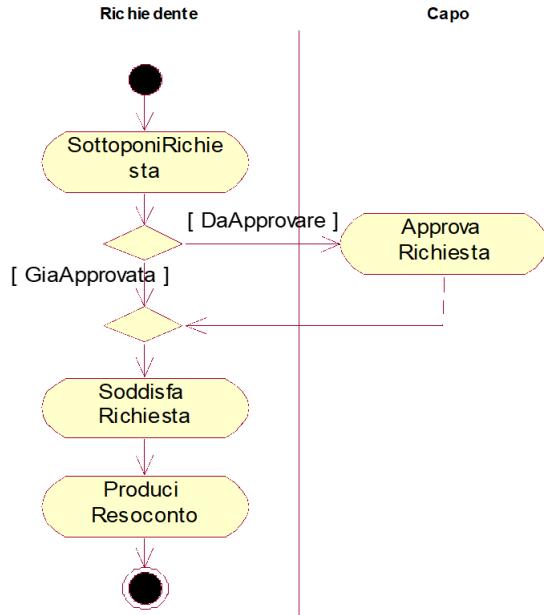
Un esempio:



2.10 Diagramma delle attività

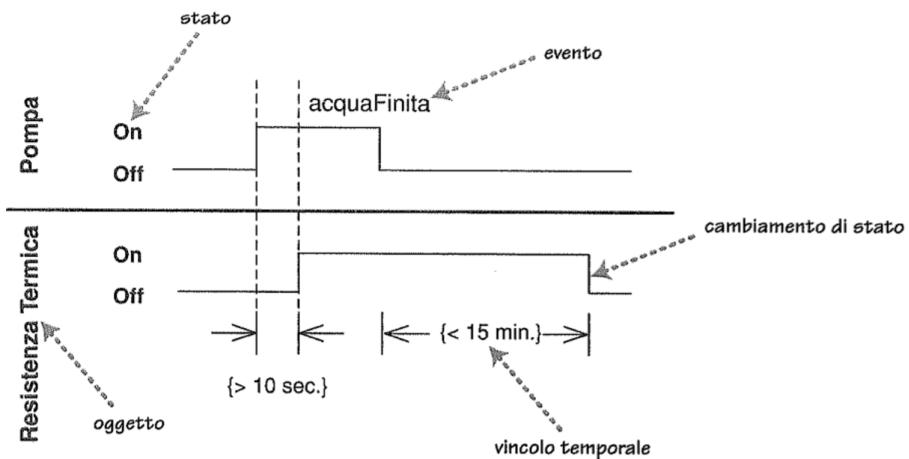
È un classico diagramma di flusso, contenente stati azione, transizioni, guardie, inizio e fine, decisioni, swimlane e punti di merge.

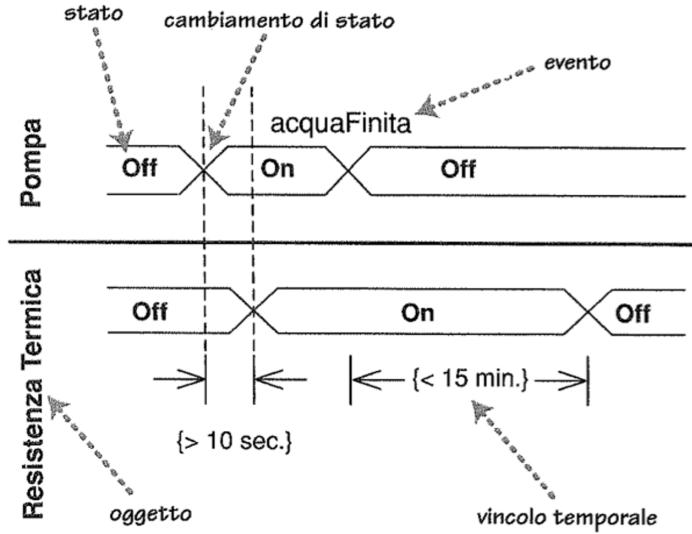
Un esempio:



2.11 Timing Diagrams

Rappresentano eventi, cambiamenti di stato, vincoli temporali. Il tempo è espresso orizzontalmente. Esistono due notazioni/varianti: robusta e coincisa.





2.12 Considerazioni sull'uso di UML

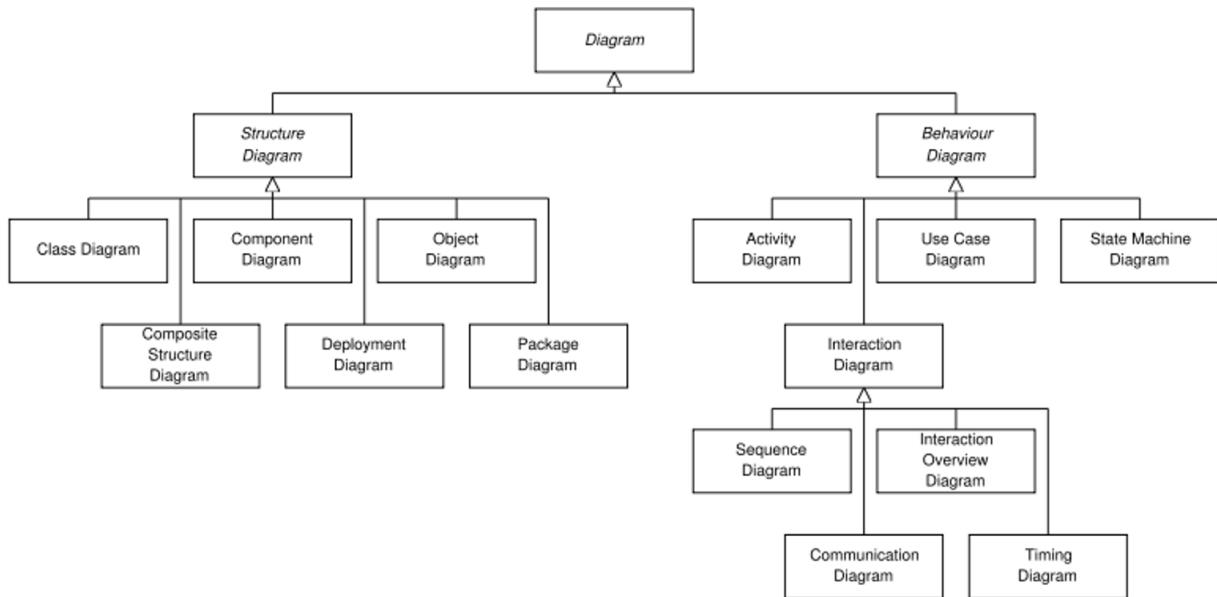
UML viene principalmente utilizzato per creare schizzi veloci, informali, come lingua franca. Può anche essere utilizzato con più criterio come strumento di progettazione, creando diagrammi formali e completi. Può anche essere visto come un linguaggio di programmazione e venir tradotto in codice eseguibile a partire dai diagrammi secondo uno dei due approcci:

- MDA (Model Driven Architecture)
 - PIM (Platform Indipendent Model): modello UML indipendente dalla tecnologia e prodotto da un umano;
 - PSM (Platform Specific Model): istanziato su una specifica tecnologia, modello prodotto da un tool;
 - Codice sorgente: generato automaticamente dal PSM.
- Executable UML: modello PIM più compilazione direttamente in codice.

L'uso di UML può essere visto da due prospettive:

- Concettuale: descrizione del dominio, del problema, OOA.
- Software (specifica + implementazione): descrizione del software, della soluzione, OOD, OOP.

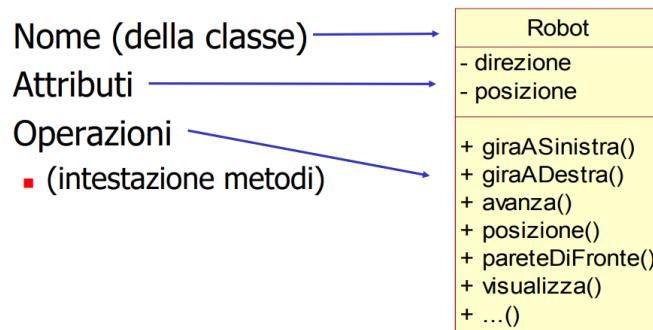
Uno schema riassuntivo dei diagrammi presenti in UML 2.0:



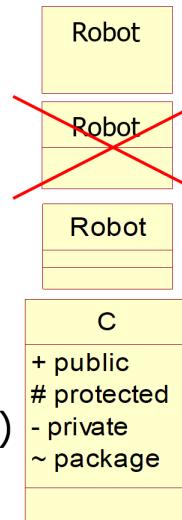
3 UML per il progetto OO: diagrammi di classe

In questa sezione vedremo i diagrammi di classe più in dettaglio per affrontare alcuni concetti della progettazione OO.

Simbolo di classe:

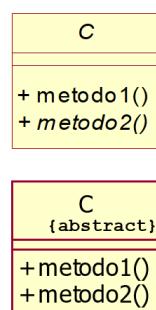


- **Forme abbreviate**
 - Senza attributi e operazioni
 - (Senza attributi o operazioni)
- **Visibilità:**
 - +: **public**
 - #: **protected**
 - -: **private**
 - ~: **package**
- **Sottolineatura: di classe (**static**)**



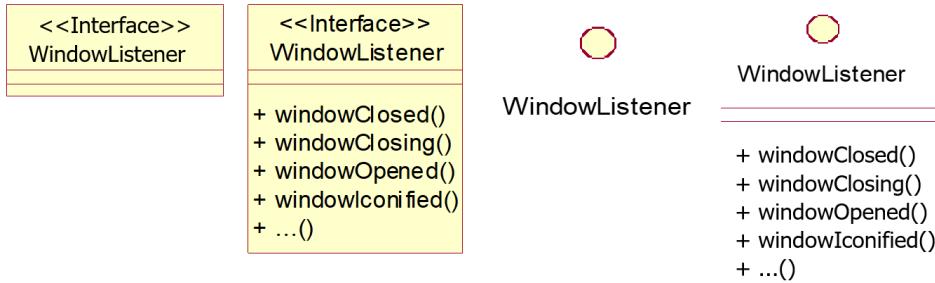
Classi astratte:

- In *corsivo*
- Oppure l'etichetta {**abstract**}
- Anche i metodi astratti



Interfacce:

- Come le classi, si usa lo stereotipo <<Interface>>



Le note sono assimilabili al concetto di commento nel codice sorgente. Permettono di rendere più chiari i diagrammi, soprattutto in caso di utilizzo di notazione "particolare" o raramente utilizzata.



3.1 Relazioni

Vediamo ora la notazione per le relazioni tra classi.

- Dipendenza generica;
- Associazione generica;
- Eredità;
- Composizione/aggregazione (tutto/parte).

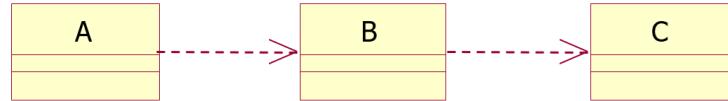
Dipendenza generica: A dipende da B se A ha bisogno di B per funzionare, ad esempio:

- A chiama metodi di B
- A usa il tipo B

Una modifica in B può richiedere modifica in A



Le dipendenze sono di solito direzionali e non transitive. Nell'esempio sottostante, una modifica a C richiede una modifica a B ma non necessariamente ad A.



Le dipendenze possono essere etichettate, ad esempio:

- <<call>>: invocazione metodo
- <<create>>: invocazione costruttore
- <<use>>: ne ha bisogno per funzionare

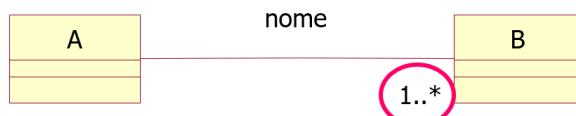
Non serve etichettare tutto ed indicare tutte le dipendenze.

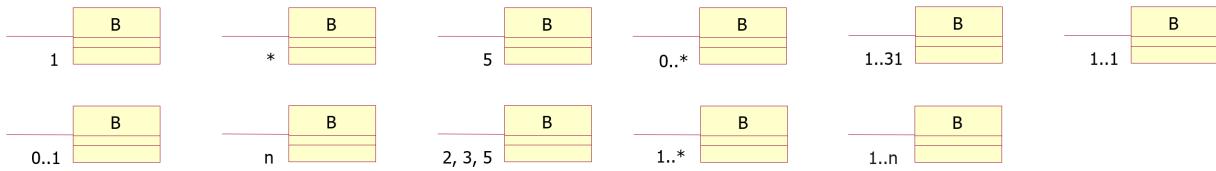
Associazione generica: Relazione qualsiasi ma di interesse fra oggetti (istanze).



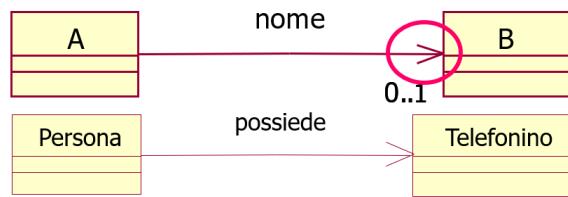
Nell'arco di associazione è possibile specificare il nome dell'associazione.

La molteplicità indica quante istanze per ogni legame. La molteplicità per A indica quante istanze legate a un'istanza di A trovo all'altro estremo (si indica vicino a B).





Navigabilità indica se dall'istanza di A c'è un accesso diretto all'istanza di B.

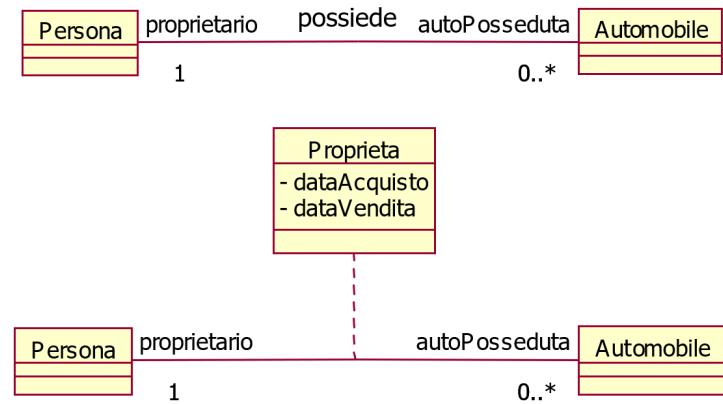


Il ruolo che ha una classe nell'associazione si indica alle estremità delle associazioni.

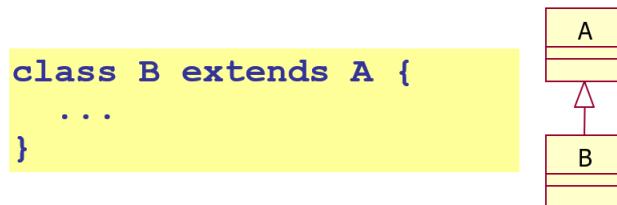


Si indica vicino alla classe e alla sua molteplicità ma dalla parte opposta rispetto all'attributo.

Classe di associazione: Nei diagrammi UML, una classe di associazione fa parte di una relazione di associazione tra altre due classi. È possibile associare una classe di associazione a una relazione di associazione per fornire ulteriori informazioni sulla relazione. Una classe di tale tipo è identica ad altre classi e può contenere operazioni, attributi e altre associazioni.

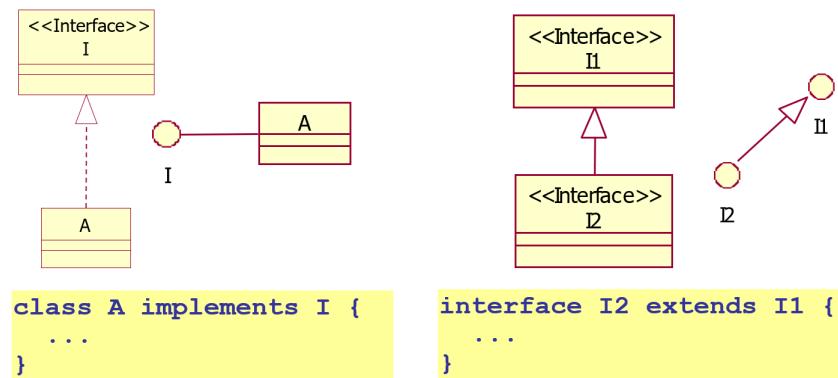


Eredità: B è sottoclasse di A

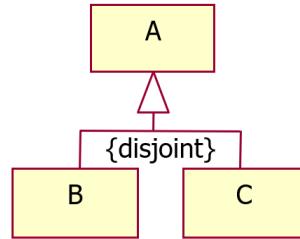


Gli attributi e metodi ereditati possono essere mostrati o no, entrambe le opzioni sono corrette.

Eredità e implementazione da interfacce:



Può inoltre essere indicato il partizionamento delle sottoclassi tra parentesi graffe, ad esempio: disjoint, overlapping, complete, incomplete, mandatory, not mandatory, static, dynamic etc.

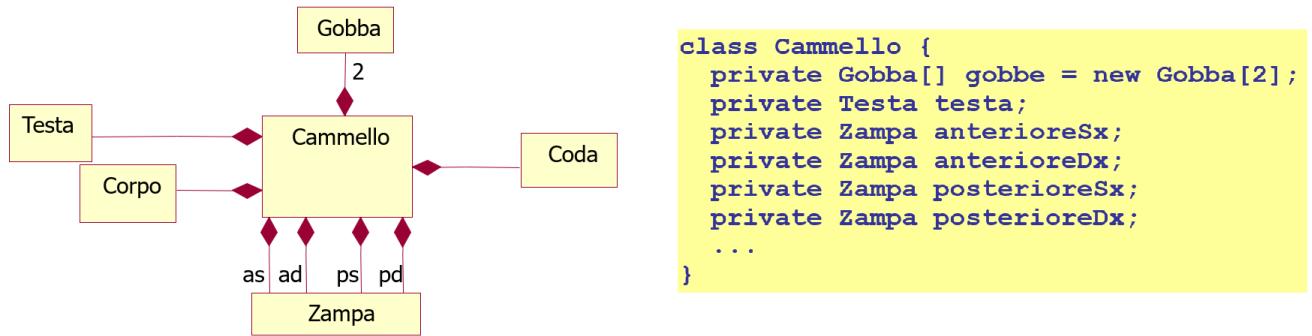


- disjoint/overlapping: se le sottoclassi sono o meno disgiunte;
- complete/incomplete: se il diagramma elenca o meno tutte le sottoclassi;
- mandatory/not mandatory: se l'istanza di superclasse deve essere o meno istanza di una delle sottoclassi (se mandatory, la superclasse non viene mai istanziata);
- static/dynamic: se un'istanza di una sottoclasse può o meno diventare un'istanza di un'altra sottoclasse;

Composizione: Esempio: le istanze della classe tutto contengono istanze della classe parte:



Altro esempio:



Aggregazione: Per comprendere l'aggregazione analizziamo le differenze con la composizione.



- Composto con zero componenti non esiste (sedia)
- Un componente può essere in un solo composto
- Eterogenea (le parti sono ≠)
- Aggregato con zero aggregandi può esistere (associazione)
- Aggregando può essere in più aggregati
- Omogenea (le parti sono =)

4 UML di qualità: linee guida, elementi di stile e colore

In questo capitolo tratteremo alcune linee guida per produrre diagrammi UML di qualità. Nel produrre un buon diagramma UML è importante tenere a mente la sua leggibilità.

- Evitare linee che si incrociano, se inevitabili, disegnarle con il "saltino";
- Evitare linee diagonali o curve;
- Utilizzare la simmetria, aiuta la comprensione;
- Organizzare i diagrammi in alto a sinistra (cultura occidentale);
- Non ignorare il significato della grandezza dei simboli (correlazione tra grandezza e importanza).

Un buon diagramma UML è semplice.

- Mostrare solo il necessario, troppo dettaglio rende incomprensibile il diagramma;
- Preferire la notazione nota (il 20% di UML basta per l'80% delle esigenze);
- Evitare diagrammi troppo estesi (divide et impera - dividere in sotto diagrammi);
- Prendere come riferimento un foglio A4.
- Descrivere ogni diagramma con una nota (leggenda);

La scelta dei nomi è fondamentale per rendere comprensibile un diagramma UML.

- Scegliere e utilizzare una convenzione di riferimento;
- Usare terminologia del dominio;
- Utilizzare convenzioni del linguaggio di programmazione solo per la fase di progetto;
 - Meglio "Send Message" di "sendMessage()";
- Coerenza nella nomina degli oggetti.

Nei diagrammi concettuali (di analisi) è più appropriato utilizzare la responsabilità piuttosto che attributi e/o operazioni. Escludere i simboli di visibilità (è un dettaglio di progetto) e i tipi di attributi/operazioni.

Ad esempio:

Analysis	Design															
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 2px 5px;">Order</td></tr> <tr><td style="padding: 2px 5px;">Placement Date</td></tr> <tr><td style="padding: 2px 5px;">Delivery Date</td></tr> <tr><td style="padding: 2px 5px;">Order Number</td></tr> <tr><td style="padding: 2px 5px;">Calculate Total</td></tr> <tr><td style="padding: 2px 5px;">Calculate Taxes</td></tr> </table>	Order	Placement Date	Delivery Date	Order Number	Calculate Total	Calculate Taxes	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 2px 5px;">Order</td></tr> <tr><td style="padding: 2px 5px;">- deliveryDate: Date</td></tr> <tr><td style="padding: 2px 5px;">- orderNumber: int</td></tr> <tr><td style="padding: 2px 5px;">- placementDate: Date</td></tr> <tr><td style="padding: 2px 5px;">- taxes: Currency</td></tr> <tr><td style="padding: 2px 5px;">- total: Currency</td></tr> <tr><td style="padding: 2px 5px; font-family: monospace;"># calculateTaxes(Country, State): Currency</td></tr> <tr><td style="padding: 2px 5px; font-family: monospace;"># calculateTotal(): Currency</td></tr> <tr><td style="padding: 2px 5px; font-family: monospace;">getTaxEngine() {visibility=implementation}</td></tr> </table>	Order	- deliveryDate: Date	- orderNumber: int	- placementDate: Date	- taxes: Currency	- total: Currency	# calculateTaxes(Country, State): Currency	# calculateTotal(): Currency	getTaxEngine() {visibility=implementation}
Order																
Placement Date																
Delivery Date																
Order Number																
Calculate Total																
Calculate Taxes																
Order																
- deliveryDate: Date																
- orderNumber: int																
- placementDate: Date																
- taxes: Currency																
- total: Currency																
# calculateTaxes(Country, State): Currency																
# calculateTotal(): Currency																
getTaxEngine() {visibility=implementation}																

- Per i nomi di classe utilizzare nomi singolari ed evitare abbreviazioni;
- Usare verbi chiari per i nomi di operazioni;
- Usare nomi del dominio per gli attributi;
- Non modellare dettagli implementativi (get/set etc.).

Relativamente al simbolo di classe:

- Non disegnare classi con 2 compartmenti (o 1 o 3);
- Etichettare i compartmenti non standard;
- ”...” per evidenziare la non completezza;
- Elencare attributi e operazioni in ordine di visibilità decrescente;
- Elenicare i tipi, non i nomi dei parametri formali;
- Se una classe implementa/realizza un’interfaccia, non ripetere le operazioni dell’interfaccia nella classe.
- Preferire linee orizzontali (non per l’eredità);
- Preferire relazioni ad albero, anzichè separate;
- Indicare sempre la molteplicità evitando quelle generiche, se possibile.

Relativamente alle associazioni:

- Centrare il nome sull’associazione;
- Usare verbi in forma attiva;
- Indicare la direzione per chiarire il nome;

- Se possibile, associazioni da sx a dx;
- Usare i nomi dei ruoli quando chiariscono;
- Dubitare delle molteplicità min-max (perché tendono a cambiare con il tempo);

Relativamente all'ereditarietà è buona norma "leggerla a parole" per verificarne la correttezza, inoltre, è consigliabile disegnarla verso l'alto.

Anche l'aggregazione e composizione è buona normale leggerla per verificarne la sensatezza. Evitare modellazioni troppo fini o troppo grossolane, modellare solo quello che è richiesto. Idealmente l'intero a sx e le parti a dx.

Relativamente ai casi d'uso:

- Usare verbi chiari, "forti" e "netti" per i casi d'uso;
- Usare la terminologia del dominio;
- Impilare temporalmente i casi d'uso: in alto quello che viene eseguito prima, anche se l'ordine temporale è solo隐式 e potrà cambiare, ma ordinarlo aiuta a comprendere il diagramma;
- Le linee oblique sono ammesse;
- Utilizzare la system boundary box.

Per quanto riguarda gli attori:

- Posizionare gli attori primari in alto a sinistra e verso l'esterno;
- Usare nomi singolari, presi dal dominio;
- I nomi devono modellare ruoli, non titoli (gestore progetto, non capo progetto);
- Ogni attore deve essere associato con almeno un caso d'uso;
- Usare <<system>> per indicare attori-sistemi;
- Usare un attore Tempo per eventi automatici;
- Non fare interagire gli attori fra di loro.

Per quanto riguarda le relazioni:

- Usare associazioni attore - caso d'uso non direzionali;
- Usare <<extend>> con parsimonia;
- Non utilizzare la "s": uses, includes, extends... rimuovere la s;
- Evitare associazioni più profonde di 2 livelli;

- <<include>> verso dx, <<extend>> e generalizzazione verso l'alto.

Relativamente a exetnd vs include nei casi d'uso:

- Include: chiamata di procedura, inclusione di un altro caso d'uso che c'è già per conto suo;
- Extend: gestione di flusso di controllo "eccezionale";

Diagrammi di package:

- Usare diagrammi dei package per organizzare il progetto;
- Usare nomi corti e descrittivi;
- Usare stereotipi per indicare i livelli architetturali: <<user interface>>, <<domain>>, <<database>>, ...
- Evitare dipendenze cicliche.

Diagrammi di sequenza:

- Messaggi da sinistra a destra (preferibilmente);
- Organizzare "a livelli" (orizzontalmente);
- Includere una descrizione in prosa: sulla sinistra, eventualmente come note, soprattutto per diagrammi complessi e/o di analisi;
- Dare un nome agli oggetti quando usati nei messaggi;
- Preferire i nomi dei parametri ai tipi.
- Non modellare i valori restituiti quando ovvi: modellarli se usati altrove nel programma; eventualmente modellarli nell'invocazione.
- Allineare/"Giustificare" le invocazioni in modo coerente: a sinistra ordine invocazioni, flusso di controllo, o a destra: vicino alla freccia, classe che contiene il metodo.

Diagrammi degli stati:

- Stato iniziale in alto a sinistra, stato finale in basso a destra;
- Attenzione a "buchi neri" e "buchi bianchi";
- Usare gerarchie di sottostati per situazioni complesse;
- Scelta nomi per gli stati: aggettivi, passato prossimo, presente;

Diagrammi dei componenti:

- Preferire notazione lollipop per la realizzazione di interfacec;

- Mettere ogni lollipop a sinsitra del componente che lo realizza;
- Rappresentare solo le interfacce importanti;
- Dipendenze da sx a dx, o dall'alto in basso;
- Dipendenze da componenti a interfacce, non altri componenti (notazione lollipop-socket).

4.0.1 Uso del colore per l'analisi

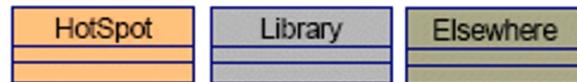
Diagrammi e colore: obiettivi. Modelli più espressivi con poco sforzo, chiave di lettura di un diagramma, capire a colpo d'occhio il ruolo di una classe all'interno di un diagramma. Aiuta a distinguere i diversi ruoli nel caso di analisi o di progetto. Quattro ruoli principali:

- Actor: classe principale, ruolo attivo;
- Moment: classe tempo-evento;
- Thing: classe secondaria;
- Description: classe descrittiva, passiva.

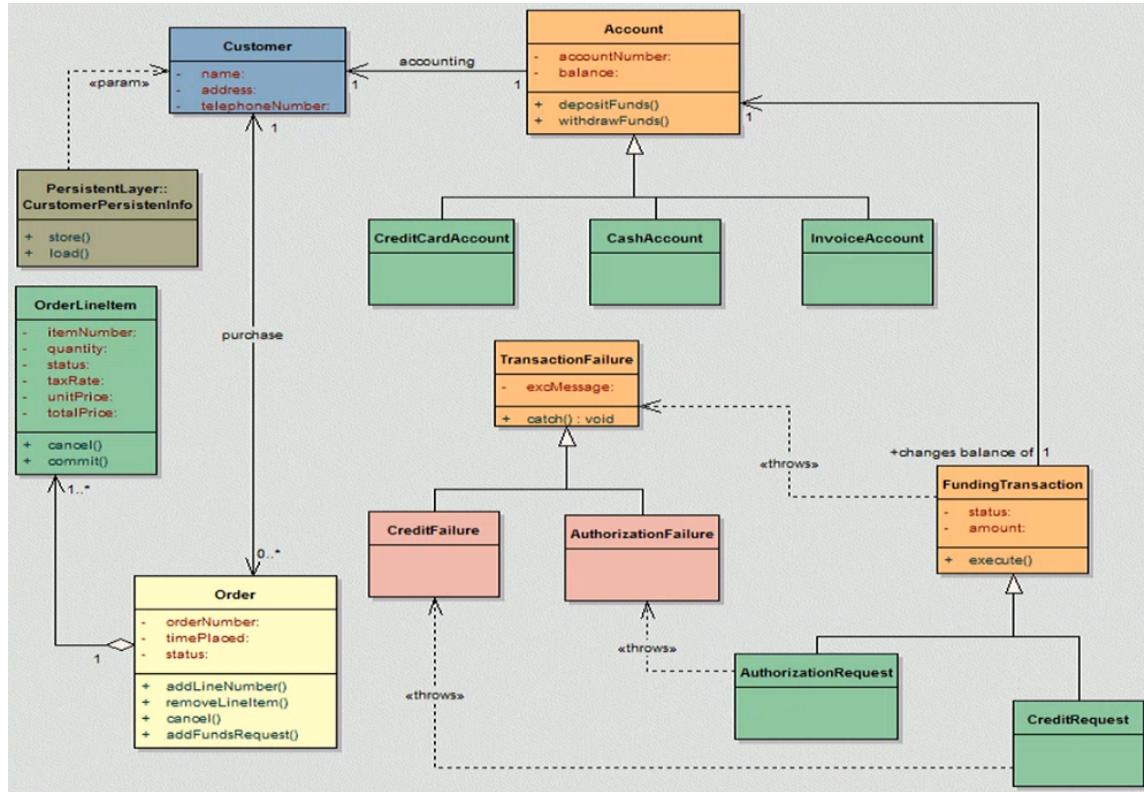


L'idea è stata estesa da Pescio. Altri tre ruoli:

- HotSpot: punto di estensione;
- Library: classe di libreria di terze parti;
- Elsewhere: classe definita altrove.



Esempio di diagramma complesso colorato:



5 Principi di Progetto Object Oriented

Vedremo ora i principi cardine della progettazione Object-Oriented:

- Incapsulamento
- Dipendenza
- Domini
- Ingombro
- Legge di Demeter

5.1 Incapsulamento

Racchiudere codice in una "capsula". Permette l'information hiding, occultando informazioni e implementazione. Nascondere l'implementazione e far vedere solo l'interfaccia (funzioni per manipolare i TDA). "Un meccanismo del linguaggio di programmazione atto a limitare l'accesso diretto agli elementi dell'oggetto".

Esistono più livelli di incapsulamento:

- Livello 0: codice grezzo, nessun incapsulamento;
- Livello 1: sottoprogramma, insieme di istruzioni (incapsulamento procedurale/funzionale);
- Livello 2: classe, insieme di sottoprogrammi, package.

Misure di buona modularizzazione a livello 1:

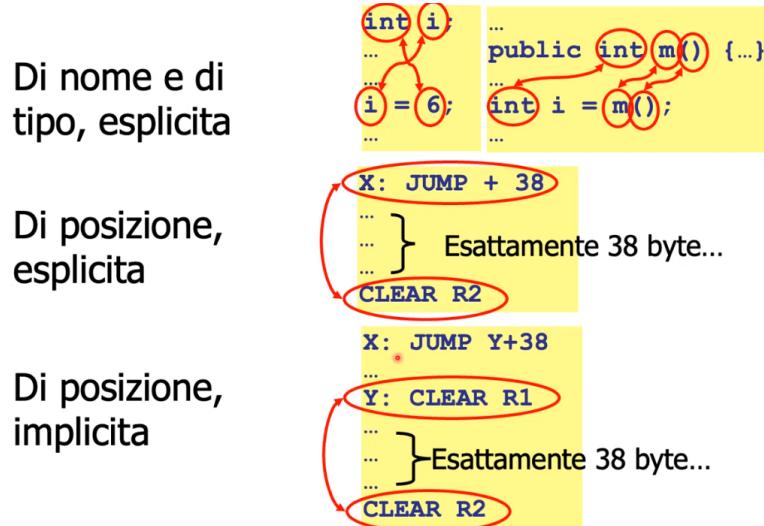
- **Coesione:** misura di quanto le linee di codice interne a un sottoprogramma sono "dedicate" alla "mission" del sottoprogramma;
- **Accoppiamento:** misura della forza delle connessioni fra sottoprogrammi diversi;
- **Ventaglio:** misura del numero di riferimenti ad altri sottoprogrammi;
- Si mira a: alta coesione, basso accoppiamento (e basso ventaglio).

Misure di buona modularizzazione a livello 2:

- **Coesione di classe:** misura di quanto le linee di codice interne a una classe sono dedicate alla mission della classe;
- **Accoppiamento di classe:** misura della forza delle connessioni tra classi diverse;
- **Ingombro:** misura del numero di riferimenti ad altre classi;
- Si mira a alta coesione di classe, basso accoppiamento di classe (e basso ingombro).

5.2 Dipendenza

Iniziamo analizzando esempi di dipendenza:



Due elementi α e β software sono dipendenti se:

- Esistono modifiche di α che richiedono la modifica di β ;
- Esistono modifiche che richiedono di modificare insieme α e β per mantenere la correttezza;

Esistono diversi tipi di dipendenza:

1. Di nome: dichiarazione e invocazione di metodi, attributi ereditati dalla superclasse, metodi ereditati;
2. Di tipo o classe: dichiarazione e uso di variabile o metodo;
3. Di convenzione; ad es: `nord = 0`; `sud = 1` etc.
4. Di algoritmo: funzioni di inserimento e ricerca in una tabella hash, codifica e decodifica, iteratore su un insieme che restituisce nell'ordine di inserimento;
5. Di posizione (ordine parametri);
6. Di esecuzione: inizializzare una variabile (o istanza di classe) prima di usarla, lettura e modifica di variabili condivise/globali.
7. Temporale: sistemi real-time, terminazioni condizionali;
8. Di valore: rispetto di invarianti, valori duplicati su base di dati, duplicazione e ridondanza;
9. Di identità: alias di variabili.

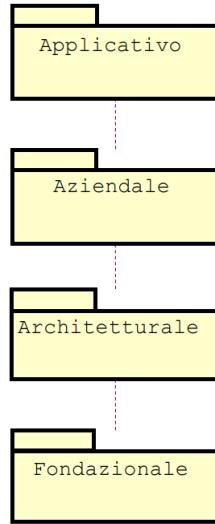
È normale che siano presenti dipendenze, alcune sono preferibilmente da evitare: quelle che creano un effetto domino, quando non sono limitate e attraversano/perforano i confini dell'incapsulamento. Le classi sono un ottimo freno alle dipendenze, attributi e metodi privati sono confinati alla classe, similmente alle variabili locali.

Antidipendenza: in certi casi ho bisogno che "due cose siano diverse" (concetto inverso di dipendenza). Dipendenza di differenza. α e β devono essere diversi (ad esempio: nomi di variabili in un metodo, due metodi in due sottoclassi della stessa gerarchia).

5.3 Domini

Le classi di un sistema non sono tutte uguali, possono avere diversa complessità, specializzazione, appartenere a domini diversi, etc. Abbiamo 4 domini principali:

1. Fondazionale;
2. Architetturale;
3. Aziendale;
4. Applicativo;



Fondazionale

Classi di base, fondamentali (Integer, Boolean, Char, etc.), strutturali (List, Set, etc.), semantiche (Ora, Denaro, Punto, Angolo, etc.).

Architetturale

Classi per una piattaforma hardware/software (GUI, rete, DB, etc.), per le comunicazioni di rete (Host, Port, Connection, etc.), gestione di basi di dati (Transazione, Query, etc.), interfaccia utente (Window, Button, etc.).

Aziendale

Classi relative a uno specifico contesto aziendale (un dominio applicativo): attributo, ruolo, relazione, ad esempio in dominio bancario: Saldo.

Applicativo

Classi relative a una specifica applicazione, di riconoscimento/gestione degli eventi. Business logic specifica.

Domini e riuso

Tendenzialmente il riuso cresce verso il basso (massimo sulle classi fondazionali). Ad esempio, le librerie, tendenzialmente contengono solo classi fondazionali e/o architetturali.

5.4 Ingombro

Con ingombro di una classe c ci si riferisce al numero di classi di cui c ha bisogno per funzionare. Si distingue riferimento diretto e indiretto; ingombro diretto e indiretto.

C fa riferimento diretto a D se:

- C eredita da D
- C ha un attributo di tipo D
- C ha un metodo con argomento di tipo D
- C ha un metodo con variabile locale di tipo D
- C invoca un metodo di D
- C istanzia un tipo parametrico di tipo D

Riferimento indiretto: chiusura transitiva, "quelli delle classi di cui fa riferimento".

Se l'insieme di riferimenti diretti di c è:

$$\text{RD}(C) = \{C_1, C_2, \dots, C_n\}$$

Insieme di riferimenti indiretti di c è:

$$RI(C) = RD(C) \cup \bigcup_i RI(C_i)$$

$$RI(C) = RD(C) \cup RI(C_1) \cup \dots \cup RI(C_n)$$

Ingombro:

Def. Ingombro diretto di C:

- 0 se $C \in$ dominio fondazionale
- # riferimenti diretti se
 $C \notin$ dominio fondazionale

Def. Ingombro indiretto di C:

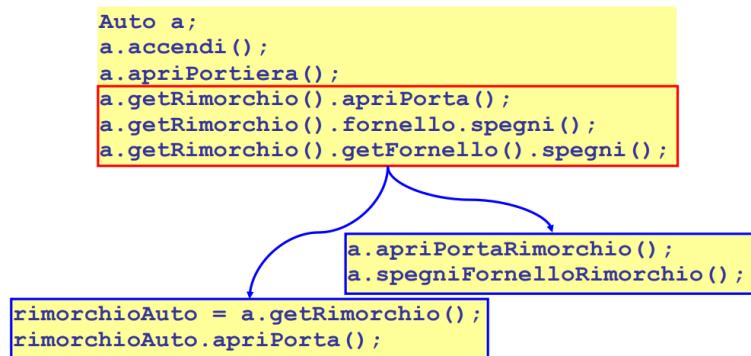
- 0 se $C \in$ dominio fondazionale
- # riferimenti indiretti se
 $C \notin$ dominio fondazionale

5.5 Legge di Demeter

Afferma che un oggetto *A* può richiedere un servizio (ovvero, chiamare un metodo) di un altro oggetto *B* ma l'oggetto *A* non può usare l'oggetto *B* per raggiungere un terzo oggetto *C* che possa soddisfare le sue richieste. Per qualsiasi metodo *m* di un oggetto *x* della classe *A*, il destinatario dei messaggi nel corpo di *m* deve essere:

1. *x* stesso (*this*)
2. un oggetto presente come argomento di *m*
3. un oggetto riferito da un attributo di *x*
4. un oggetto creato da *m*
5. un oggetto riferito da una variabile globale

Non c'è oggetto restituito da un altro metodo. Si può riassumere come "non più di un punto".



La legge di Demeter limita i riferimenti diretti, e quindi l'ingombro diretto delle classi e le dipendenze che attraversano l'incapsulamento. "Una classe deve usare solo quello che le serve, niente di più". Più cose uso, più c'è il rischio di errore quando una di queste cambia (inoltre rende difficoltoso il riuso e la modifica).

È una ragionevole linea guida, con svantaggi e vantaggi, non per forza la scelta migliore; utile per il refactoring.

Analizziamo ora altri principi del Design Object Oriented:

- Coesione
- Spazio degli stati
- Transizioni e comportamento
- Asserzioni

5.6 Coesione

La coesione di classe è una misura "esterna" di:

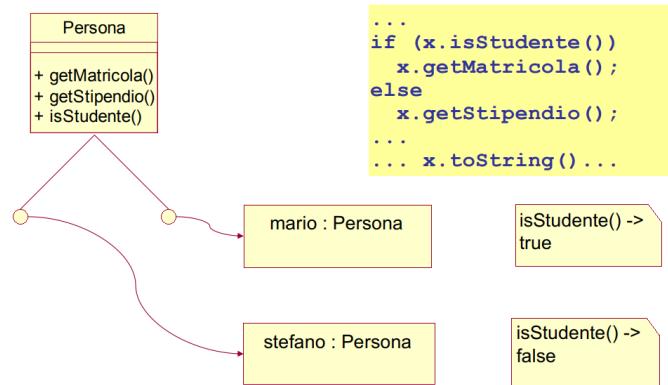
- Quanto coesa è l'interfaccia pubblica di una classe;
- Quanta corrispondenza reciproca c'è fra i pezzi (operazioni, attributi) dell'interfaccia pubblica di una classe;
- Quanto "sta bene insieme" ciò che c'è nell'interfaccia pubblica di una classe.

Possiamo identificare quattro livelli di coesione:

1. Coesione a istanza mista (peggiore);
2. Coesione a dominio misto;
3. Coesione a ruolo misto;
4. Coesione ideale (migliore).

Coesione a istanza mista:

Definizione: Classe con attributi o operazioni non definiti per alcune istanze. Classe troppo grossolana che "mette insieme" istanze che andrebbero separate. Le istanze sono divise in due sottoinsiemi: quelle con attributo/operazione non definito o definito.



Un approccio migliore contiene due sottoclassi: studente e lavoratore, studente non contiene `getStipendio()`, lavoratore non contiene `getMatricola()`.

Coesione a dominio misto:

Definizione: Classe ingombrata con una classe estrinseca appartenente a un dominio diverso (B è estrinseca rispetto ad A sse è possibile definire A senza alcun riferimento a B). Esempi:

Data è intrinseca rispetto Persona; Elefante è estrinseca rispetto Persona. stampa() in Documento; è meglio `toString()`; ingombro la classe Documento con String e non con Printer, molto meglio.

Coesione a ruolo misto:

Definizione: È ingombrata con una classe estrinseca appartenente allo stesso dominio. Come la precedente, ma la classe che ingombra è nello stesso dominio della classe ingombrata.

Esempio: classe Persona con attributo caniPosseduti e metodo getCaniposseduti(). Persona non ha coesione a istanza mista né coesione a dominio misto; ma il problema sta che Cane e Persona sono estrinseci; si ha una fat interface nel momento in cui Persona ha anche barche, auto, case, gatti, telefoni posseduti etc. etc. Questo approccio non è ideale per il riuso: se voglio riutilizzare Persona in un contesto dove non tengo conto degli oggetti posseduti risulta complesso e non ottimale.

Considerazioni sulla coesione: una classe deve essere un'astrazione uniforme sugli oggetti reali.

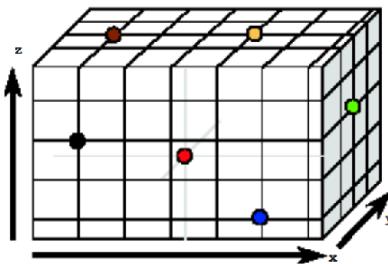
- **Astrazione:** non deve rappresentare tutte le caratteristiche degli oggetti reali.
- **Uniforme:** tutte le istanze devono essere uniformi, se ci interessa un attributo per una istanza deve interessarci per tutte; ovvero evitare coesione a istanza mista.

È buona norma evitare casi di **coesione alternativa** (più comportamenti alternativi in un'operazione: `scalaORuota()`); e di **coesione multipla** (`scalaERuota()`). La coesione funzionale (o ideale) vuole che si abbia un unico comportamento nell'operazione, che fa un'unica cosa.

5.7 Spazio degli stati

Con spazio degli stati (SdS) di una classe C ci si riferisce all'insieme degli stati in cui un'istanza di C può ritrovarsi. Si può vedere quindi lo stato come un valore possibile per l'istanza della classe C.

```
class Punto3d {  
    private double x;  
    private double y;  
    private double z;  
    ...  
}
```



Con dimensioni dello SdS ci si riferisce alle coordinate necessarie per specificare lo stato di un oggetto ("gradi di libertà"). Ad esempio, punto2D: 2 coordinate; punto3D: 3 coordinate; Persona: nome, cognome, dataDiNascita (che a sua volta ha 3 coordinate: gg/mm/aa).

Si consideri la classe VeicoloStradale e Automobile. Che relazione c'è fra i due SdS?

- **Restrizione:** dovunque ci deve essere `VeicoloStradale` ci posso mettere `Automobile`.
- **Estensione:** `Automobile` estende `VeicoloStradale`, ad esempio con un nuovo attributo `numeroPasseggeri`.

Quindi, nel caso di una classe B che eredita da A:

- **Estensione:** lo SdS di B può aggiungere dimensioni (nuova coordinata - attributo) allo SdS di A, ma...
- **Restrizione:** ...la proiezione dello SdS di B sulle coordinate dello SdS di A deve essere contenuta nello SdS di A.

5.8 Transizioni e comportamento

Con transizione ci si riferisce al passaggio da un punto nello SdS ad un altro. Con comportamento di una classe ci si riferisce all'insieme delle transizioni lecite.

Nel caso di una sottoclasse B di A, quali sono le transizioni in B (rispetto ad A)? di più o di meno, dipende (Cavallo zoppo negli scacchi (di meno), cavallo 3D (di più)).

5.9 Asserzioni

Un'asserzione è un'espressione logica che deve essere sempre vera in un certo punto del codice (durante qualsiasi esecuzione). In ambito Object Oriented queste sono: invarianti di classe, pre e post condizioni.

Invarianti di classe: Sono asserzioni sugli oggetti (istanze) di una classe. Definizione: condizione che ogni oggetto della classe deve soddisfare (quando in "equilibrio", ovvero dopo il costruttore, e non durante una transizione - durante l'esecuzione di un suo metodo).

Esempio: classe Triangolo con disuguaglianza triangolare, somma di due lati maggiore del terzo lato.

Invarianti e spazio degli stati: le invarianti possono essere viste come vincoli sullo SdS di una classe. I vincoli dell'invariante possono diminuire la dimensionalità (gradi di libertà) dello SdS di una classe. Le invarianti di classe definite per una specifica classe (ad esempio Rettangolo) possono variare in base all'implementazione. Se implementiamo un Rettangolo con 4 punti 2D abbiamo 8 vincoli, se salviamo centro, altezza, larghezza e angolo ne abbiamo 5.

- Dimensionalità di tipo: $\min(\text{dimensionalità di classe})$
- Dimensionalità di classe - dimensionalità di tipo: numero di vincoli che l'invariante deve specificare.
- Più vincoli ho e più è difficile scrivere l'invariante.

Pre-condizioni: Le precondizioni di un metodo sono condizioni che devono essere vere subito prima dell'esecuzione del metodo. Ad esempio $x \geq 0$ per il metodo `sqrt(int x)`. Se la precondizione non è vera, il risultato non è predicibile. La precondizione può essere usata sui parametri, sullo stato dell'istanza, sullo stato di altre istanze, etc.

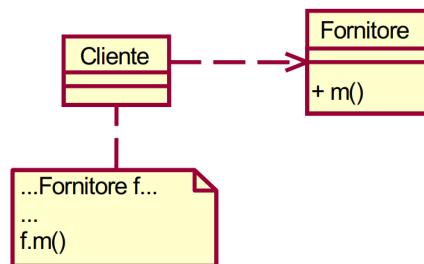
Post-condizioni: Le postcondizioni di un metodo sono condizioni che devono essere vere subito dopo l'esecuzione di un metodo. Se la postcondizione non è verificata (e ovviamente tutte le pre erano vere), allora il metodo è errato . La postcondizione è tipicamente sul valore restituito.

Vedremo ora i seguenti ulteriori concetti dell'Object-Oriented-Design:

- Progetto per contratto;
- Conformità di tipo;
- Comportamento chiuso;
- Pericoli di ereditarietà e polimorfismo;

5.10 Progetto per contratto

Invocazione di un metodo m (con pre e post condizione) della classe Fornitore dalla classe Cliente (con invarianti). Si ha un contratto tra cliente e fornitore; il fornitore fornisce il servizio dato dal metodo m . Il tutto si basa su invarianti, pre e post condizioni.



Il fornitore garantisce che se pre e inv sono vere, allora, dopo l'esecuzione di m , vale la post condizione (e l'invariante). Il fornitore chiede che valgano pre e inv, altrimenti non garantisce la post. Pre = richiesta che il metodo/fornitore fa al chiamante/cliente. Post = assicurazione/garanzia che il metodo/fornitore fornisce.

Il cliente ha la responsabilità di assicurare la validità della pre-condizione; il fornitore (metodo chiamato) può non controllarla.

- Pre non valida: errore nel chiamante;
- Post o inv. non valida: errore nel fornitore;

Una pre forte, restrittiva, è una buona notizia per il fornitore (chi scrive il metodo m) o per il cliente (chi chiama m)? Una post forte, restrittiva, è una buona notizia per il fornitore o per il cliente?

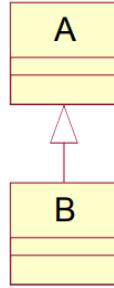
Una pre forte, restrittiva, è una cattiva notizia per il cliente, che deve soddisfare una richiesta più “difficile” ma una buona notizia per il fornitore, che ha la responsabilità di rispettare il contratto in meno casi.

Una post forte, restrittiva, è una cattiva notizia per il fornitore, che deve garantire qualcosa di più una buona notizia per il cliente, a cui viene fornito un servizio più “impegnativo”.

5.11 Conformità di tipo

Ogni gerarchia deve rispettare il principio della conformità di tipo: un'istanza della sottoclasse può essere fornita ovunque ci si aspetti un'istanza della sopraclasse e il programma funziona ancora correttamente quando qualunque operazione di accesso viene eseguita.

Invariante, eredità e conformità di tipo: devo poter mettere un'istanza di B ovunque ci va un'istanza di A; esempio: Triangolo e TriangoloIsoscele.



- $\text{inv}(B)$ deve garantire $\text{inv}(A)$
- $\text{inv}(B)$ deve essere restrittiva almeno quanto $\text{inv}(A)$

Quindi per garantire la conformità di tipo:

- invariante della sottoclasse forte almeno quanto l'invariante della sopraclasse;
- pre nel metodo della sottoclasse potenzialmente più debole;
- post nel metodo della sottoclasse forte almeno quanto quello della sopraclasse.

Relativamente a post e pre condizioni si può riassumere il tutto con: "il sottofornitore deve chiedere di meno e garantire di più".

5.12 Comportamento chiuso

Una classe è chiusa rispetto al comportamento se tutte le transizioni lasciano gli oggetti nello SdS. Il comportamento che una sottoclasse eredita da una sopraclasse deve rispettare l'invariante della sottoclasse. L'esecuzione di qualsiasi metodo di una classe C, compresi i metodi ereditati, deve obbedire all'inv. di C. Ciò non è automatico, va progettato e verificato esplicitamente. Chi progetta una (sotto)classe deve garantire la chiusura del suo comportamento.

Le sopraclasse non sanno nulla delle loro sottoclassi. La dipendenza è dalla sottoclasse alla sopraclasse, non viceversa. Quando progetto una sottoclasse devo sovrascrivere/ridefinire i metodi ereditati che non rispettano l'invariante della sottoclasse. Devo assumere che non posso modificare la sopraclasse (spesso è così: classi delle API). La sottoclasse deve occuparsi di verificare che i metodi ereditati non violino le proprie invarianti; e qualora fosse così, sovrascrivere i metodi.

Riepilogando: per avere gerarchie a "prova di bomba"; devo rispettare il principio della **conformità di tipo** (accesso) e del **comportamento chiuso** (modifica):

- invariante della sottoclasse forte almeno quanto quella della sopraclasse
- pre nel metodo della sottoclasse potenzialmente più debole
- post nel metodo della sottoclasse forte almeno quanto quello della sopraclasse
- metodi ereditati dalla sopraclasse devono rispettare l'invariante della sottoclasse (comportamento chiuso)

I prossimi punti relativi all'Object Oriented che analizzeremo sono:

- Genericità
- Anelli di operazioni
- Tipologie di stati e transizioni di un'interfaccia
- S.O.L.I.D principles

5.13 Genericità

Una classe parametrica (o "generica/generics"; o "template") è una classe che ha come parametro un nome di classe ad ogni creazione di istanza; esempio: List<float> su C#.

5.14 Anelli di operazioni

Organizzare una classe con operazioni più interne e più esterne; operazioni definite in termini di altre operazioni.

```
class Pila {
    public void pop(int x){
        if (this.numElem!=0) {
            ...
        }
    }
    public boolean vuota (){
        ...
    }
}
```

Si migliora l'incapsulamento.

5.15 Tipologie di stati e transizioni di un'interfaccia

Un oggetto si muove nello SdS in seguito a messaggi ricevuti; in quali stati? con quali transizioni? Gli stati raggiungibili dall'interfaccia pubblica di una classe possono essere:

- illegali: non rispettano l'inv.
- incompleti: l'interfaccia non consente di raggiungere tutti gli stati legali
- non pertinenti: l'interfaccia fornisce visibilità a stati che non c'entrano
- ideali: ok

I comportamenti (transizioni) possibili dall'interfaccia pubblica di una classe possono essere:

- illegali: consente transizioni illegali
- pericolosi: consente di raggiungere stati non appartenenti allo SdS
- non pertinenti: consente comportamenti che non hanno a che fare con la classe
- incompleti: non consente tutti i comportamenti leciti
- scomodi: consente tutti i comportamenti legali, ma per alcuni servono più messaggi
- replicati: lo stesso comportamento può essere ottenuto in più modi
- ideali: legali e non pericolosi, completi, pertinenti, non scomodi e non replicati

5.16 S.O.L.I.D.

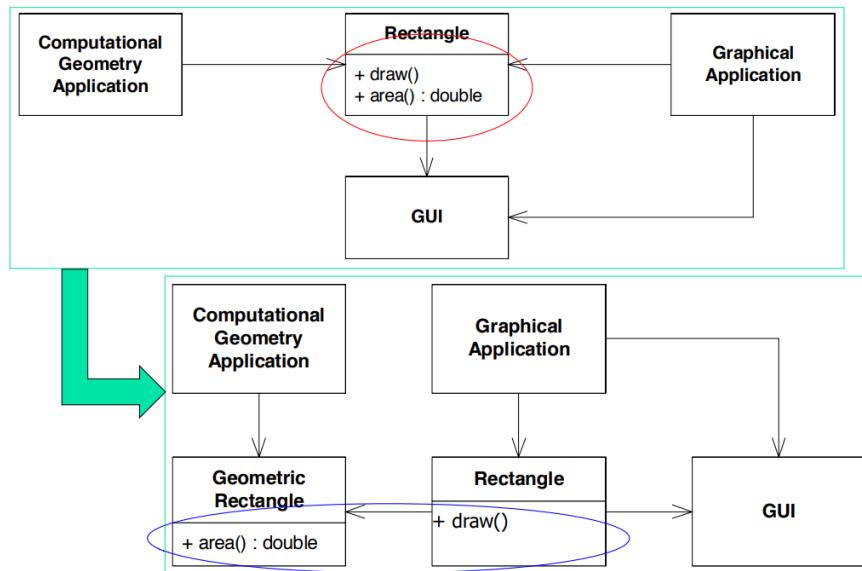
Sono 5 dei principi per l'OOD più riusabili e modificabili:

- Single responsibility
- Open-closed
- Liskov substitution
- Interface segregation
- Dependency inversion

Single responsibility

Il principio di singola responsabilità afferma che ogni elemento di un programma (classe, metodo, variabile) dovrebbe avere una sola responsabilità, e che tale responsabilità (o singolo motivo di cambiare) debba essere interamente incapsulata dall'elemento stesso. Tutti i servizi offerti dall'elemento dovrebbero essere strettamente allineati a tale responsabilità. Implica alta coesione.

Esempio:



Open-closed

Il principio aperto-chiuso afferma che le entità (classi, moduli, funzioni, ecc.) software dovrebbero essere aperte all'estensione, ma chiuse alle modifiche; in maniera tale che un'entità possa permettere che il suo comportamento possa essere modificato senza alterare il suo codice sorgente.

In particolare, una volta completata l'implementazione di un'entità, questa non dovrebbe essere più modificata, eccetto che per eliminare errori di programmazione.

L'introduzione di nuove caratteristiche o la modifica di quelle esistenti dovrebbe richiedere la creazione di nuove entità.

Ciò è particolarmente importante in un ambiente di produzione, in cui i cambiamenti al codice sorgente possono richiedere la revisione del codice, test di unità, e altre procedure tali da garantire la qualità di un prodotto software: il codice che rispetta il principio non cambia quando viene esteso, e quindi non ha bisogno di tale sforzo.

Liskov substitution

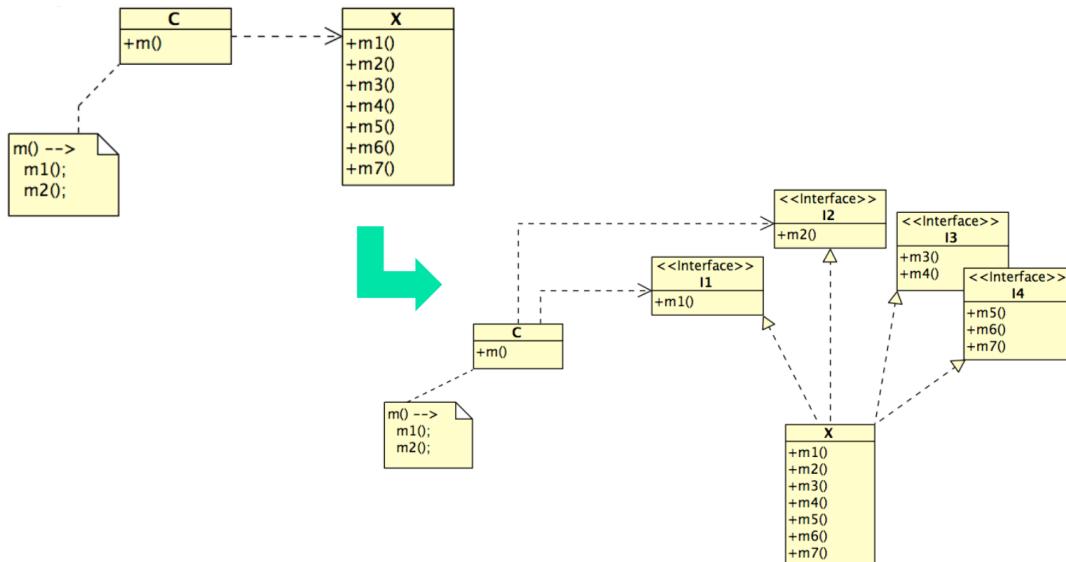
Gli oggetti in un programma dovrebbero essere rimpiazzabili da istanze dei loro sottotipi senza alterare la correttezza del programma.

Interface segregation

Il principio di segregazione delle interfacce è un principio di progettazione del software (soprattutto object-oriented) che afferma che un client non dovrebbe dipendere da metodi che non usa, e che pertanto è preferibile che le interfacce siano molte, specifiche e piccole (composte da pochi metodi) piuttosto che poche, generali e grandi.

Questo consente a ciascun client di dipendere da un insieme minimo di metodi, ovvero quelli appartenenti alle interfacce che effettivamente usa. In ossequio a questo principio, un oggetto dovrebbe tipicamente implementare numerose interfacce, una per ciascun ruolo che l'oggetto stesso gioca in diversi contesti o diverse interazioni con altri oggetti.

Esempio:



Dependency inversion

Il principio di inversione delle dipendenze è una tecnica di disaccoppiamento dei moduli software, che consiste nel rovesciare la pratica tradizionale secondo cui i moduli di alto livello dipendono da quelli di basso livello.

Sintetizzato:

I moduli di alto livello non dovrebbero dipendere dai moduli di basso livello. Entrambi dovrebbero dipendere da astrazioni. Le astrazioni non dovrebbero dipendere dai dettagli, i dettagli dovrebbero dipendere dalle astrazioni.

Nella pratica convenzionale della programmazione, i componenti software sono organizzati in una gerarchia di astrazione che coincide con una gerarchia di uso e definisce una corrispondente struttura di dipendenze. In altre parole, i componenti di alto livello realizzano le proprie funzioni facendo uso di componenti di più basso livello, attraverso le interfacce esposte da questi ultimi, e questo normalmente implica una dipendenza dei componenti di alto livello da quelli di basso livello.

Il principio dell'inversione delle dipendenze ha lo scopo di impedire che le dipendenze riproducano in questo modo la gerarchia d'uso e di astrazione. Anziché riferirsi direttamente alle interfacce dei componenti di basso livello, i componenti di alto livello fanno solo riferimento ad astrazioni del funzionamento di tali componenti. Alle stesse astrazioni fanno riferimento i componenti di basso livello. Il riferimento è diverso nei due casi: un componente di alto livello usa determinate astrazioni, mentre uno di basso livello le implementa. Entrambe queste relazioni possono concretizzarsi in dipendenze di compilazione, nel senso che sia la compilazione dei componenti di alto livello che quella dei componenti di basso livello viene fatta con riferimento alle astrazioni usate come "collante"; tuttavia, non si instaura nessuna dipendenza di compilazione dai componenti di alto livello a quelli di basso livello. Poiché la definizione delle astrazioni "usate" da un componente di alto livello è concettualmente a carico di tale componente, e quindi solitamente appartiene allo stesso "contesto" (per esempio package o namespace), la dipendenza del componente a basso livello dall'astrazione è di fatto una dipendenza dal componente di basso livello a quello di alto livello, da cui l'idea che la dipendenza sia "invertita" rispetto a quella tradizionale.

5.17 Frameworks

Un framework può essere visto come un "insieme di classi" che forniscono certe funzionalità di uno specifico dominio.

Rispetto le librerie, i cui metodi vengono invocati dal client/programmatore, il programmatore scrive i metodi che verranno chiamati dal codice del framework. Si ha un'inversione di controllo; il programmatore "copre i buchi" del codice del framework.

Se le librerie possono essere viste come un'insieme di funzioni per creare più agilmente un'applicazione, un framework può essere visto come uno scheletro generico da "completare/specializzare" di un'applicazione.

Un framework può avere 3 diverse caratteristiche/dimensioni:

Ambito

- Applicativo: funzioni comuni a varie applicazioni
- Dominio: specifico per un dominio applicativo
- Supporto: specifico di un task

Metodo di configurazione

- White-box: il programmatore d'istanza deve scrivere codice
- Black-box: il programmatore d'istanza deve selezionare il codice da eseguire

Metodo di interazione

- Chiamante: canonico, inversione del controllo
- Chiamato: come una libreria software

6 Design patterns

Gli elementi per il riuso del software ad oggetti (design patterns - schemi progettuali) sono dei modelli logici che descrivono una soluzione generale a un problema di progettazione ricorrente, verificabile in fase di progetto e sviluppo del software.

Nello specifico, ciascun design pattern attribuisce un nome al problema risolto, astrae e identifica gli aspetti principali della struttura utilizzata per la soluzione del problema, identifica le classi e le istanze partecipanti e la distribuzione delle responsabilità, infine, descrive quando e come può essere applicato. In breve definisce un problema, i contesti tipici in cui si verifica e la soluzione ottimale allo stato dell'arte.

Il concetto di design pattern nasce con la pubblicazione, nel 1994, del libro "Design Patterns: Elements of Reusable Object-Oriented Software", i cui autori vengono spesso chiamati "Gang of Four (GoF)". Nel libro, vengono illustrati 23 pattern, suddivisi in 3 categorie:

- **Strutturali**

1. Adapter (adattatore)
2. Façade (facciata)
3. Composite (composto)
4. Decorator (decoratore)
5. Bridge (ponte)
6. Proxy (proxy)
7. Flyweight (peso piuma)

- **Creazionali**

8. Factory method (metodo fabbrica)
9. Abstract factory (fabbrica astratta)
10. Singleton (singololetto)
11. Prototype (prototipo)
12. Builder (costruttore)

- **Comportamentali**

13. Template method (metodo sagoma)
14. Strategy (strategia)
15. State (stato)
16. Command (comando)
17. Observer (osservatore)
18. Mediator (mediatore)
19. Memento (ricordo)
20. Iterator (iteratore)
21. Visitor (visitatore)
22. Chain of responsibility (catena di responsabilità)
23. Interpreter (interprete)

Le tre categorie possono essere descritte come segue:

- **Strutturali:** spiegano come organizzare e assemblare oggetti e classi in strutture più complesse, mantenendo un elevato livello di flessibilità ed efficienza;
- **Creazionali:** forniscono meccanismi di creazione di oggetti che aumentano la flessibilità e il riuso del codice preesistente;
- **Comportamentali:** si prendono cura di fornire una comunicazione efficace tra gli oggetti e di distribuire correttamente le singole responsabilità.

Adapter

Scopo

Il fine dell'adapter è di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti. Il problema si presenta ogni volta in fase di progetto del software si debbano utilizzare sistemi di supporto (come per esempio librerie/API) la cui interfaccia non è perfettamente compatibile con quanto richiesto da applicazioni già esistenti.

Invece di dover riscrivere parte del sistema, compito oneroso e non sempre possibile se non si ha a disposizione il codice sorgente, può essere comodo scrivere un adapter che faccia da tramite tra l'interfaccia già presente e il codice del cliente.

In breve, quindi, l'adapter adatta l'interfaccia di una classe già pronta (il cui codice sorgente spesso non è accessibile) all'interfaccia che il cliente si aspetta. La metafora è quella del riduttore/adattatore per prese di corrente.

Casi d'uso

- È richiesto l'utilizzo di una classe esistente che presenti un'interfaccia diversa da quella desiderata dal cliente;
- È richiesta la scrittura di una classe senza poter conoscere a priori le altre classi con cui dovrà operare, in particolare senza poter conoscere quale specifica interfaccia sia necessario che la classe debba presentare alle altre.

Diagramma UML

Distinguiamo due tipologie di adapter:

- **Object-Adapter:** basato su delega e composizione;
- **Class-Adapter:** basato su ereditarietà in cui l'adattatore eredita sia dall'interfaccia attesa (che deve essere un'interfaccia) sia dalla classe adattata.

Diagramma UML dell'Object-Adapter:

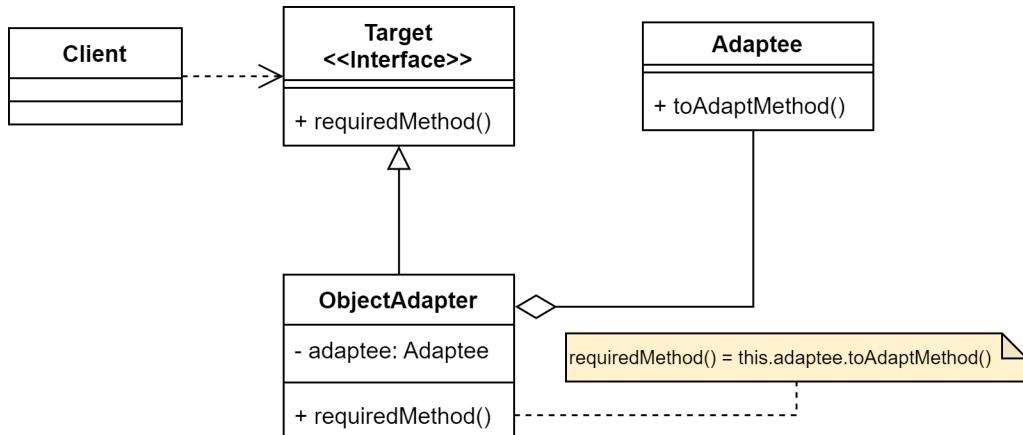
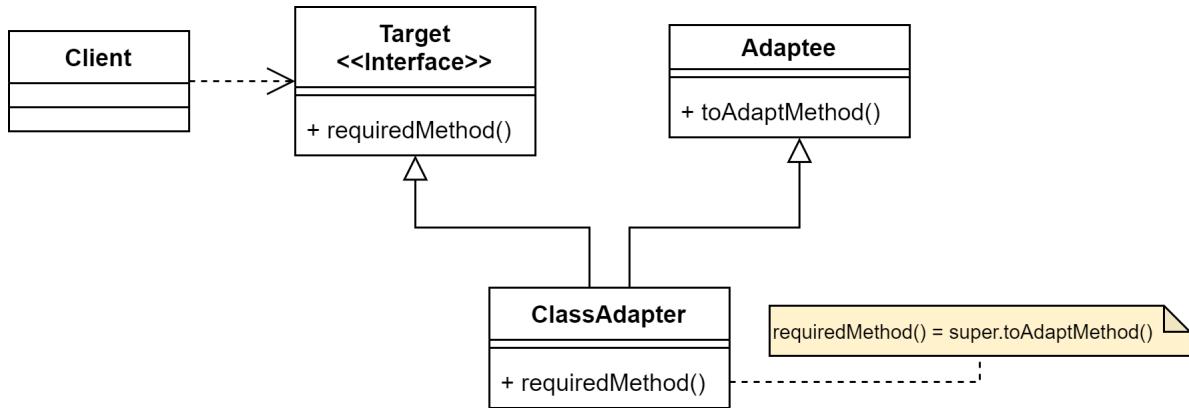


Diagramma UML del Class-Adapter:

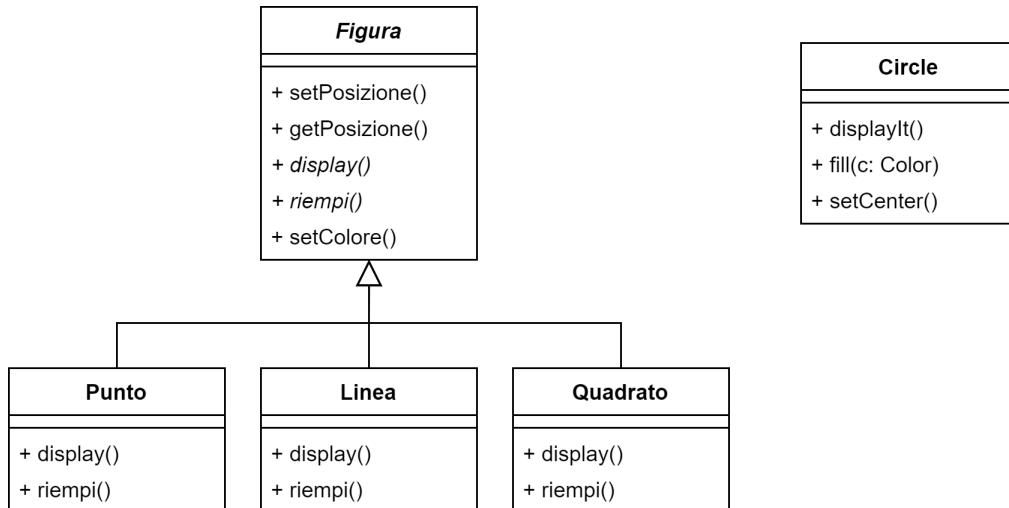


Partecipanti

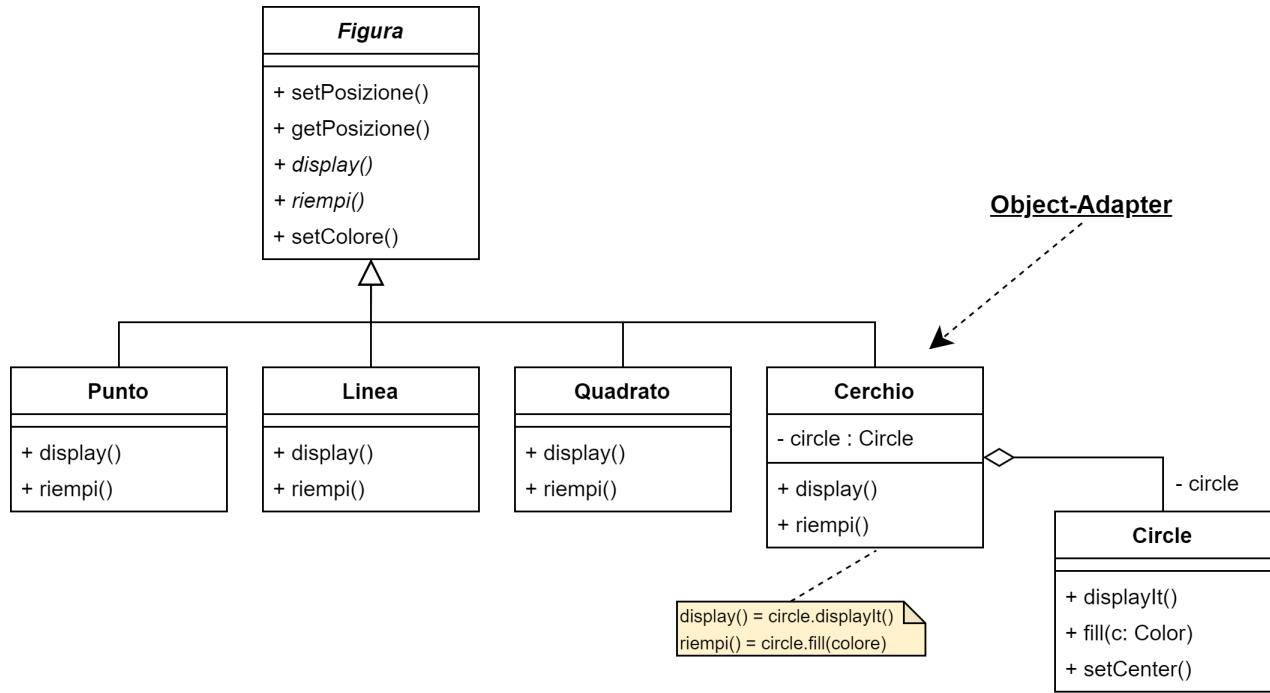
- **Adaptee**: l'interfaccia (o classe) che ha bisogno di essere adattata (ad esempio l'API);
- **Target**: definisce l'interfaccia che usa il cliente;
- **Client**: colui che utilizza indirettamente i servizi dell'interfaccia adattata (**Adaptee**) tramite l'interfaccia attesa **Target** (grazie all'adapter);
- **Adapter**: adatta l'interfaccia **Adaptee** all'interfaccia **Target**.

Esempio

Possediamo la gerarchia qui presente e vogliamo aggiungere la classe **Circle** (non compatibile e il cui codice sorgente non è accessibile).



Creiamo un adapter per la classe **Circle**:



La classe **Cerchio** deve avere una certa interfaccia attesa, imposta dall'ereditarietà della gerarchia indotta dalla superclasse astratta **Figura**. La classe **Circle** non è compatibile come sottoclasse e viene quindi adattata tramite un adapter. Questo è un esempio di Object-Adapter.

Façade

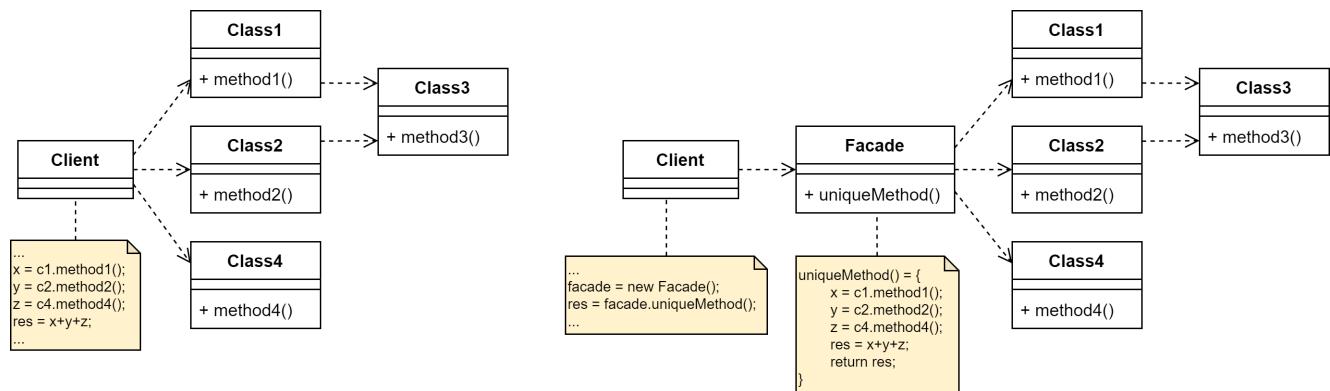
Scopo

Il fine del façade (facciata) è di fornire un'interfaccia più semplice di sottosistemi che presentano interfacce molto complesse e diverse/sparse tra loro. Promuove un accoppiamento debole fra cliente e sottosistema (la dipendenza non è transitiva), nascondendo al cliente le componenti complesse del sottosistema. Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema.

Casi d'uso

- Il client deve accedere a diverse classi molto differenti per compiere un'operazione;
- Si vuole rendere indipendente l'implementazione della classe client dalle varie classi utilizzate;
- Si vuole fornire un'interfaccia “context-specific” per funzionalità più generiche;
- Si vuole migliorare la leggibilità e l'usabilità di una libreria software mascherando l'interazione con componenti più complesse dietro un'unica (e spesso semplificata) API.

Diagramma UML



Partecipanti

- **Classi:** le classi che forniscono i singoli servizi alla classe Client;
- **Façade:** colui che semplifica l'accesso e l'utilizzo dei servizi di classi multiple al client tramite un'unica interfaccia;
- **Client:** colui che utilizza i servizi delle classi per compiere un'operazione.

Esempio

Nel seguente esempio possiamo vedere come l'utilizzo di una façade (`ComputerFacade`) semplifichi l'utilizzo delle sottoclassi al `Client`, che non è più tenuto a interagire con le singole classi e a dipendere da esse per implementare il comportamento voluto.

```
1 class CPU {
2     public void freeze() { ... }
3     public void jump(long position) { ... }
4     public void execute() { ... }
5 }
6
7 class Memory {
8     public void load(long position, byte[] data) { ... }
9 }
10
11 class HardDrive {
12     public byte[] read(long lba, int size) { ... }
13 }
14
15 class ComputerFacade {
16     private CPU processor;
17     private Memory ram;
18     private HardDrive hd;
19
20     public ComputerFacade() {
21         this.processor = new CPU();
22         this.ram = new Memory();
23         this.hd = new HardDrive();
24     }
25
26     public void start() {
27         processor.freeze();
28         ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
29         processor.jump(BOOT_ADDRESS);
30         processor.execute();
31     }
32 }
33
34 class Client {
35     public static void main(String[] args) {
36         ComputerFacade computer = new ComputerFacade();
37         computer.start();
38     }
39 }
```

Façade vs Adapter vs Wrapper

Il concetto di wrapper è molto generico, tendenzialmente viene definito come “un modulo software che ne riveste un altro”, concetto applicabile a metodi, tipi, classi, oggetti, etc.

Façade e Adapter sono di fatto dei wrapper, entrambi si basano su un’interfaccia, ma:

- Façade la semplifica;
- Adapter la converte/adatta ad un’altra.

Composite

Scopo

Lo scopo del composite è di permettere di trattare un gruppo di oggetti come se fossero l’istanza di un oggetto singolo. Il design pattern composite organizza gli oggetti in una struttura ad albero per rappresentare gerarchie parte-tutto. In tale albero i nodi sono delle composite e le foglie sono oggetti semplici.

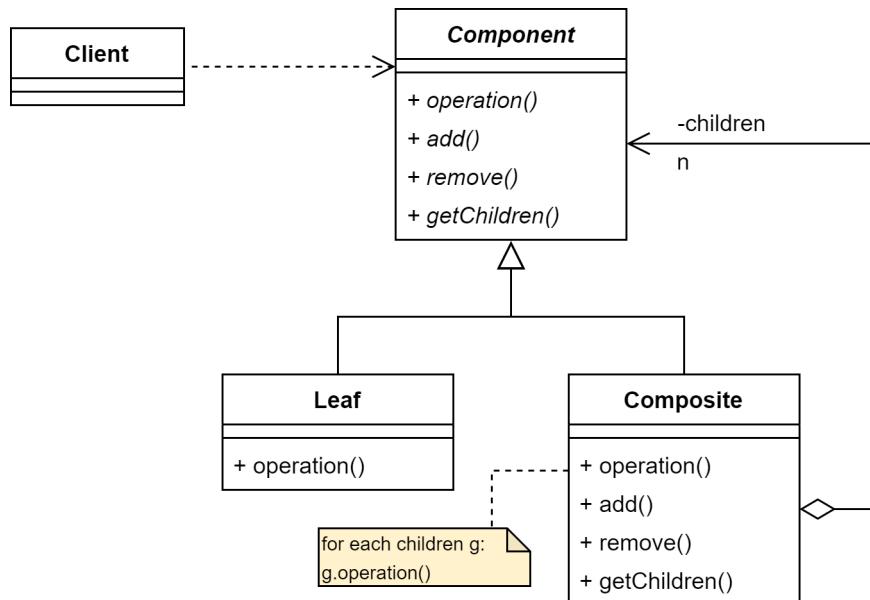
È utilizzato per dare la possibilità al cliente di manipolare oggetti singoli e composizioni in modo uniforme, senza necessità di distinzioni.

Il concetto fondamentale è che il programmatore manipola ogni oggetto dell’insieme dato nello stesso modo: sia esso un “raggruppamento” o un oggetto singolo.

Casi d’uso

- Il client ha la necessità di ignorare la differenza tra oggetti compositi e oggetti singoli;
- Si verificano casi in cui molti oggetti vengono utilizzati in modo simile, compreso il codice per la loro gestione.

Diagramma UML



Partecipanti

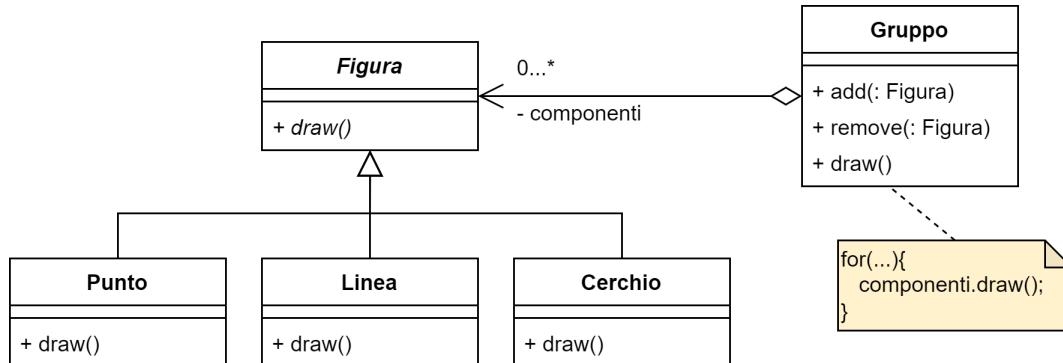
- **Client:** manipola gli oggetti attraverso l’interfaccia Component.
- **Component:** dichiara l’interfaccia per gli oggetti nella composizione, per accedervi e manipolarli, imposta un comportamento di default per l’interfaccia comune a tutte le classi. Può definire un’interfaccia per l’accesso al padre del componente e la implementa se è appropriata.
- **Composite:** definisce il comportamento per i componenti aventi figli, salva i figli e implementa le operazioni ad essi connesse nell’interfaccia Component.
- **Leaf:** definisce il comportamento degli oggetti primitivi, cioè delle foglie.

Esempio

Si consideri un programma di grafica che deve consentire di:

- Creare oggetti semplici (punto, linea, cerchio, etc.);
- Raggruppare dinamicamente oggetti semplici in oggetti composti (ad esempio: rettangolo = 4 linee, etc.);
- Si ha l’esigenza di trattare gli oggetti composti come se fossero oggetti semplici nel svolgere operazioni di spostamento, ridimensionamento, etc.

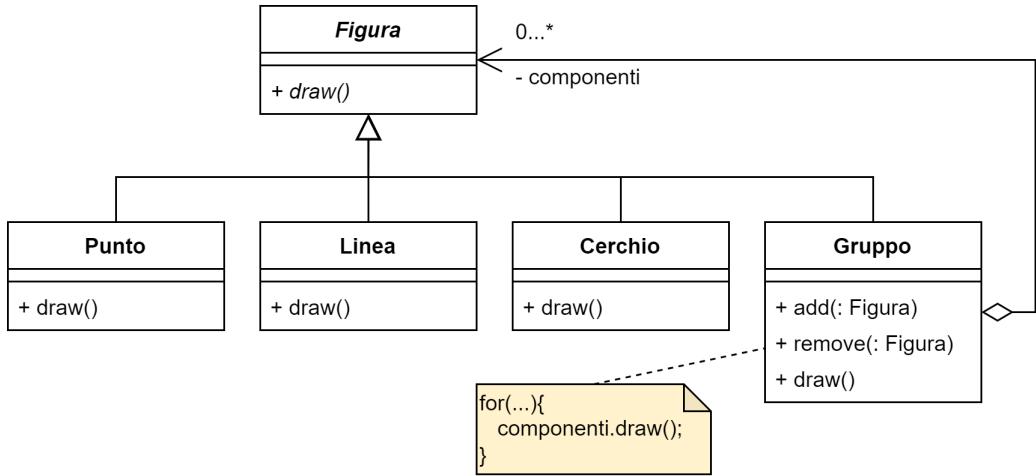
Un esempio che non fa uso del composite:



Questa implementazione non è ottimale in quanto la gestione dei gruppi (ad esempio Rettangolo) e delle figure primitive è diversa.

Facendo uso del pattern, **Gruppo** diventa sottoclasse di **Figura**. È un pattern “ricorsivo” in quanto **Gruppo** contiene **Figura** (e siccome un gruppo è una figura, ossia alcune figure sono gruppi; un gruppo può contenere gruppi).

Il composite semplifica il cliente, che tratta strutture composte e singoli oggetti allo stesso modo (permettendo così di passare oggetti composti anche dove oggetti primitivi sono richiesti). **Leaf** e **Composite** possono essere astratte ed avere sottoclassi; **Component** può essere un’interfaccia.



Un esempio di implementazione:

```

1 interface Graphic {
2     public void print();
3 }
4
5 class CompositeGraphic implements Graphic {
6     private List<Graphic> mChildGraphics = new ArrayList<Graphic>();
7
8     public void print() {
9         for (Graphic graphic : mChildGraphics) {
10             graphic.print();
11         }
12     }
13
14     public void add(Graphic graphic) {
15         mChildGraphics.add(graphic);
16     }
17
18     public void remove(Graphic graphic) {
19         mChildGraphics.remove(graphic);
20     }
21 }
22
23 class Ellisse implements Graphic {
24     public void print() {
25         System.out.println("Ellisse");
26     }
27 }
28
29 class Rettangolo implements Graphic {
30     public void print() {
31         System.out.println("Rettangolo");
32     }
33 }
  
```

Il codice del client potrebbe essere il seguente; in cui si può notare come per eseguire il print di una figura (**Graphic**) semplice o un gruppo di figure (**CompositeGraphic**) non ci siano differenze (si ignora la differenza tra oggetti compositi e singoli).

```
1  public static void main(String[] args) {
2
3      Ellisse ellisse1 = new Ellisse();
4      Ellisse ellisse2 = new Ellisse();
5      Rettangolo rettangolo1 = new Rettangolo();
6      Rettangolo rettangolo2 = new Rettangolo();
7
8      CompositeGraphic graphicEllissi = new CompositeGraphic();
9      CompositeGraphic graphicRettangoli = new CompositeGraphic();
10     CompositeGraphic graphicMixed = new CompositeGraphic();
11
12     graphicEllissi.add(ellisse1);
13     graphicEllissi.add(ellisse2);
14
15     graphicRettangoli.add(rettangolo1);
16     graphicRettangoli.add(rettangolo2);
17
18     graphicMixed.add(graphicEllissi);
19     graphicMixed.add(graphicRettangoli);
20
21     // Stampa tutte le figure
22     graphicMixed.print();
23
24     // Stampa solo le elissi
25     graphicEllissi.print();
26
27     // Stampa solo il primo rettangolo
28     rettangolo1.print();
29 }
```

Decorator

Scopo

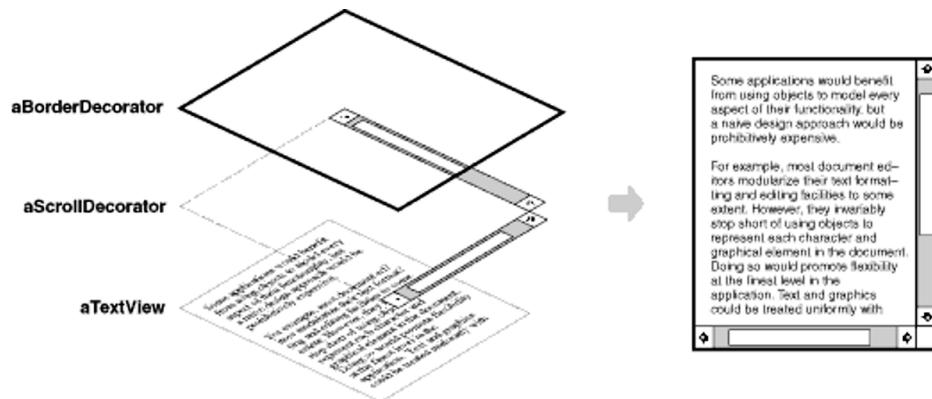
Consente di aggiungere nuove funzionalità ad oggetti già esistenti. Questo avviene costruendo una nuova classe decoratore che “avvolge” l’oggetto originale.

Al costruttore del decoratore si passa come parametro l’oggetto originale. È altresì possibile passarvi un differente decoratore. In questo modo, più decoratori possono essere concatenati l’uno all’altro, aggiungendo così in modo incrementale funzionalità alla classe concreta (che è rappresentata dall’ultimo anello della catena).

Il decorator si pone come valida alternativa all’uso dell’ereditarietà singola o multipla (alternativa più flessibile all’uso delle sottoclassi per estendere le funzionalità).

Con l’ereditarietà, infatti, l’aggiunta di funzionalità avviene staticamente secondo i legami definiti nella gerarchia di classi e non è possibile ottenere al run-time una combinazione arbitraria delle funzionalità, né la loro aggiunta/rimozione.

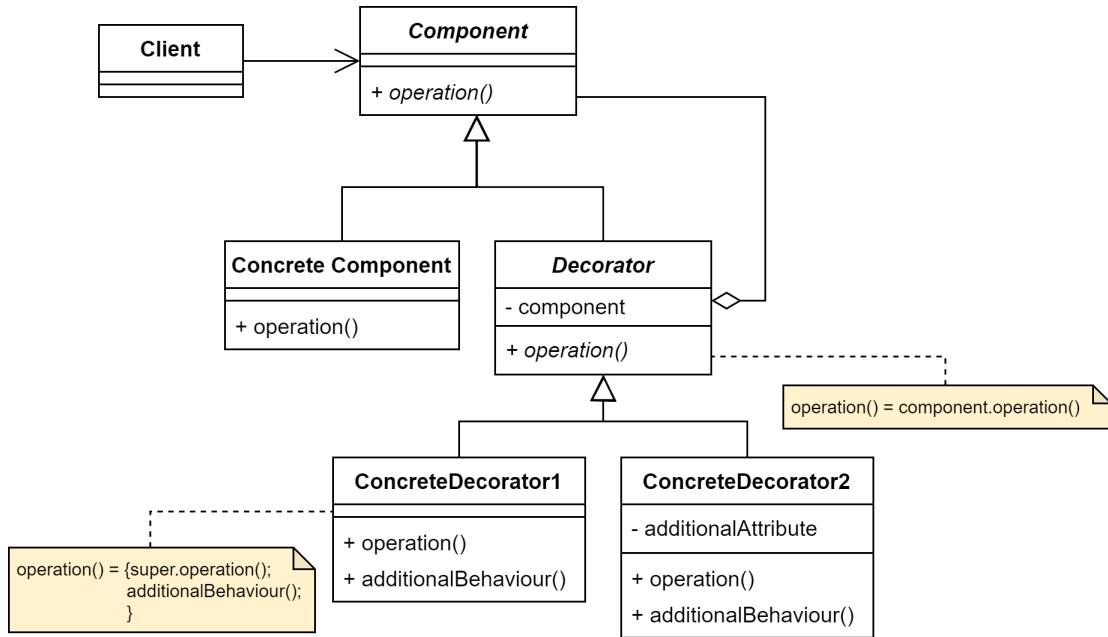
Decoratore	Ereditarietà
Dinamico (run-time)	Statico (compile-time)
Non vincolante per il cliente	Vincolante per il cliente, che non può inventarsi combinazioni non previste
Aggiunge responsabilità ad un singolo oggetto	Aggiunge responsabilità a tutte le istanze di una classe
Evita l’esplosione combinatoria	Inclina all’esplosione combinatoria



Casi d’uso

- Si vuole estrema libertà a run-time relativamente alla gestione (aggiunta/rimozione) di funzionalità agli oggetti/istanze;
- Il numero di combinazioni di comportamenti distinti per i vari oggetti possibili è troppo numeroso da gestire “staticamente” tramite una gerarchia di classi.

Diagramma UML

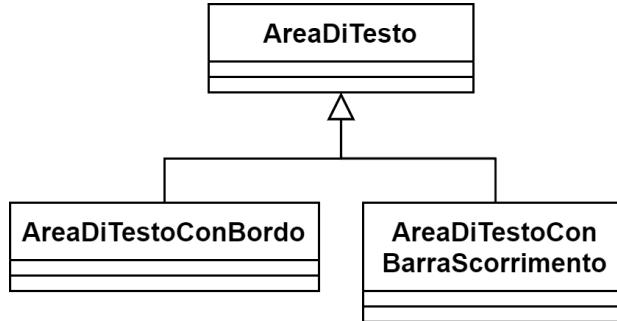


Partecipanti

- **Component**: definisce l'interfaccia dell'oggetto a cui verranno aggiunte nuove funzionalità;
- **ConcreteComponent**: definisce l'oggetto concreto al quale aggiungere le funzionalità;
- **Decorator**: mantiene un riferimento a un'istanza di **Component** e definisce un'interfaccia ad esso conforme;
- **ConcreteDecorator**: aggiunge funzionalità al **Component**, implementando l'interfaccia definita da **Decorator**.

Esempio

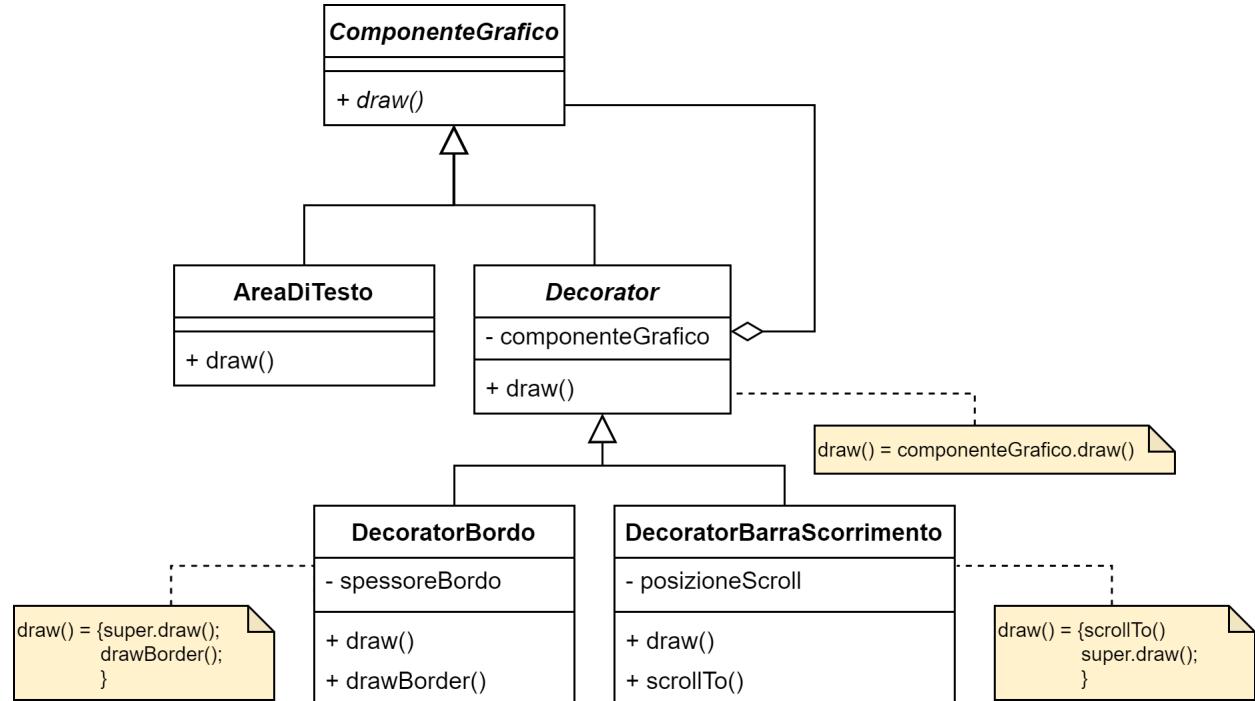
In ambiente grafico, se vogliamo due o più tipologie di **AreaDiTesto**, potremmo, tramite l'ereditarietà, costruire la seguente gerarchia di classi:



Così facendo posso creare le istanze della classe voluta. Ma cosa succede se voglio un'area di testo con bordo e barra di scorrimento? creo un'altra sottoclassa? si ha un'esplosione combinatoria. Contrariamente all'approccio statico appena descritto, si può seguire un approccio dinamico, racchiudendo l'oggetto da "decorare" in un altro oggetto, responsabile di gestirne la decorazione: il decorator.

Il decorator ha il compito di trasferire/delegare e aggiungere funzionalità all'oggetto. Il decorator ha l'interfaccia conforme all'oggetto decorato, e quindi è trasparente al client che può non sapere se sta parlando con un oggetto decorato o non decorato.

Adottando il decorator al nostro esempio otteniamo:



Possiamo ora ottenere una catena di decorazioni, con alla fine un'istanza di `AreaDiTesto`.

```
1 ComponenteGrafico x =  
2     new DecoratorBarraScorrimento(  
3         new DecoratorBordo(  
4             new AreaDiTesto()  
5         )  
6     );
```



Bridge

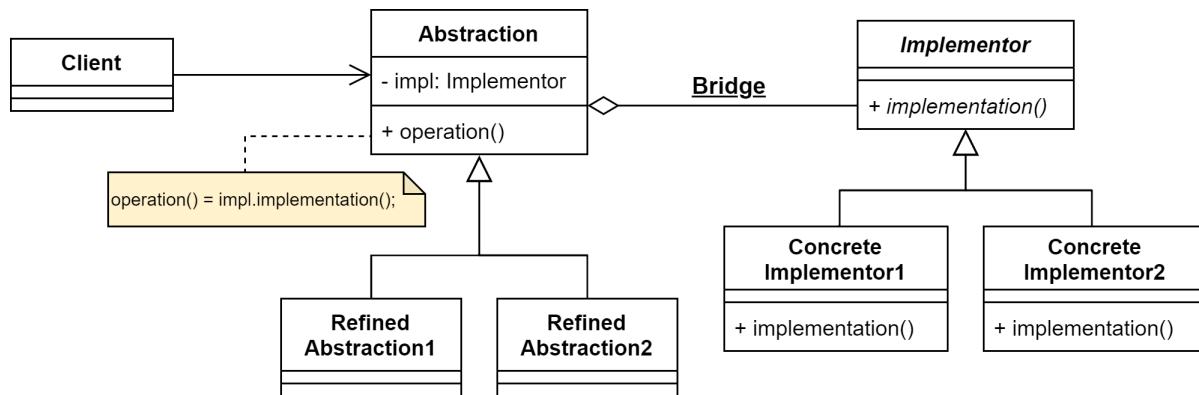
Scopo

Lo scopo del pattern bridge (“ponte”) è di separare l’interfaccia di una classe dalla sua implementazione. In tal modo è possibile sfruttare l’ereditarietà per far evolvere l’interfaccia o l’implementazione in modo separato e indipendente. Il bridge quindi disaccoppia (“decoupling”) un’astrazione dalla sua implementazione, rendendolo uno meno dipendente dall’altro.

Casi d’uso

- Si vuole dividere e organizzare una classe monolitica che ha più varianti di alcune funzionalità;
- Si vuole estendere con maggiore libertà una classe in diverse modalità indipendenti;
- Si vuole cambiare implementazione a run-time;

Diagramma UML



Partecipanti

- **Abstraction:** definisce l’interfaccia astratta e mantiene una referenza dell’**Implementor**;
- **Implementor:** definisce l’interfaccia per le classi che implementano le operazioni;
- **Concrete Implementor:** implementa l’interfaccia **Implementor** e le singole operazioni;
- **Refined Abstraction:** estende l’interfaccia definita dall’**Abstraction**.

Esempio

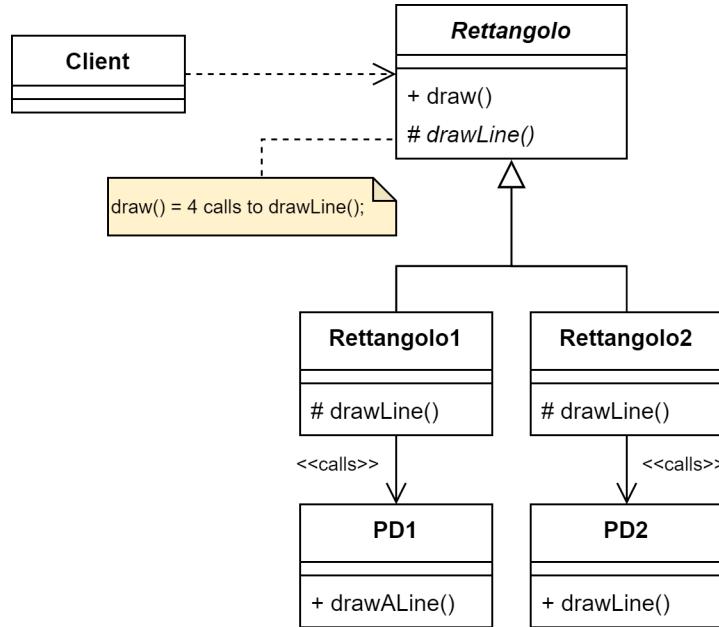
Si richiede un programma che permetta di disegnare rettangoli, usando in alternativa uno dei due programmi di disegno:

- PD1: `drawALine(x1, y1, x2, y2)`
- PD2: `drawLine(x1, x2, y1, y2)`

Si vuole far sì che una volta creato un rettangolo, non sia necessario distinguere che questo sia stato creato con PD1 o PD2. Il codice del cliente è il seguente:

```
1 Rettangolo[] r = new Rettangolo[...];
2 r[0] = new Rettangolo1(...); // di PD1
3 r[1] = new Rettangolo2(...); // di PD2
4 r[2] = new Rettangolo2(...); // di PD2
5 ...
6 for(i = ...){
7     r[i].draw();
8 }
```

Abbiamo quindi 2 tipologie di rettangoli, uno usa PD1 e l'altro PD2; una prima soluzione è astrarre `Rettangolo`; all'istanziazione so se istanziare `Rettangolo1` o `Rettangolo2`.



La soluzione sembra funzionare; ma immaginiamo di voler disegnare anche cerchi, quadrati, etc. Avremmo quindi per ciascuno di essi una funzione `draw()` del PD1 e del PD2, similmente ai rettangoli; consentendo sempre di astrarre le figure al cliente.

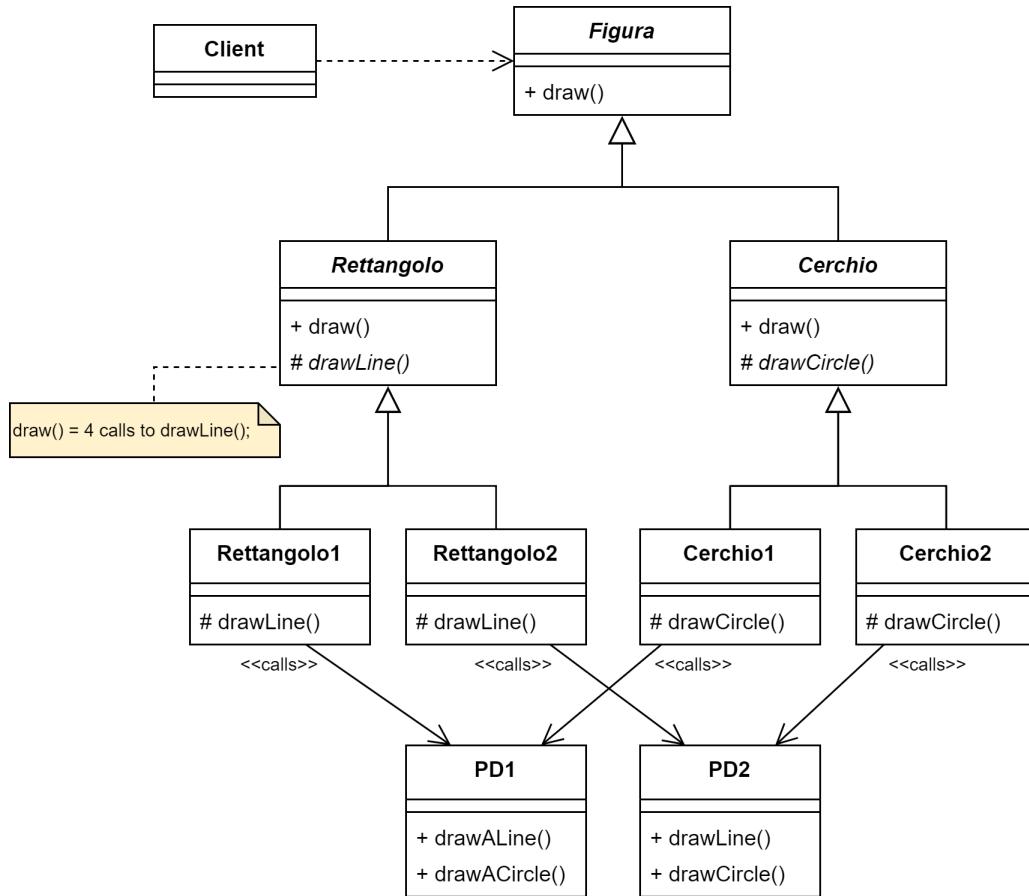
Aggiungendo ad esempio il Cerchio avremmo casi come il seguente codice del cliente:

```

1 Figura[] f = new Figura[...];
2 f[0] = new Rettangolo1(...); // di PD1
3 f[1] = new Rettangolo2(...); // di PD2
4 f[2] = new Cerchio1(...); // di PD1
5 f[3] = new Cerchio1(...); // di PD1
6 f[4] = new Cerchio2(...); // di PD2
7 ...
8 for(i = ...){
9     f[i].draw();
10 }

```

Con la seguente organizzazione delle classi:

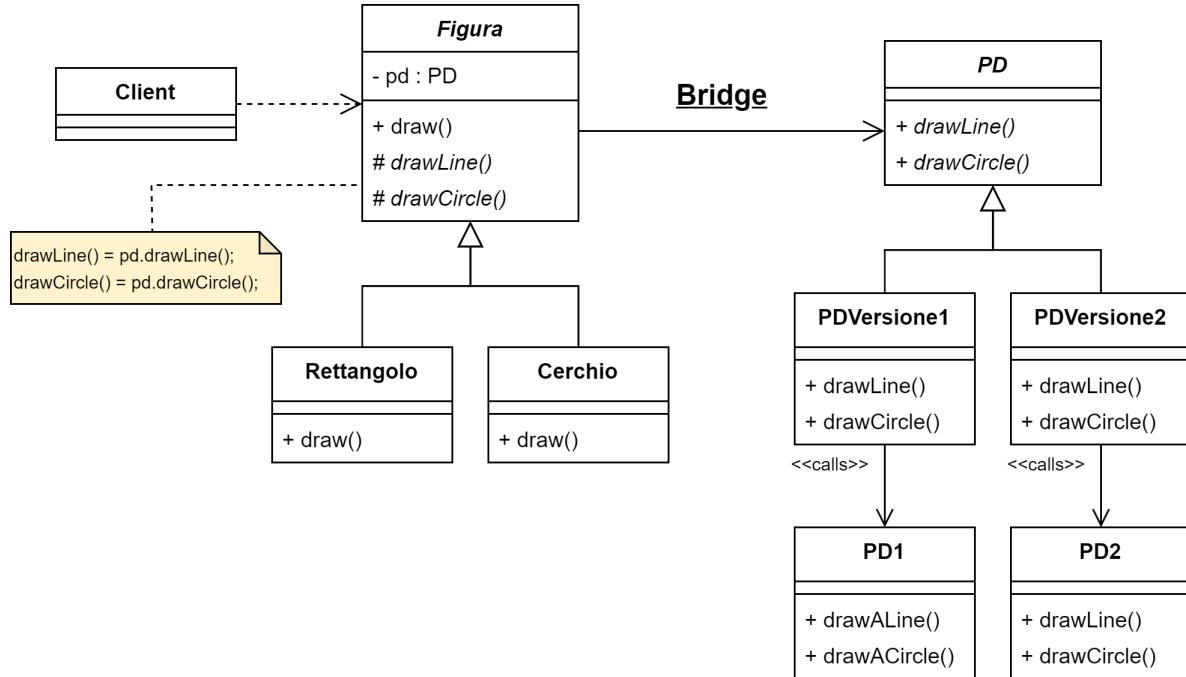


La soluzione sembra ok, ma se devo gestire molte figure? per ciascuna devo aggiungere una piccola gerarchia: una classe astratta e una sottoclasse per ciascun PD. E se ho molti altri programmi di disegno PD? le gerarchie diventano più estese. Di per sé la soluzione funziona ma può esplodere a livello combinatorio.

L'alternativa duale è scomporre prima per tipologia di PD e poi per tipo di figura; scambiando l'ordine. Non cambia molto, l'esplosione combinatoria è la stessa.

Queste due soluzioni presentano alto accoppiamento (non fra classi ma fra le astrazioni delle figure e le implementazioni - o viceversa nella soluzione duale).

C'è ridondanza e non c'è indipendenza tra le due. Appreso il problema, analizziamo una soluzione: usiamo un bridge pattern per separare le astrazioni sulle figure dalle loro implementazioni dei vari PD.



Il codice del cliente diventa, ad esempio:

```

1 Figura[] f = new Figura[...];
2 PD p1 = new PDVersione1();
3 PD p2 = new PDVersione2();
4 f[0] = new Rettangolo(p1); // di PD1
5 f[1] = new Rettangolo(p2); // di PD2
6 f[2] = new Cerchio(p1); // di PD1
7 ...
8 for(i = ...){
9     f[i].draw();
10 }
  
```

Quando si crea un'istanza di **Figura** specifico anche con quale implementazione. Questa può essere omessa e impostarne una come scelta di default.

Proxy

Scopo

Fornire un surrogato o un segnaposto per un oggetto, al fine di controllarne l'accesso.

Un proxy, nella sua forma più generale è una classe che fa da interfaccia per qualcosa' altro. In breve, un proxy può essere visto come un wrapper che viene chiamato dal cliente per accedere a oggetti molteplici. Si noti come nel proxy, oltre a richiami a servizi dell'oggetto "wrappato", può essere fornita logica aggiuntiva, ad esempio del codice per il controllo di alcune precondizioni.

Per il cliente, l'uso del proxy e dell'oggetto reale è molto simile, in quanto implementano la stessa interfaccia.

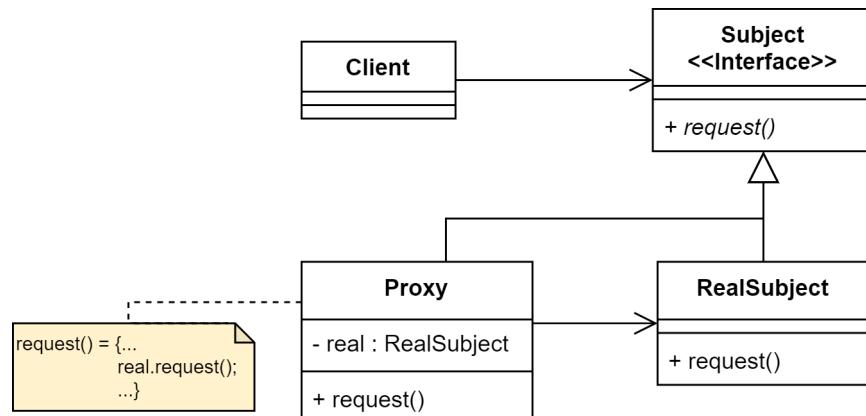
Esistono diverse tipologie di proxy:

- **Virtuale:** gestisce la creazione su richiesta di oggetti "costosi/pesanti"; si parla di "lazy initialization";
- **Protezione:** gestisce diritti di accesso per diversi oggetti;
- **Remoto:** rappresenta localmente un oggetto di uno spazio diverso;
- **Riferimento intelligente:** sostituisce un puntatore.

Casi d'uso

- Si vuole controllare l'accesso a un oggetto (sicurezza e/o efficienza);
- Si vogliono fornire funzionalità aggiuntive ad un oggetto quando gli si accede.

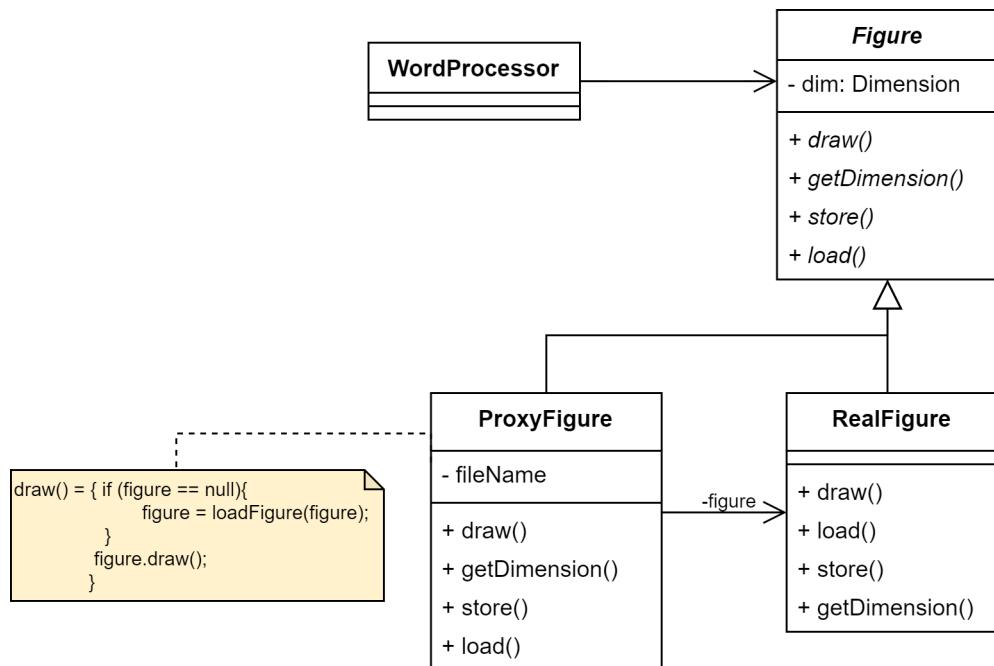
Diagramma UML



Partecipanti

- **Subject:** dichiara l'interfaccia per i singoli servizi; il Proxy deve essere congruo a tale interfaccia;
- **Proxy:** mantiene una referenza che punta all'istanza di un servizio reale (**RealSubject**). Dopo che il proxy ha finito i suoi compiti (ad esempio: lazy initialization, logging, access control, caching, etc.), inoltra la vera richiesta all'oggetto che fornisce il servizio concreto;
- **RealSubject:** classe che implementa gli effettivi servizi/business logic.

Esempio



Flyweight

Scopo

Permettere di separare la parte variabile di una classe dalla parte che può essere riutilizzata, in modo tale da condividere quest'ultima fra differenti istanze.

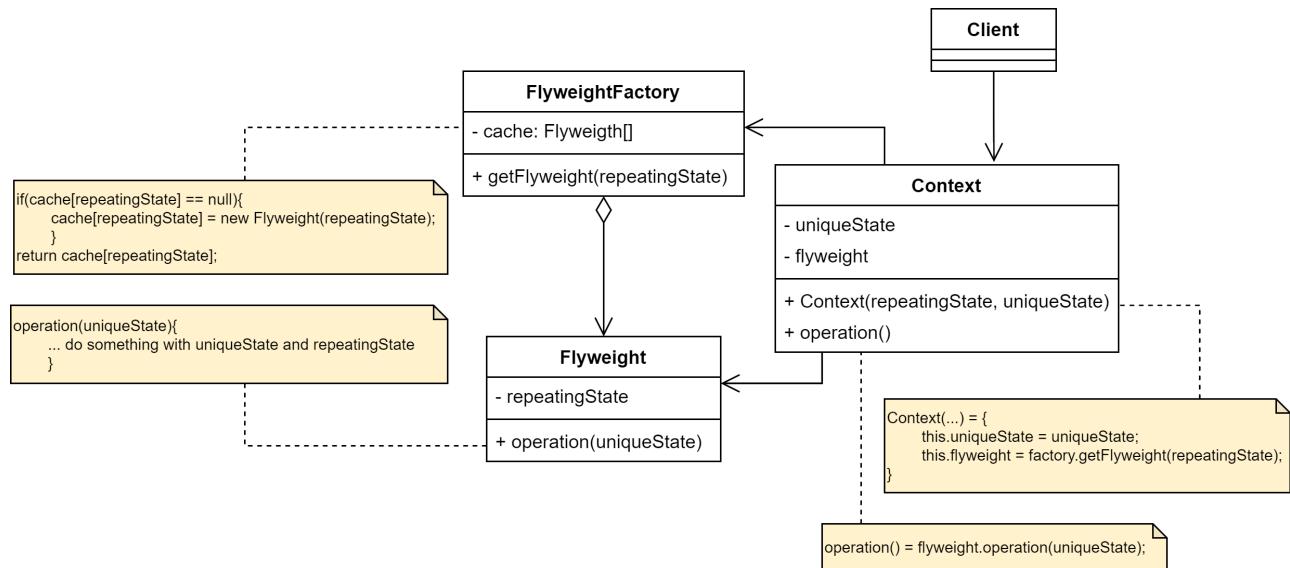
L'oggetto flyweight deve essere un oggetto immutabile, per permetterne la condivisione tra diversi client e thread.

Si ottiene così un risparmio in utilizzo della memoria, visto che le parti uguali tra gli oggetti sono condivise e non replicate.

Casi d'uso

- Si hanno molte istanze della stessa classe che variano di poco o che hanno la maggioranza degli attributi sempre uguale;

Diagramma UML

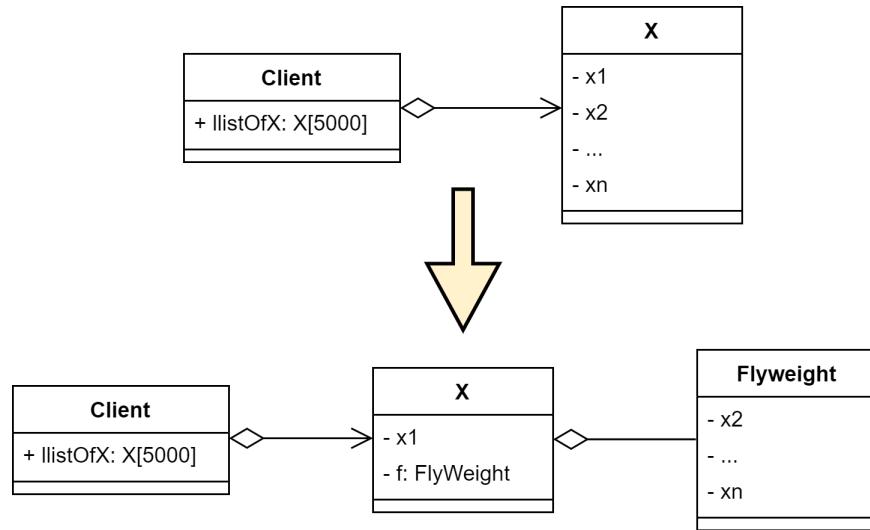


Partecipanti

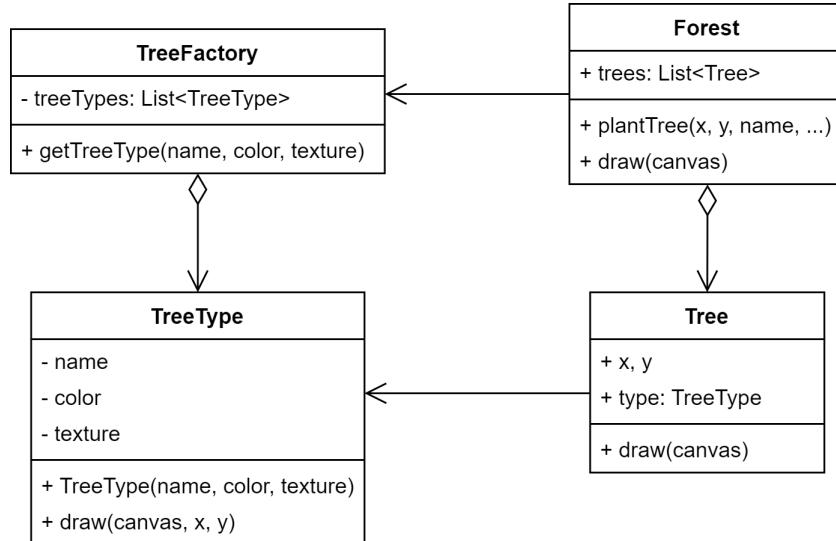
- **Context**: contiene lo stato effettivo dell'oggetto (composto dallo stato unico “intrinseco” e da una versione di quelli condivisi - contenuta nel flyweight); unico tra tutti gli oggetti;
- **FlyweightFactory**: gestisce un insieme di flyweight esistenti e ne crea di nuovi quando necessari;
- **Flyweight**: contiene una porzione dello stato della classe/istanza iniziale (**Context**), condiviso con altre istanze. Questa porzione di stato è detta estrinseca.

Esempio

Nel seguente esempio generico, si ipotizzi di avere una classe X con n attributi, l'attributo x_1 è quasi sempre diverso (stato estrinseco), mentre gli attributi x_2, \dots, x_n sono spesso simili (pochi/condivisi - stato estrinseco). Invece di creare molte istanze di X con gli attributi simili duplicati, è buona idea riorganizzare i parametri x_2, \dots, x_n in una classe flyweight, istanziata poche volte (per i vari casi) e utilizzarla come attributo di X:



Un esempio più complesso è il seguente: si vuole ridurre l'uso di memoria nel rendering di milioni di oggetti “albero” in un canvas. Applicando il flyweight pattern, si estraе lo stato estrinseco dalla classe Tree e lo si sposta nella classe flyweight TreeType.



Implementazione dell'esempio:

```
1 class TreeType {
2     public String name;
3     public Color color;
4     public Texture texture;
5
6     public TreeType(name, color, texture){ ... }
7
8     public void draw(canvas, x, y){ ... }
9 }
10
11 class TreeFactory {
12     public List<TreeTypes> types = ...
13
14     public TreeType getTreeType(name, color, texture){
15         type = types.find(name, color, texture);
16         if(type == null){
17             type = new TreeType(name, color, texture);
18             types.add(type);
19         }
20         return type;
21     }
22 }
23
24 class Tree {
25     public float x, y;
26     public TreeType type;
27
28     public Tree(x, y, type){ ... }
29
30     public void draw(canvas, this.x, this.y){ ... }
31 }
32
33 class Forest {
34     public List<Tree> trees = ...
35     public TreeFactory = ...
36
37     public void plantTree(x, y, name, color, texture){
38         type = TreeFactory.getTreeType(name, color, texture);
39         tree = new Tree(x, y, type);
40         trees.add(tree);
41     }
42
43     public void draw(canvas){ foreach tree ... }
44 }
```

Factory Method

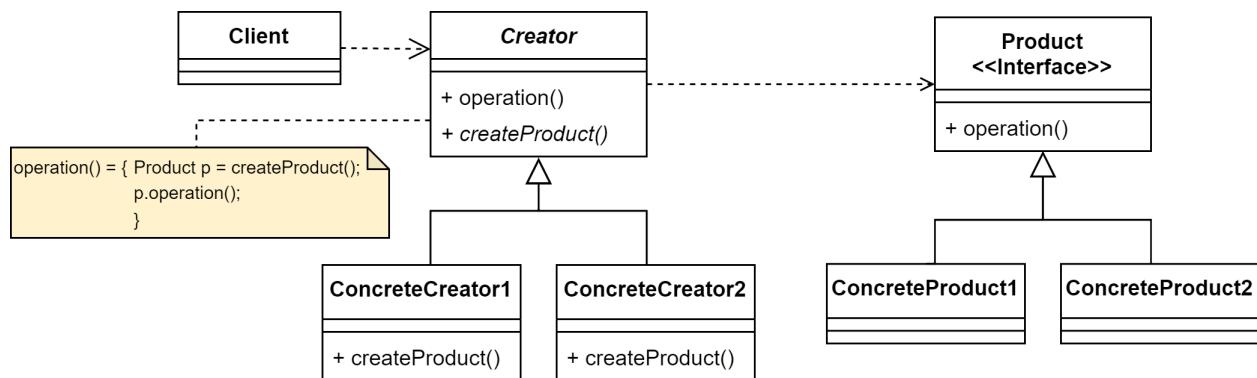
Scopo

L'obiettivo del pattern è di delegare l'istanziazione di una classe a una propria sottoclasse, che deciderà quale istanza creare e quale costruttore utilizzare. La creazione di un oggetto può spesso richiedere processi complessi la cui collocazione all'interno della classe di composizione potrebbe non essere appropriata; il pattern indirizza questo problema fornendo un metodo separato delegato alla creazione. Il factory method è tipicamente utilizzato nei framework object-oriented.

Casi d'uso

- La creazione di un oggetto preclude il suo riuso senza una significativa duplicazione di codice;
- La creazione di un oggetto richiede l'accesso ad informazioni o risorse che non dovrebbero essere contenute/esposte nella classe di composizione/Client;
- La gestione del ciclo di vita degli oggetti deve essere centralizzata in modo da assicurare un comportamento coerente all'interno dell'applicazione.

Diagramma UML



Partecipanti

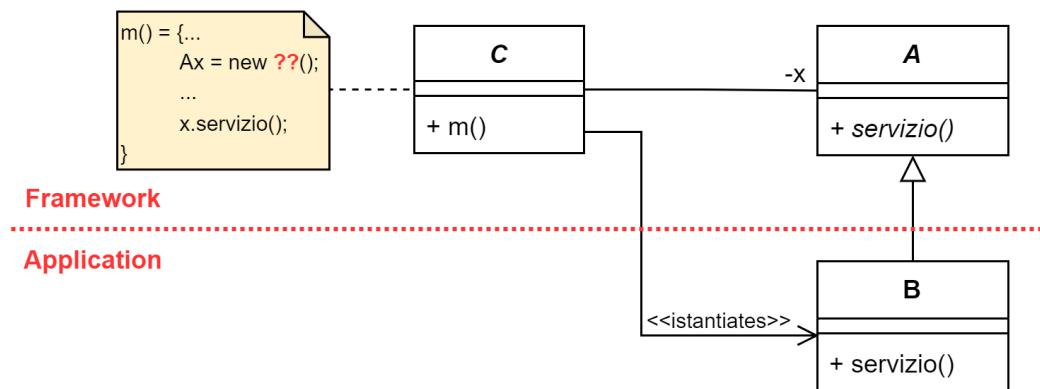
- **Creator**: dichiara il factory method `createProduct()` e ritorna il nuovo prodotto corretto. Il factory method può essere dichiarato astratto così da forzare le sottoclassi a implementare la propria versione del metodo.
- **Concrete Creators**: sovrascrivono il factory method `createProduct()` così da ritornare il prodotto corretto;

- **Product:** dichiara l'interfaccia che è comune a tutti gli oggetti che possono venir creati dal **Creator** e le sue sottoclassi;
- **Concrete Products:** le differenti implementazioni dell'interfaccia Product.

Esempio

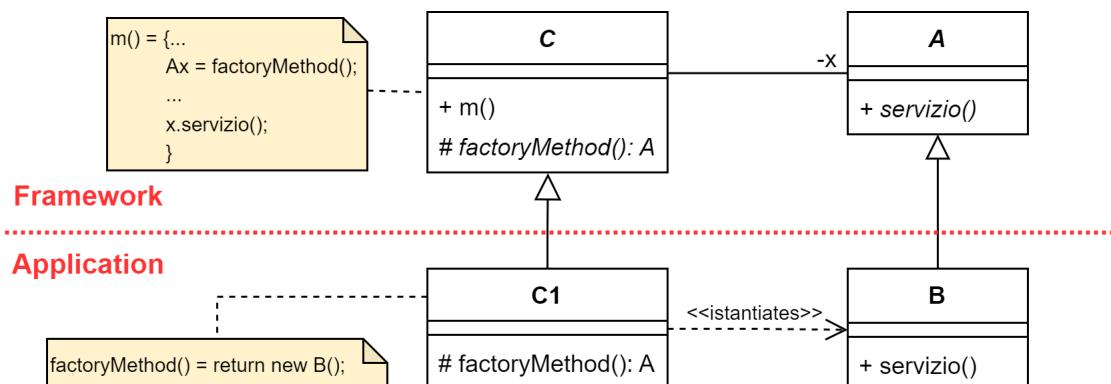
Si analizzerà un esempio tipico, relativo ai framework. Il programmatore del framework scrive un metodo in una classe C che deve creare, in un certo punto preciso, un'istanza di una sottoclasse B di un'altra classe astratta A.

Quale sottoclasse (B) utilizzare è ignoto, perché non deciso dal programmatore del framework ma dal programmatore dell'applicazione (che fa uso del framework). B viene addirittura scritta dopo, se il framework è di tipo white-box.



Soluzioni:

- Il programmatore del framework delega tutto al programmatore dell'applicazione... non è una soluzione;
- Il programmatore del framework scrive quello che può sulla base delle informazioni a lui fornite: un'invocazione a un metodo astratto **factoryMethod()** che deve creare l'istanza, e chiede al programmatore dell'applicazione di scrivere, oltre alla classe B, anche una sottoclasse C1 di C che implementi il metodo **factoryMethod()**.



Così facendo, invece di avere la conoscenza esplicita sulla creazione di una classe (`C`), la delego a una sottoclassa `C1`. Si noti come il `factoryMethod()` potrebbe non essere astratto ma fornire un'implementazione di default (eventualmente sovrascritta nelle sottoclassi).

Esempio di implementazione:

```
1 public interface IPerson
2 {
3     string GetName();
4 }
5
6 public class Villager : IPerson
7 {
8     public string GetName()
9     {
10         return "Village Person";
11     }
12 }
13
14 public class CityPerson : IPerson
15 {
16     public string GetName()
17     {
18         return "City Person";
19     }
20 }
21
22 public class IslandPerson : IPerson
23 {
24     public string GetName()
25     {
26         return "Island Person";
27     }
28 }
```

Abbiamo la gerarchia sopra descritta, tre tipologie di `IPerson`. Nel codice del cliente, senza uso di un factory method, dovremmo distinguere la tipologia di persona, prima di chiamare il costruttore corretto, come ad esempio:

```
1 public enum PersonType{Rural, Urban, Island}
2
3 public static void main(String args[]){
4     ...
5     PersonType type = getTypeFromUser();
6     IPerson newPerson;
7
8     if(type == PersonType.Rural){
9         newPerson = new Villager();
10    } else if(type == PersonType.Urban){
11        newPerson = new CityPerson();
12    } else if(type == PersonType.Island){
13        newPerson = new IslandPerson();
14    }
}
```

Questa implementazione è chiaramente non ideale. Con l'uso di un factory method (`getPerson()`), il codice del cliente diventerebbe il seguente:

```
1 public enum PersonType{Rural, Urban, Island}
2
3 public class Factory
4 {
5     public IPerson GetPerson(PersonType type)
6     {
7         switch (type)
8         {
9             case PersonType.Rural:
10                 return new Villager();
11             case PersonType.Urban:
12                 return new CityPerson();
13             default:
14                 throw new NotSupportedException();
15         }
16     }
17 }
18
19 public static void main(String args[]){
20     ...
21     PersonType type = getTypeFromUser();
22     IPerson newPerson;
23
24     Factory personFactory = new Factory();
25
26     newPerson = personFactory.GetPerson(type);
27 }
```

In base al tipo passato all'oggetto `Factory`, stiamo restituendo l'oggetto concreto corretto. Qui, la scelta di quale sottoclasse utilizzare è delegata al factory method `getPerson()`.

L'aggiunta di nuove tipologie di persone, oltre che essere più semplice (è necessario estendere l'enumeratore e il factory method e basta, invece che tutti i controlli richiesti in n sezioni di codice dove è richiesta una `IPerson`).

Il codice del `main()` (e qualsiasi porzione di codice che crea istanze di `IPerson`) rimarrebbe invariato in caso di aggiunta/rimozione di sottoclassi di `IPerson`.

Abstract Factory

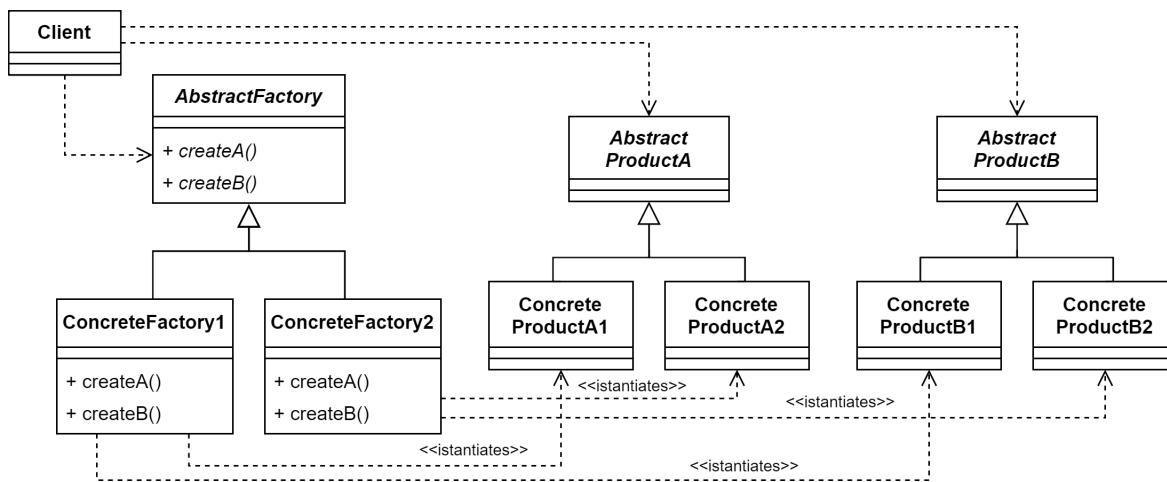
Scopo

Il pattern fornisce un'interfaccia per creare famiglie di oggetti connessi o dipendenti tra loro, in modo che non ci sia necessità da parte dei client di specificare i nomi delle classi concrete all'interno del proprio codice. In questo modo si permette che un sistema sia indipendente dall'implementazione degli oggetti concreti e che il client, attraverso l'interfaccia, utilizzi diverse famiglie di prodotti. Rispetto al factory method, qui l'enfasi è sulle famiglie di oggetti correlati.

Casi d'uso

- Si vuole rendere il sistema/client indipendente da come gli oggetti vengono creati, composti e rappresentati;
- Si vuole permettere la configurazione del sistema come scelta tra diverse famiglie di oggetti/prodotti;
- Si vuole che gli oggetti organizzati in famiglie siano vincolati ad essere utilizzati con oggetti della stessa famiglia;
- Si vuole fornire una libreria di classi mostrando solo le interfacce e nascondendo le implementazioni.

Diagramma UML



Partecipanti

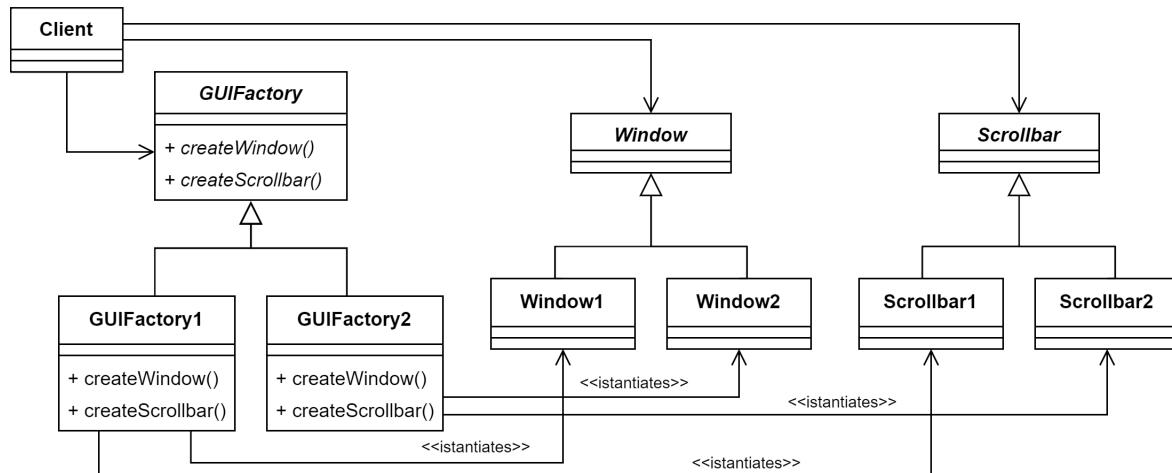
- **AbstractFactory:** dichiara l'interfaccia per le operazioni che creano gli oggetti;
- **ConcreteFactory:** implementa le operazioni per creare gli oggetti concreti;
- **AbstractProduct:** dichiara l'interfaccia per la famiglia di oggetti producibili;
- **ConcreteProduct:** implementa l'interfaccia astratta **AbstractProduct** e definisce l'oggetto prodotto che deve essere creato dalla factory concreta corrispondente;
- **Client:** utilizza solo le interfacce dichiarate **AbstractFactory** e **AbstractProduct** per gestire una famiglia di prodotti.

Esempio

Stiamo progettando un framework per interfacce grafiche multipiattaforma. Abbiamo la seguente situazione:

- se eseguito su **architettura1**, abbiamo **Window1** e **ScrollBar1**;
- se eseguito su **architettura2**, abbiamo **Window2** e **ScrollBar2**;
- ... e così via.

Indipendentemente dalla piattaforma, il client (che istanzierà quelle classi) non deve sapere quali classi istanzia; bisogna evitare, ad esempio, che il client sbagli e accoppi **Window1** e **ScrollBar2**. Il client deve poter astrarre dalla singola architettura.



Senza fabbrica astratta abbiamo:

```
1 Window w = new Window1();
2 ...
3 ScrollBar s = new ScrollBar1();
```

Con la fabbrica astratta abbiamo:

```
1 Factory f = new GUIFactory1();
2 ...
3 Window w = f.createWindow();
4 ...
5 Scrollbar s = f.createScrollbar();
```

Senza Abstract Factory	Con Abstract Factory
Client deve conoscere Window1 e ScrollBar1	Client chiede una fabbrica “di tipo 1”
Client crea una Window1	Client chiede alla fabbrica una Window
Client crea una ScrollBar1	Client chiede alla fabbrica una ScrollBar
Client ha la responsabilità di accoppiare Window1 e ScrollBar1	La responsabilità di accoppiare Window1 e ScrollBar1 è delegata alla fabbrica

Si ottiene separazione delle responsabilità: si disaccoppia la responsabilità della creazione (nella fabbrica) dalla responsabilità dell’uso (nel client). Si dice “astratta” perché il cliente conosce solo le classi astratte.

Factory Method vs Abstract Factory

Questi due pattern vengono spesso confusi. La differenza sostanziale tra il factory method e l’abstract factory sta nel fatto che, di fatto, il primo è un metodo, il secondo è un oggetto.

Il factory method è un semplice metodo utilizzato per creare oggetti in/di una classe. Tipicamente viene aggiunto nella classe principale.

L’abstract factory, invece, crea una classe base con metodi astratti che definiscono gli oggetti che possono venir creati. Ciascuna factory (concreta) deriva dalla classe base e può creare/avere la propria implementazione per ciascun oggetto.

Il factory method è pensato per creare un oggetto singolo (esponendo al cliente un metodo), l’abstract factory è pensata per creare una famiglia di oggetti dipendenti (esponendo una famiglia di oggetti correlati).

Il factory method nasconde la costruzione/implementazione di un singolo oggetto, l’abstract factory di una intera famiglia di oggetti.

Si noti come l’abstract factory è tipicamente implementata mediante l’uso molteplice del factory method.

L’abstract factory utilizza la composizione per delegare la responsabilità di creare un oggetto mentre il factory method utilizza l’ereditarietà da classe a sottoclasse.

Singleton

Scopo

Fornire la certezza che una classe abbia un'unica istanza e fornire un punto di accesso globale a quest'ultima. Si delegano alla classe le responsabilità di consentire un'unica istanza (impedire creazioni di più istanze) e consentirne l'accesso.

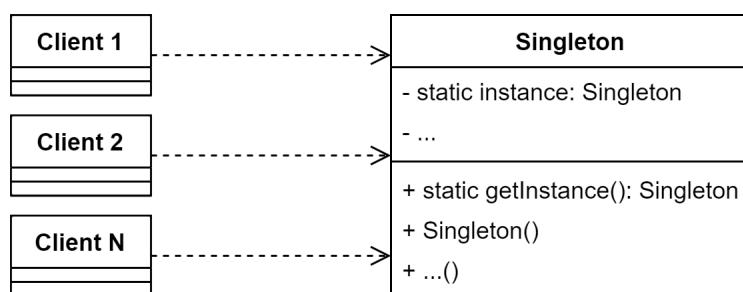
L'implementazione più semplice di questo pattern prevede che la classe singleton abbia un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe. La classe fornisce inoltre un metodo “getter” statico che restituisce l'istanza della classe (sempre la stessa), creandola preventivamente o alla prima chiamata del metodo, e memorizzandone il riferimento in un attributo privato anch'esso statico.

Il secondo approccio si può classificare come basato sul principio della lazy initialization (letteralmente: “inizializzazione pigra”) in cui la creazione dell'istanza della classe viene rimandata nel tempo e messa in atto solo quando ciò diventa strettamente necessario (al primo tentativo di uso).

Casi d'uso

- Ci si vuole assicurare che una certa classe venga istanziata una singola volta (o che in un certo istante abbia una sola istanza attiva);
- Si vuole fornire un accesso globale a un'istanza unica di una classe.

Diagramma UML



Partecipanti

- **Singleton:** la classe unica durante l'esecuzione;
- **Clienti:** i clienti dei servizi della classe unica Singleton.

Esempio

Il singleton è un pattern molto orientato al codice, vediamo un esempio di implementazione con “lazy initialization”:

```
1 public class MySingleton {  
2     private static MySingleton istanza = null;  
3  
4     // Il costruttore privato impedisce l'istanza di oggetti  
5     // da parte di classi esterne  
6     private MySingleton() {...}  
7  
8     // Metodo della classe impiegato per accedere al singleton  
9     public static synchronized MySingleton getMySingleton() {  
10         if (istanza == null) {  
11             istanza = new MySingleton();  
12         }  
13         return istanza;  
14     }  
15 }
```

Una versione non lazy (con “early initialization”) è:

```
1 public class MySingleton {  
2     private static MySingleton istanza = new MySingleton();  
3  
4     // Il costruttore privato impedisce l'istanza di oggetti  
5     // da parte di classi esterne  
6     private MySingleton() {...}  
7  
8     // Metodo della classe impiegato per accedere al singleton  
9     public static synchronized MySingleton getMySingleton() {  
10         return istanza;  
11     }  
12 }
```

Bisogna comunque precisare che questo tipo di approccio risolutivo potrebbe presentare dei difetti (per esempio questa implementazione non è thread safe), infatti il progettista della classe che è chiamato tra l'altro a rispettare il pattern singleton, potrebbe, per esempio, erroneamente dichiarare all'interno della classe un metodo non statico con visibilità public che restituisca un'istanza di un nuovo oggetto della stessa classe, ed ecco che allora il vincolo viene violato in contraddizione a quanto sopra detto.

Qualora questo metodo venga chiamato da più thread e si verifichi una race-condition non opportunamente gestita (tramite meccanismi di sincronizzazione), potremmo avere due o più istanze del singleton.

In definitiva questo significa che in realtà per adempiere pienamente bisogna gestire il tutto con il meccanismo delle eccezioni.

Prototype

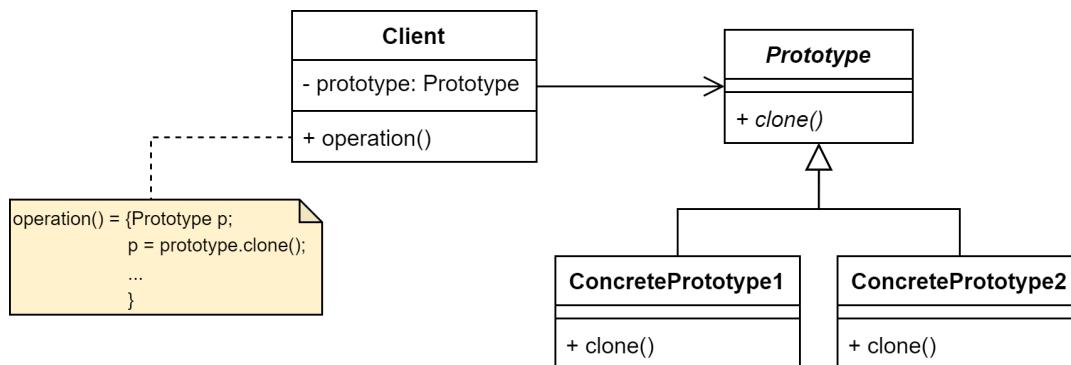
Scopo

Il fine del pattern è di rendere possibile la creazione di nuovi oggetti clonando un oggetto iniziale, detto prototipo. Si ottiene il vantaggio di nascondere le classi concrete al cliente, che comunica solo con la classe prototipo.

Casi d'uso

- Le classi da istanziare sono specificate solamente a tempo d'esecuzione, per cui un codice statico non può occuparsi della creazione dell'oggetto;
- Si vuole evitare di costruire una gerarchia di factory in parallelo a una gerarchia di prodotti, come avviene utilizzando abstract factory e factory method;
- Quando le istanze di una classe possono avere soltanto un limitato numero di stati, per cui può essere più conveniente clonare al bisogno il prototipo corrispondente piuttosto che creare l'oggetto e configurarlo ogni volta.

Diagramma UML



Partecipanti

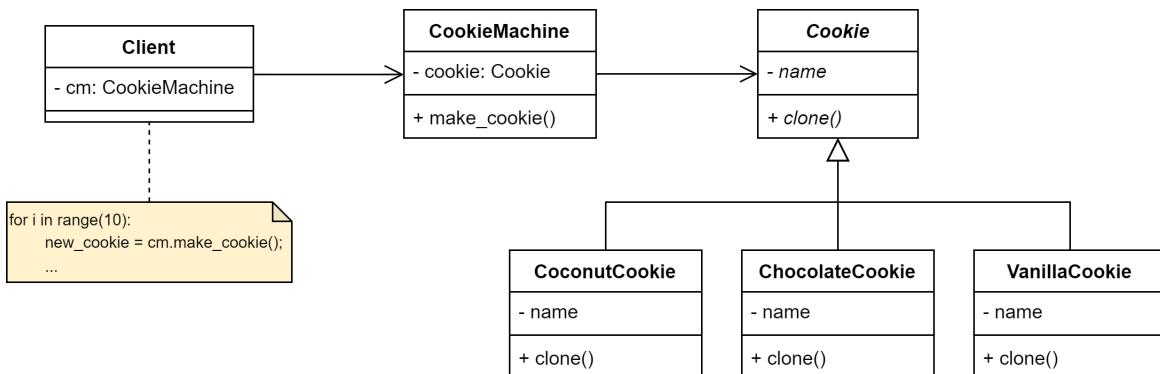
- **Prototype:** definisce un'interfaccia per clonare se stesso;
- **ConcretePrototype:** implementa Prototype fornendo un'operazione per clonarsi;
- **Client:** crea oggetti del tipo desiderato chiedendo a un prototipo concreto di clonarsi qualora necessario.

Esempio

Un esempio di uso del prototype pattern in python.

```
1  class Cookie:    # Classe Prototype
2      def __init__(self, name):
3          self.name = name
4
5      def clone(self):
6          return copy.deepcopy(self)
7
8
9  class CoconutCookie(Cookie):    # Prototipi concreti
10     def __init__(self):
11         Cookie.__init__(self, 'Coconut')
12
13 class ChocolateCookie(Cookie):
14     def __init__(self):
15         Cookie.__init__(self, 'Choco')
16
17 class VanillaCookie(Cookie):
18     def __init__(self):
19         Cookie.__init__(self, 'Vanilla')
20
21
22 class CookieMachine:    # Classe cliente
23     def __init__(self, cookie):
24         self.cookie = cookie
25
26     def make_cookie(self):
27         return self.cookie.clone()
28
29 if __name__ == '__main__':
30     prototype = CoconutCookie()
31     cm = CookieMachine(prototype)
32
33     for i in xrange(10):
34         # Uso del pattern, chiamata a clone()
35         temp_cookie = cm.make_cookie()
```

Il diagramma UML relativo all'esempio:



Builder

Scopo

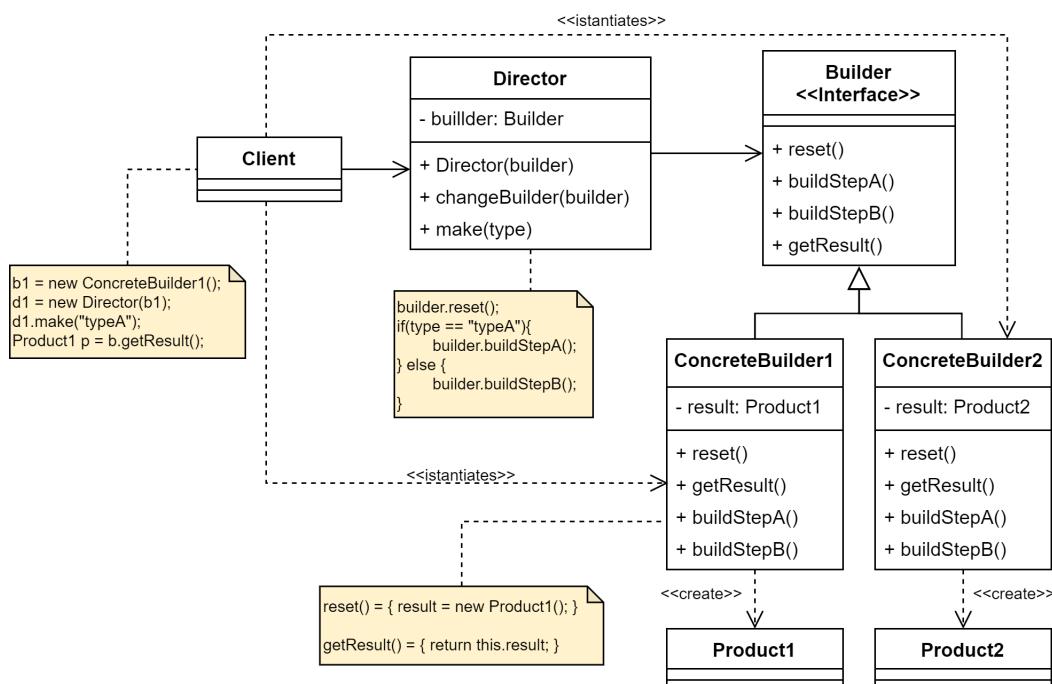
Separare la costruzione di un oggetto complesso dalla sua rappresentazione cosicchè il processo stesso possa creare diverse rappresentazioni. L'algoritmo per la creazione di un oggetto complesso diventa indipendente dalle varie parti che costituiscono l'oggetto e da come vengono assemblate. Ciò ha l'effetto di rendere più semplice la classe, permettendo a una classe builder separata di focalizzarsi sulla costruzione di un'istanza e lasciando che la classe originale si concentri sul funzionamento degli oggetti. Mentre il costruttore di una classe specifica la logica della costruzione di un'istanza, il builder delega il codice per la creazione delle istanze al di fuori della classe da istanziare, non più contenuta nel costruttore.

Il funzionamento è il seguente: il client crea l'oggetto director e lo configura per farlo operare con il builder; il director informa il builder ogni volta che una parte del product deve essere costruita; il builder riceve le richieste dal director e aggiunge le parti al product. Alla fine, il client recupera il product creato. Rispetto ad altri pattern creazionali, nel builder il focus è sul processo di costruzione; product è costruito non in una sola operazione, ma passo dopo passo sotto il controllo del director.

Casi d'uso

- Si vuole separare la logica di costruzione dalla logica di controllo di un oggetto;
- Si vuole permettere la creazione passo passo di un oggetto complesso.

Diagramma UML

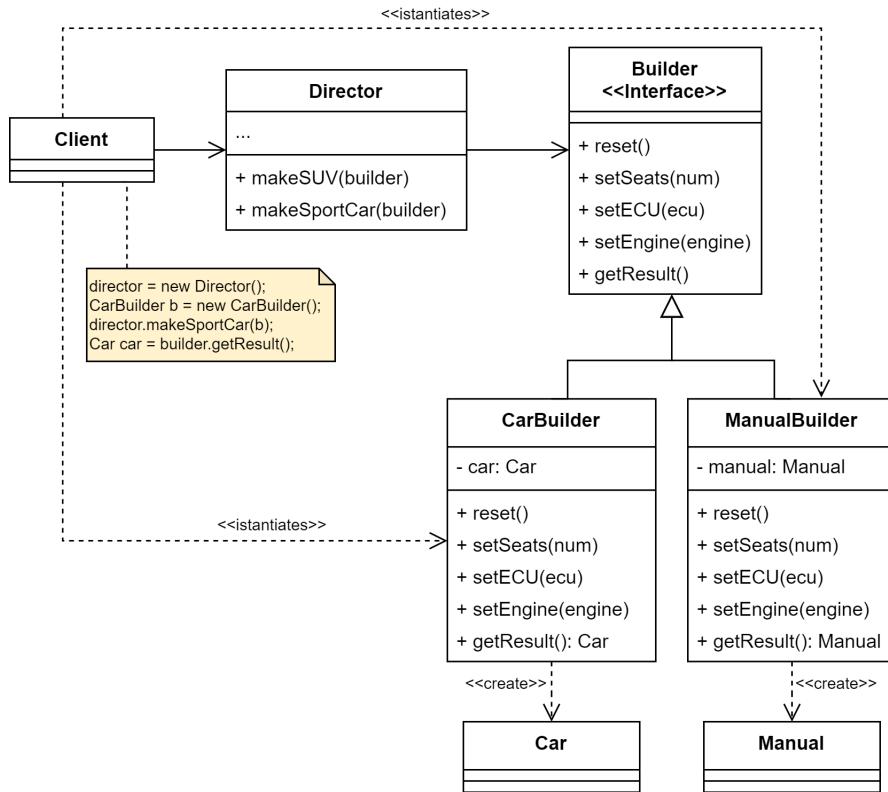


Partecipanti

- **Builder:** specifica la classe astratta che crea le parti dell'oggetto **Product** e i vari passi necessari;
- **ConcreteBuilder:** costruisce e assembla le parti del prodotto implementando l'interfaccia **Builder**; definisce e tiene traccia della rappresentazione che crea;
- **Director:** costruisce un oggetto utilizzando l'interfaccia **Builder** e definisce l'ordine dei passi di costruzione;
- **Product:** rappresenta l'oggetto complesso e include le classi che definiscono le parti che lo compongono, includendo le interfacce per assemblarle in un oggetto finale.

Esempio

Nel seguente esempio di uso del builder viene illustrato un approccio alla creazione di differenti tipologie di auto con relativo manuale.



Un oggetto di tipo **Car** è un oggetto complesso che può essere costruito (si immagini una gerarchia più estesa) in molti modi diversi. Invece di rendere complessa la classe **Car** e il suo costruttore, facciamo uso di un builder. **Client** delega la costruzione passo passo della corretta tipologia di **Car** al **Director**. Per ciascuna auto è necessario anche un manuale (**Manual**); a tal proposito si può riusare il processo di costruzione dell'automobile per creare il corretto manuale per ciascuna tipologia di **Car**.

Segue l'implementazione del diagramma UML precedente:

```
1 public class Car{
2     public Engine engine;
3     public int numSeats;
4     public ECU ecu;
5     ...
6
7     public Car() {...}
8 }
9
10 public class Manual{
11     ...
12
13     public Manual() {...}
14 }
15
16 public interface Builder{
17     public void reset()
18     public void setSeats(...){...}
19     public void setEngine(...){...}
20     public void setECU(...){...}
21 }
22
23
24 public class CarBuilder implements Builder{
25     private Car car;
26
27     public CarBuilder() {
28         this.reset();
29     }
30
31     public void reset(){
32         this.car = new Car();
33     }
34
35     public void setSeats(...){...}
36
37     public void setEngine(...){...}
38
39     public void setECU(...){...}
40
41     public Car getProduct(){
42         product = this.car;
43         this.reset();
44         return product;
45     }
46 }
47
48 public class CarManualBuilder implements Builder{
49     private field manual:Manual
50
51     public CarManualBuilder(){
52         this.reset();
```

```

53     }
54
55     public void reset(){
56         this.manual = new Manual();
57     }
58
59     public void setSeats(...){...}
60
61     public void setEngine(...){...}
62
63     public void setECU(...){...}
64
65     public Manual getProduct(){
66         product = this.manual;
67         this.reset();
68         return product;
69     }
70 }
71
72 public class Director{
73     private Builder builder;
74
75     public void setBuilder(builder:Builder){
76         this.builder = builder;
77     }
78
79     public void constructSportsCar(builder: Builder){
80         builder.reset();
81         builder.setSeats(2);
82         builder.setEngine(new SportEngine());
83         builder.ECU(new SportECU());
84     }
85
86     public void constructSUV(builder: Builder){...}
87 }
88
89
90 public class Application{
91     Director director = new Director();
92
93     CarBuilder builder = new CarBuilder();
94     director.constructSportsCar(builder);
95     Car car = builder.getProduct();
96
97     CarManualBuilder builder = new CarManualBuilder();
98     director.constructSportsCar(builder);
99
100    Manual manual = builder.getProduct();
101 }
```

Template method

Scopo

Si vuole definire/incapsulare la struttura di un algoritmo all'interno di un metodo, lasciandone alcune parti non specificate.

L'implementazione di tali parti non specificate è contenuta in altri metodi, la cui implementazione è delegata alle sottoclassi, che, ridefiniscono solo alcuni passi dell'algoritmo, lasciando invariata la struttura generale.

In questo modo si può ridefinire e personalizzare parte del comportamento nelle varie sottoclassi senza dover riscrivere più volte il codice in comune.

L'idea del template method è simile a quella del factory method, ma sono diversi. In una classe, metodi astratti vengono invocati da un altro metodo e specificati nelle sottoclassi (metodi “gancio/hook” a cui agganciare un'implementazione specifica); ma:

- Nel template (comportamentale) è il metodo non astratto che invoca i metodi astratti;
- Il factory method (creazionale) è un metodo astratto che deve creare e restituire un'istanza.

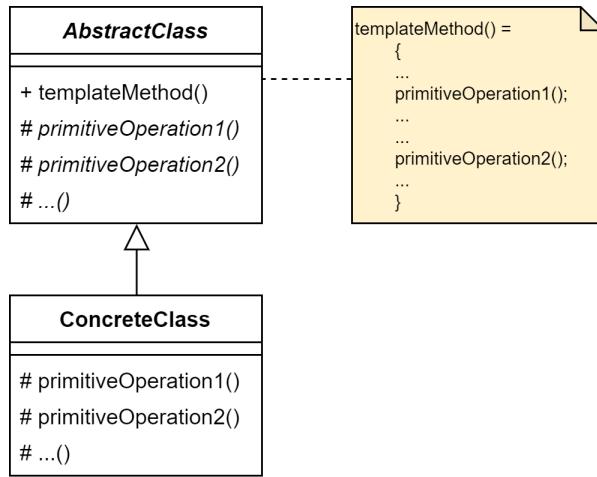
Il pattern in un certo senso ribalta il meccanismo dell'ereditarietà secondo quello che viene scherzosamente chiamato “principio di Hollywood”: non chiamarci, ti chiameremo noi.

Normalmente sono le sottoclassi a chiamare i metodi delle classi genitrici; in questo pattern è il metodo template, appartenente alla classe genitrice, a chiamare i metodi specifici ridefiniti nelle sottoclassi.

Casi d'uso

- Si vuole implementare la parte invariante di un algoritmo una volta sola e lasciare che le sottoclassi implementino il comportamento che può variare;
- Il comportamento comune di più classi può essere fattorizzato in una classe a parte per evitare di scrivere più volte lo stesso codice;
- Poter controllare come le sottoclassi ereditano dalla superclasse, facendo in modo che i metodi template chiamino dei metodi “gancio”; unici metodi sovrascrivibili.

Diagramma UML



Partecipanti

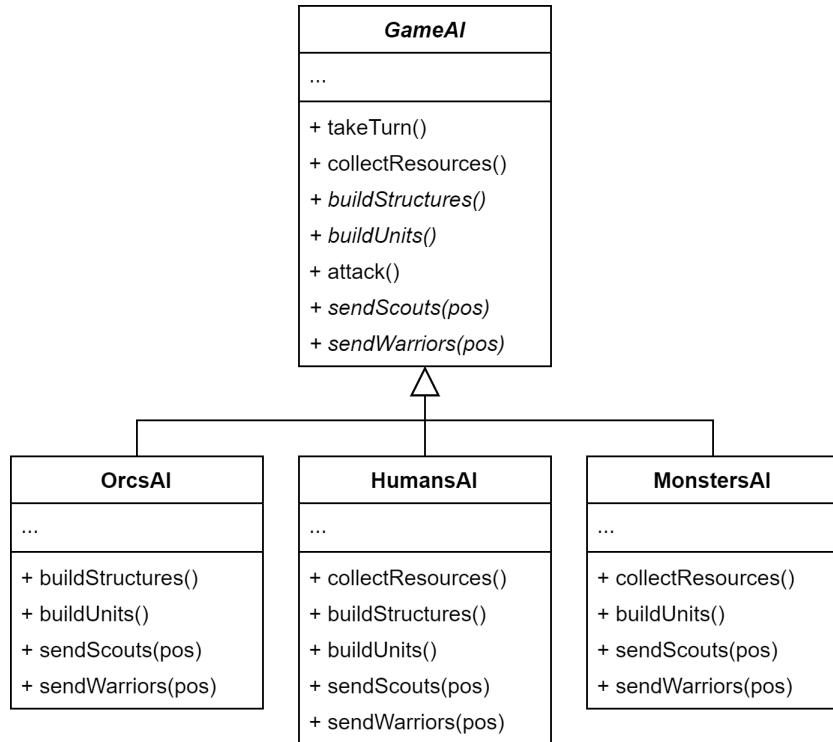
- **AbstractClass:** definisce le operazioni primitive astratte che le classi concrete sottostanti andranno ad ampliare ed implementa il metodo template (`templateMethod()`) che rappresenta la parte invariante (in comune) dell'algoritmo;
- **ConcreteClass:** implementa le operazioni primitive astratte che eredita dalla superclasse, specificando così il comportamento per i passi variabili dell'algoritmo.

Esempio

Nell'esempio sottostante, il template method fornisce uno scheletro per varie tipologie di AI in un videogioco di strategia. Tutte le razze del gioco hanno quasi gli stessi tipi di unità ed edifici. Con l'uso del pattern si può riutilizzare la stessa struttura dell'AI per varie razze, rendendo anche possibile sovrascrivere alcuni dettagli comportamentali.

Con questo approccio, si può sovrascrivere l'AI degli orchi per renderla più aggressiva, rendere gli umani più orientati alla difesa e rendere i mostri incapaci di costruire, etc.

L'aggiunta di una nuova razza al gioco richiederebbe la creazione di una nuova sottoclassa AI e l'override dei metodi predefiniti dichiarati nella classe astratta AI di base (`GameAI`), compito meno oneroso rispetto alla reimplementazione from scratch, che porterebbe anche a replicazione di codice per le parti comuni.



Il metodo `takeTurn()` è il template method, che per ciascuna razza, in ciascun turno compie n azioni, come collezionare risorse, inviare unità, etc. Ciascuna di queste azioni è implementata diversamente per ogni variante di razza/AI. Quindi, `takeTurn()` ha una struttura comune alle varie AI, ma ciascun passo di esso può essere implementato diversamente per ciascuna classe (razza).

Strategy

Scopo

L'obiettivo è di isolare un algoritmo all'interno di un oggetto, in maniera tale da risultare funzionale in quelle situazione in cui sia necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione (client).

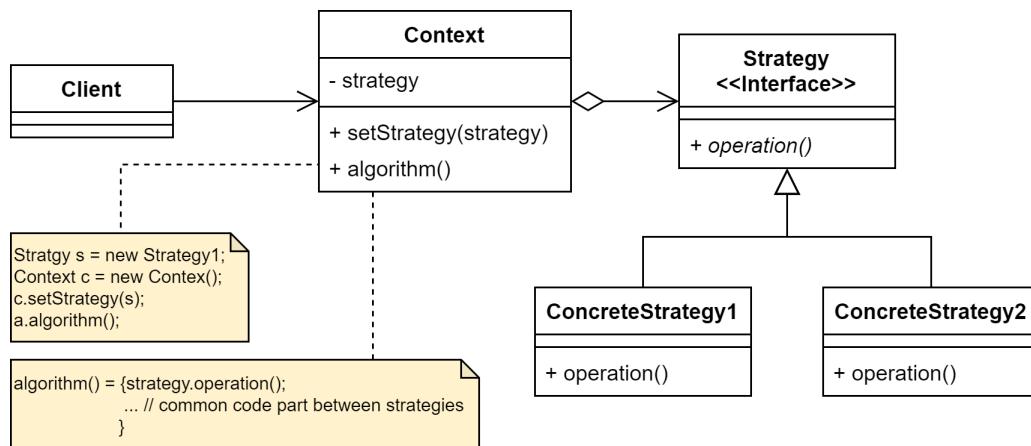
Si pensi ad esempio alle possibili visite in una struttura ad albero (visita anticipata, simmetrica, posticipata, etc.); mediante il pattern strategy è possibile selezionare a run-time una tra le tipologie di visita ed eseguirla sull'albero per ottenere il risultato voluto.

A differenza del template method che fa uso dell'ereditarietà, lo strategy pattern fa uso della composizione (permette variazioni anche al run-time).

Casi d'uso

- Si vuole poter selezionare a run-time un algoritmo (o un comportamento/sotto-algoritmo facente parte di una logica più complessa);
- Si vuole isolare i dettagli implementativi di un algoritmo dal codice (client) che ne fa utilizzo;
- Si vuole strutturare in modo più pulito (usabilità/modificabilità) un insieme di classi molto simili che differiscono solo nel modo in cui eseguono alcuni comportamenti.

Diagramma UML



Partecipanti

- **Context**: mantiene una referenza a una delle strategie concrete e comunica con tale oggetto tramite l'interfaccia **Strategy**; espone un setter per permettere al client di cambiare strategia;

- **Client:** esegue le sue operazioni, specificando una strategia qualora necessario;
- **Strategy:** definisce un’interfaccia comune a tutte le strategie concrete. Dichiara un metodo che il **Context** utilizza per eseguire la strategia;
- **ConcreteStrategy:** implementa diverse versioni dell’interfaccia **Strategy**, implementando di fatto diversi approcci risolutivi (varianti) dell’algoritmo/problema.

Esempio

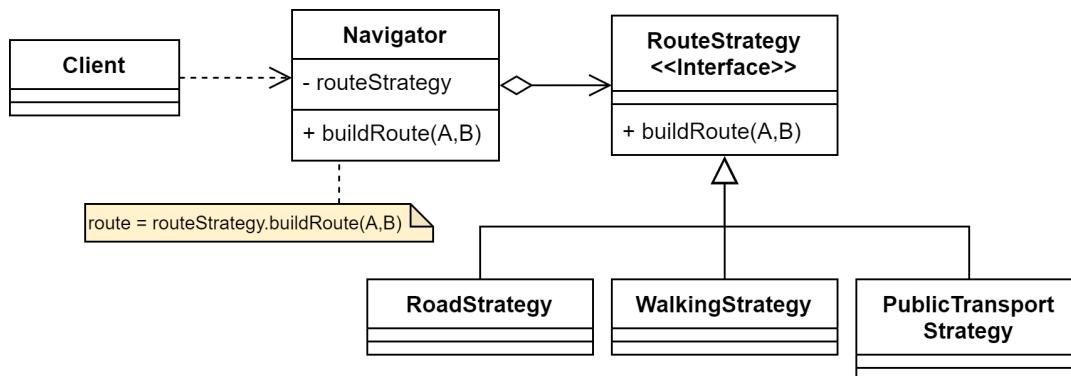
Si pensi di aver scritto un’app mobile per viaggi, focalizzata sul fornire una mappa interattiva che aiuta gli utenti a orientarsi. La feature principale consiste nella possibilità di visualizzare il percorso più veloce tra due punti (“route-planning”).

Nella prima versione dell’app era possibile visualizzare solo percorsi in auto; nelle versioni successive la funzionalità è stata estesa per le bici, poi per i pedoni, poi per i trasporti pubblici, e così via.

Dal punto di vista aziendale l’app è un successo, la parte tecnica però ha accumulato del debito tecnico; presenta dei problemi: ogni estensione dell’algoritmo di routing ha reso il codice sempre più di scarsa qualità e ha esteso la classe principale del navigatore, fino a renderla ingestibile e troppo complessa.

Una possibile soluzione è l’uso del strategy pattern: prendere una classe che svolge qualcosa di specifico in molti modi diversi (la classe **Navigator**) ed estrarre tutti gli algoritmi (le varianti di routing) in classi separate chiamate strategie. La classe originale (denominata contesto) deve avere un campo per memorizzare il riferimento a una delle strategie. Il contesto delega il lavoro all’oggetto strategia selezionato.

Non è il contesto a scegliere la strategia specifica da utilizzare, ma il cliente. Nel nostro esempio, avremmo:



Nella nostra app di navigazione, ora ogni algoritmo di routing può essere estratto nella propria classe con un unico metodo **buildRoute()**. Il metodo accetta un punto di partenza A e una destinazione B e restituisce una lista di checkpoint del percorso.

Ogni classe di routing potrebbe costruire una rotta diversa. Alla classe principale del navigatore non interessa quale algoritmo sia selezionato poiché il suo compito principale è quello di eseguire il rendering di un insieme di checkpoint sulla mappa. La classe ha un metodo per cambiare la strategia di routing attiva.

State

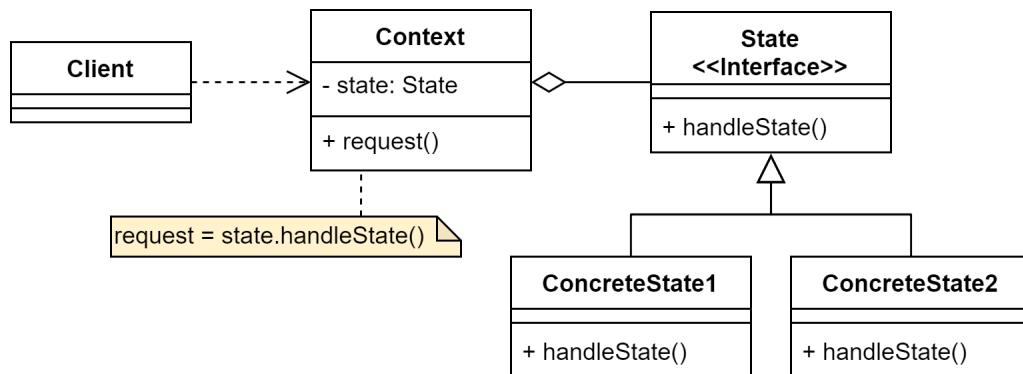
Scopo

Permettere ad un oggetto di cambiare il proprio comportamento a run-time in funzione dello specifico stato in cui si trova.

Casi d'uso

- Un oggetto deve cambiare il suo comportamento in relazione al suo stato interno;
- I comportamenti relativi a ciascun stato devono poter essere definiti in modo indipendente;
- Una classe è inquinata da molti controlli condizionali che alterano il comportamento della classe in relazione ai valori dei parametri a run-time;
- Aggiungere un nuovo stato non deve influenzare il comportamento degli stati presenti.

Diagramma UML

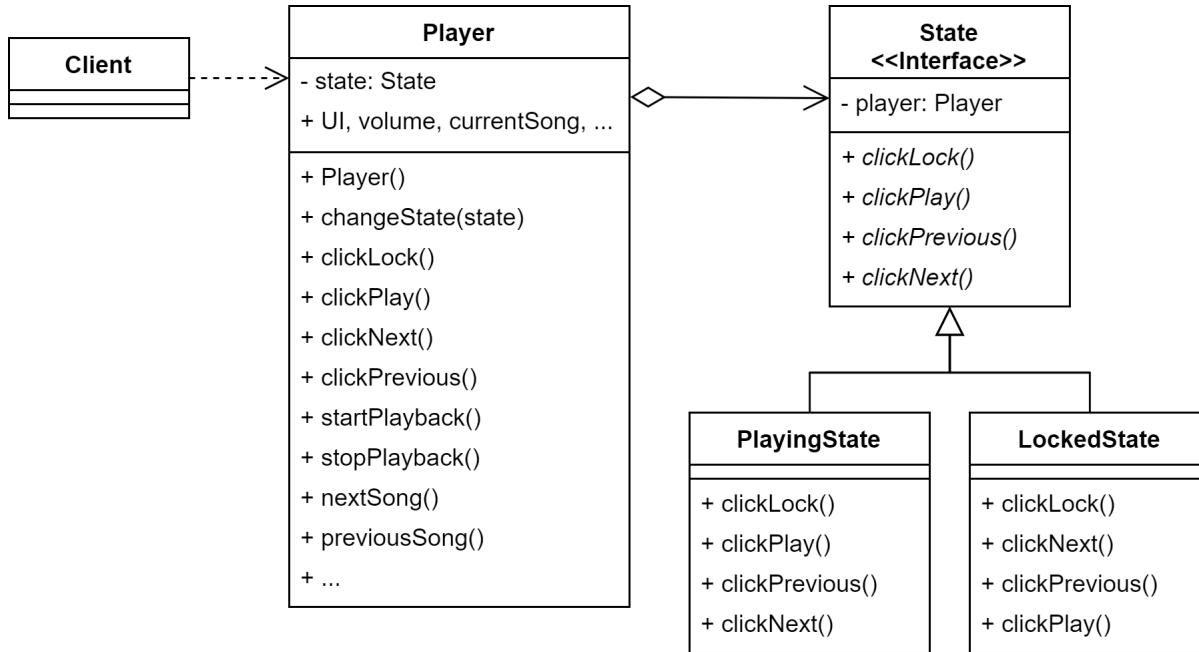


Partecipanti

- **Context:** definisce la classe utilizzata dal client e mantiene un riferimento ad un **ConcreteState**;
- **State:** definisce l'interfaccia, implementata dai **ConcreteState**, che incapsula la logica del comportamento associato ad un determinato stato;
- **ConcreteState:** implementa il comportamento associato ad un particolare stato.

Esempio

Si pensi di voler controllare un media player, che si comporta (per alcune azioni) diversamente in base allo stato corrente. Un esempio di applicazione del pattern potrebbe essere il seguente:



L'istanza di **Player** è sempre collegata a un'istanza di **State** che esegue la maggior parte del lavoro. Alcune azioni alterano lo stato del player con un altro, il che cambia il modo in cui il player reagirà ad alcune interazioni dell'utente.

Command

Scopo

Permettere di isolare la porzione di codice che effettua un’azione (eventualmente molto complessa) dal codice che ne richiede l’esecuzione.

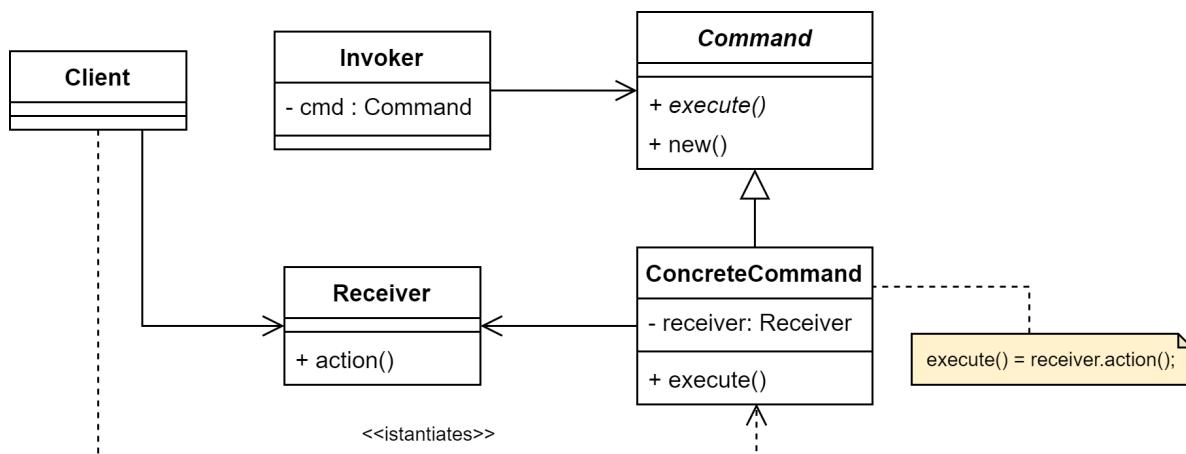
Il codice relativo alla richiesta di un servizio è incapsulato all’interno di un oggetto command. Si mira a rendere variabile l’azione del client senza però conoscere i dettagli dell’operazione stessa. L’azione può non essere decisa staticamente ma bensì ricavata/scelta a run-time.

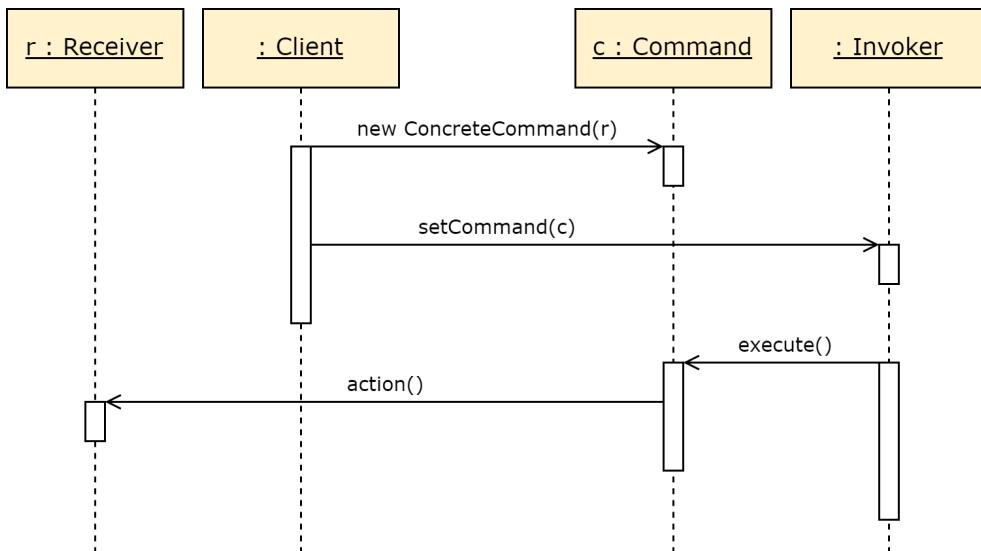
Di solito, per effettuare una richiesta di servizio (chiamata a un metodo) bisogna conoscerne il nome e il destinatario. Con il command pattern si incapsulano tali informazioni in un oggetto, riducendo le responsabilità del chiamante.

Casi d’uso

- Si vuole maggiore libertà nella scelta a run-time di una specifica operazione/azione da compiere in base al contesto;
- Si vuole incapsulare il codice al fine di nascondere l’implementazione di alcune azioni complesse;
- Le azioni diventano “gestibili”; non essendo righe di codice ma oggetti. Si ottiene maggiore libertà, esse possono essere memorizzate, passate come argomento, etc.

Diagramma UML





Partecipanti

- **Invoker:** memorizza un riferimento astratto al comando (istanza di `ConcreteCommand`, sottoclasse di `Command`) e ne invoca l'`execute()`. L'invoker è colui che ha bisogno di un determinato servizio che è raggiungibile tramite un oggetto `Command`; servizio di fatto parte del codice del `Receiver`;
- **Command:** definisce un'interfaccia per i singoli comandi “resi oggetti”;
- **Receiver:** riceve l'invocazione di `action()` dal `ConcreteCommand`; è la classe/oggetto che contiene business logic, i servizi effettivi (resi comandi) richiesti dall'invoker;
- **ConcreteCommand:** implementa l'interfaccia di `Command`, comune a tutti i comandi/richieste; invoca `action` (l'effettivo servizio richiesto dall'invoker) di `Receiver` quando invocato dall'`Invoker`;
- **Client:** crea istanze di `ConcreteCommand` e le passa al `Receiver` per utilizzarle; svolge la logica di business tramite i singoli comandi.

Esempio

Analizziamo il codice del client con e senza command pattern:

Senza l'uso del command pattern avremmo:

```
1 Receiver r = new Receiver();
2 r.action();
```

Con l'uso del command pattern avremo, nel client:

```
1 Receiver r = new Receiver();
2 Command c = new ConcreteCommand(r);
```

Sempre nel client (o chiunque notifichi all'invoker):

```
1 Invoker i = new Invoker();
2 i.setCommand(c);
```

Nell'Invoker:

```
1 c.execute();
```

Il command permette di disaccoppiare il client (che crea il comando/richiesta) e receiver (che esegue) tramite l'invoker (che invoca la richiesta al momento opportuno).

I comandi sono oggetti e non invocazioni, è possibile quindi comporre più comandi in un comando composto (ad esempio tramite il composite pattern). È più semplice aggiungere nuovi comandi, è possibile memorizzare uno storico dei comandi per l'esecuzione dell'undo, etc.

Observer

Scopo

Permettere di osservare lo stato di un oggetto tramite degli oggetti che vengono notificati ed aggiornati ad ogni variazione.

Un osservatore, in realtà, non “osserva” ma rimane in attesa e aspetta che gli venga notificato di controllare (“guardare”) l’oggetto di riferimento. Viene tipicamente utilizzato nel paradigma di programmazione ad eventi.

Il funzionamento è il seguente: gli osservatori si “registrano/abbonano” presso l’oggetto osservato che li notificherà ogni volta cambierà stato, tramite l’invocazione di un metodo (“meccanismo di callback”).

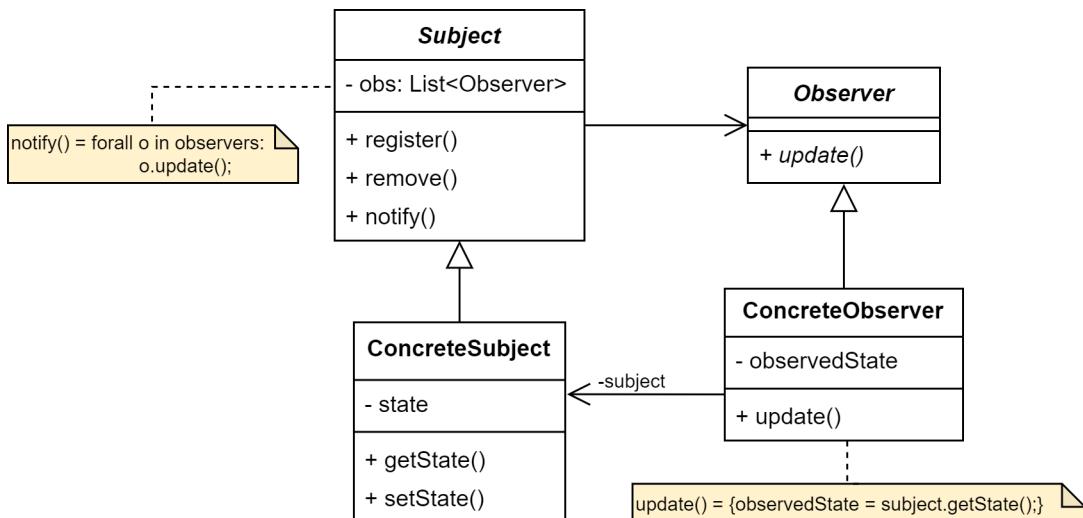
Quando un osservatore viene notificato, decide cosa fare (niente; richiedere all’osservato informazioni sul nuovo stato, etc.).

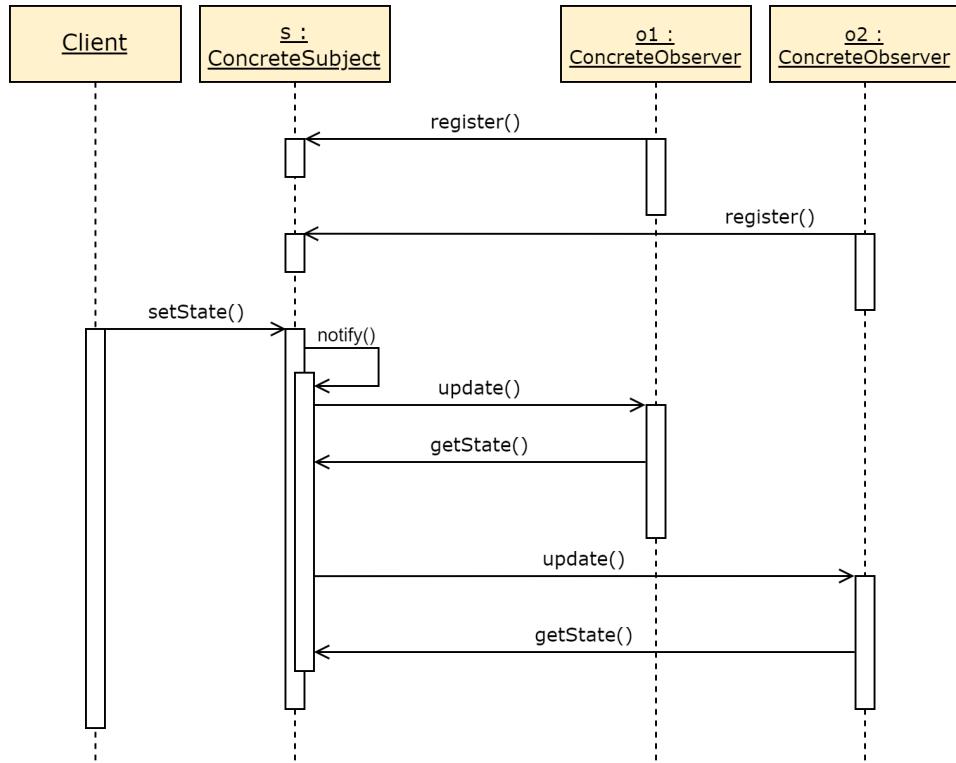
Gli osservatori possono aggiungersi/rimuoversi dalla lista degli “abbonati” a run-time. L’oggetto osservato non conosce il tipo concreto degli osservatori ma gli basta conoscere una superclasse/interfaccia da essi condivisa.

Casi d’uso

- Si vuole monitorare lo stato di un oggetto e reagire ai cambiamenti;
- Si vuole definire una dipendenza uno-a-molti senza rendere gli oggetti strettamente accoppiati;
- Si vuole aggiornare in modo automatico una serie di oggetti al cambiamento di stato di un’altra oggetto;

Diagramma UML





Partecipanti

- **Subject:** fornisce un’interfaccia per registrare, rimuovere o notificare gli observer;
- **ConcreteSubject:** fornisce lo stato dei soggetti concreti agli observer e si occupa di notificare gli observer registrati invocando la funzione `update()`;
- **Observer:** definisce un’interfaccia per tutti gli observers, per ricevere le notifiche dal soggetto osservato. È utilizzata come classe astratta per implementare i veri **Observer**, ossia i **ConcreteObserver**;
- **ConcreteObserver:** mantiene un riferimento al soggetto concreto (osservato) per riceverne lo stato quando notificato. Il **ConcreteObserver** implementa la funzione astratta `update()`: quando viene chiamata dal soggetto concreto, il **ConcreteObserver** chiama la funzione `getState()` sul soggetto concreto per aggiornare il suo nuovo stato.

Esempio

Un esempio di implementazione generica in python:

```
1 class Observable:
2     def __init__(self):
3         self._observers = []
4
5     def register_observer(self, observer):
6         self._observers.append(observer)
7
8     def notify_observers(self, *args, **kwargs):
9         for obs in self._observers:
10             obs.notify(self, *args, **kwargs)
11
12
13 class Observer:
14     def __init__(self, observable):
15         observable.register_observer(self)
16
17     def notify(self, observable, *args, **kwargs):
18         print("Received", args, kwargs, "From", observable)
19
20
21 subject = Observable()
22 observer = Observer(subject)
23 subject.notify_observers("test", kw="python")
24
25 # -- Output:
26 # prints: Got ('test',) {'kw': 'python'} From <__main__.Observable object
```

Mediator

Scopo

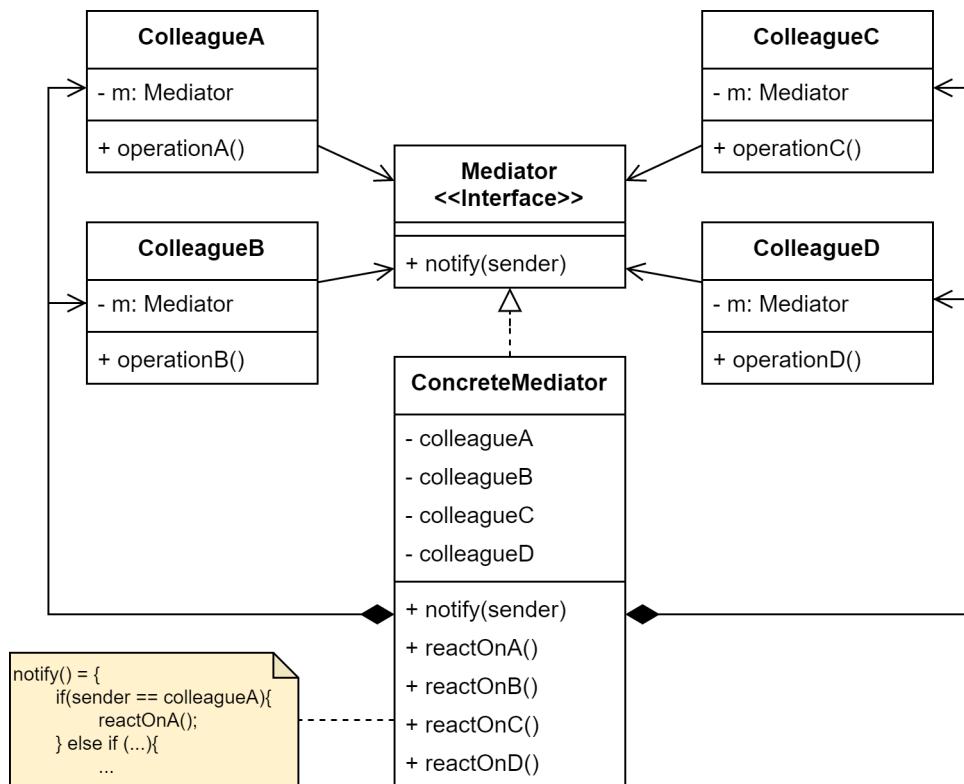
Lo scopo del mediator è di permettere di modificare agilmente le politiche di interazioni tra oggetti, poiché le entità coinvolte devono far riferimento a un solo oggetto (il mediatore).

Senza mediator, tante classi hanno alto accoppiamento, in quanto ciascuna di esse deve conoscere i metodi di molte altre classi. Cala la comprensibilità e il riuso del codice. Con il mediator si abbassa l'accoppiamento e si facilitano i cambiamenti nella logica di interazione (in quanto localizzata in una singola classe). Le classi singole si semplificano e migliora il riuso e la chiarezza del codice.

Casi d'uso

- Si vuole maggior controllo e un punto specifico dove specificare le modalità di interazioni tra oggetti molteplici;
- Si vuole nascondere ai singoli oggetti i “fratelli” con cui interagiscono.

Diagramma UML



Si noti come i singoli Colleague potrebbero essere riorganizzati gerarchicamente in una classe astratta Colleague.

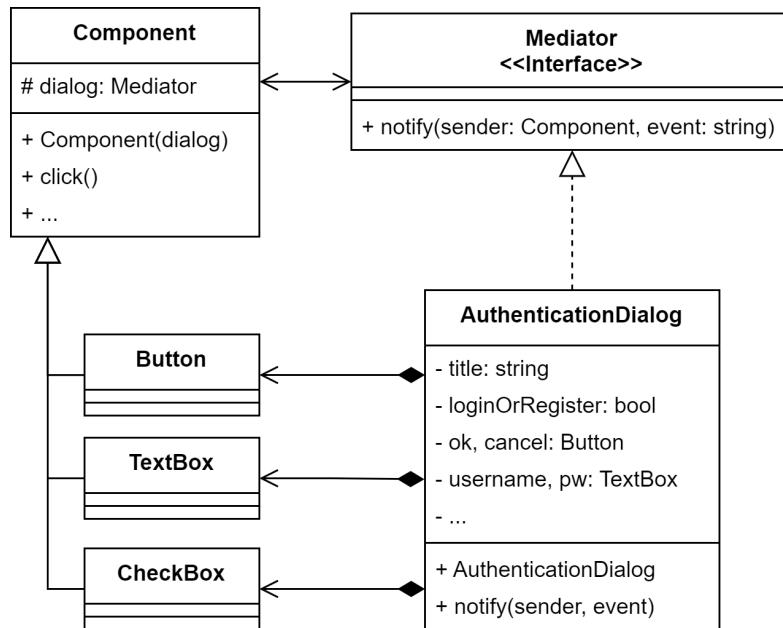
Partecipanti

- **Mediator:** funge da intermediario fra i vari “colleagues”, oggetti dei quali mantiene un riferimento interno. Riceve gli eventi (tramite calls a `notify()`) da essi e modifica lo stato del sistema di conseguenza;
- **Colleague:** gli oggetti che interagiscono fra loro mediante il mediator. I vari colleagues possono anche essere sottoclassi di una classe astratta comune. Contengono business logic; essi non sono a conoscenza della classe concreta del Mediator.

Esempio

In questo esempio vediamo come è possibile eliminare le mutue dipendenze tra singoli elementi di una API grafica: UI, buttoni, checkboxes, text labels, etc.

L'applicazione del pattern porterebbe alla seguente gerarchia:



Un elemento attivato/triggerato dall'utente, non comunica con gli altri elementi direttamente, anche se a livello logico dovrebbe. Piuttosto, fa sapere al **Mediator** dell'evento, fornendo informazioni contestuali.

Il mediator (**AuthenticationDialog**) conosce come gli elementi concreti debbano collaborare e facilita la loro comunicazione e cooperazione. Alla ricezione di una notifica, il mediator esegue le operazioni relative e aziona/altera gli altri elementi in modo appropriato.

La logica di interazione è quindi completamente delegata al mediator.

Si pensi che il **Button** a seguito di un evento debba modificare tutti gli altri componenti. Con l'uso del pattern deve notificare solo un oggetto, che si occuperà di gestire l'evento. Senza l'uso del pattern, **Button** dovrebbe avere un riferimento a tutti gli altri componenti e gestire l'evento internamente; ottenendo così una soluzione meno adatta al riuso, meno manutenibile (logica di interazione sparsa su più classi).

Memento

Scopo

Lo scopo è quello di catturare/estrarre lo stato interno di un oggetto, e di memorizzarlo, in modo da poterlo ripristinare in seguito, senza violare l'incapsulamento.

Tipico esempio è l'operazione di undo, che consente di ripristinare lo stato di uno o più oggetti a come era/erano prima dell'esecuzione di una data operazione.

Il punto chiave di questo pattern è la definizione di un oggetto di tipo memento nel quale verrà immagazzinato lo stato di un oggetto, l'originator. Tale oggetto memento disporrà di una doppia interfaccia:

- Quella verso l'originator, più ampia, che consentirà a questo di salvare il suo stato interno e di ripristinarlo;
- Quella verso gli altri, che esporrà solamente l'eventuale distruttore.

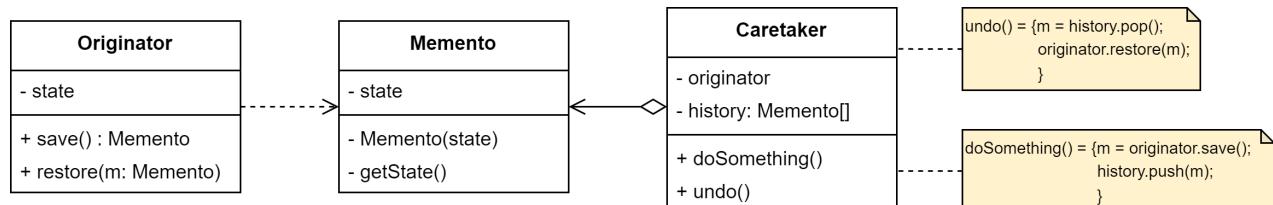
Solo l'originator conoscerà quindi la reale interfaccia del memento, e solo esso sarà in grado di istanziarlo.

Casi d'uso

- Si vuole salvare e ripristinare uno snapshot dello stato interno di un oggetto;

Diagramma UML

La classica implementazione del memento si basa sulle classi nidificate (nested-class):



In questa implementazione, la classe **Memento** è nidificata all'interno dell'originator. Ciò consente alla classe **Originator** di accedere ai campi e ai metodi del memento, anche se privati. Il **Caretaker** ha un accesso limitato ai campi e metodi del memento; questo gli consente di archiviare i singoli memento (stati/snapshots/ricordi) senza poterli alterare in alcun modo.

Implementazioni alternative si basano su un'interfaccia intermedia: in assenza di classi nidificate, si può limitare l'accesso ai campi del memento stabilendo una convenzione secondo cui i caretakers possono lavorare con un ricordo solo attraverso un'interfaccia intermedia dichiarata esplicitamente, che dichiarerebbe solo metodi relativi ai campi del memento.

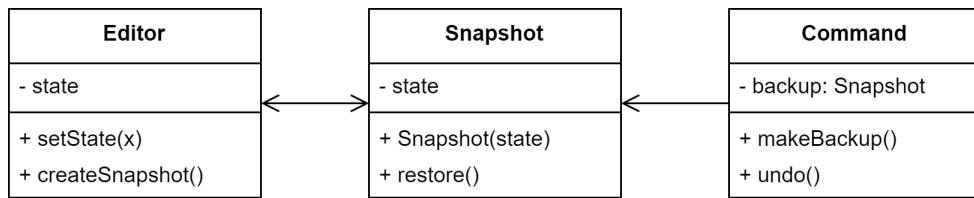
D'altra parte, gli originator possono lavorare direttamente con un oggetto memento (concreto, che implementa l'interfaccia), accedendo ai campi e ai metodi dichiarati nell'interfaccia **Memento**. Lo svantaggio di questo approccio è che è necessario dichiarare pubblici tutti i membri del memento.

Partecipanti

- **Originator:** produce uno snapshot del suo stesso stato e ripristina il suo stato a partire da uno snapshot; il client fa uso dell'originator;
- **Memento:** è di fatto lo snapshot prodotto dall'originator. Tipicamente è un oggetto immutabile dopo la creazione;
- **Caretaker:** conosce quando e perché catturare lo stato dell'originator e quando questo deve essere ripristinato. Il caretaker può tenere traccia della storia dell'originator tramite (ad esempio) uno stack.

Esempio

Il seguente esempio di utilizzo del pattern è accoppiato al command pattern. Si ipotizzi di avere un text-editor e di volerne salvare e poter successivamente ripristinare lo stato.



Gli oggetti **Command** fungono da caretakers. Essi prelevano il memento dall'editor prima di eseguire operazioni relative ai comandi. Quando il cliente chiede di annullare il comando più recente, l'editor può utilizzare il memento memorizzato in quel comando per tornare allo stato precedente.

La classe **Memento** non dichiara alcun campo pubblico, né tantomeno getter o setter. Di conseguenza nessun oggetto può alterarne il contenuto. I memento sono collegati all'oggetto editor che li ha creati, consentendo a un memento di ripristinare lo stato dell'editor collegato passando i dati tramite un setter sull'oggetto **Editor**.

Segue lo pseudocodice dell'esempio:

```
1 class Editor is
2     private field text, curX, curY, selectionWidth
3
4     method setText(text) is
5         this.text = text
6
7     method setCursor(x, y) is
8         this.curX = x
9         this.curY = y
10
11    method setSelectionWidth(width) is
12        this.selectionWidth = width
13
14    method createSnapshot():Snapshot is
15        return new Snapshot(this, text, curX, curY, selectionWidth)
```

```
1 class Snapshot is
2     private field editor: Editor
3     private field text, curX, curY, selectionWidth
4
5     constructor Snapshot(editor, text, curX, curY, selectionWidth) is
6         this.editor = editor
7         this.text = text
8         this.curX = x
9         this.curY = y
10        this.selectionWidth = selectionWidth
11
12    method restore() is
13        editor.setText(text)
14        editor.setCursor(curX, curY)
15        editor.setSelectionWidth(selectionWidth)
16
17
18 class Command is
19     private field backup: Snapshot
20
21    method makeBackup() is
22        backup = editor.createSnapshot()
23
24    method undo() is
25        if (backup != null)
26            backup.restore()
27        // ...
```

Iterator

Scopo

Fornire un metodo di accesso sequenziale agli elementi di un oggetto composto senza esporre la struttura di quest'ultimo, come ad esempio scorrere una lista o visitare un albero.

Senza iterator si necessita di mantenere la responsabilità dell'accesso nella classe dell'oggetto composto, con la conseguenza che si potrebbe aver bisogno di molteplici traversamenti. Questo causerebbe un sovraccarico dell'interfaccia dell'oggetto composto.

Con l'iterator la responsabilità di accesso e attraversamento viene separata in una classe separata, detta iteratore, che ha la responsabilità di:

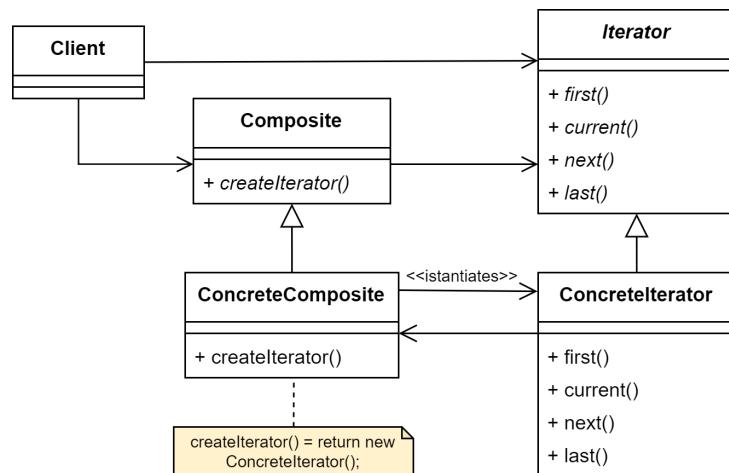
- Rappresentare un metodo/algoritmo di visita;
- Restituire l'elemento "successivo" (secondo la visita);
- Mantenere traccia dell'elemento corrente e degli elementi visitati.

A questo punto alla classe dell'oggetto composto rimane solo la responsabilità di creare l'iteratore opportuno.

Casi d'uso

- Si vuole evitare di sovraccaricare l'interfaccia della classe di un oggetto "visitabile/attraversabile";
- Si vuole evitare di centralizzare le operazioni di visita/accesso nella classe principale (il che non consente di effettuare contemporaneamente più visite/accessi indipendenti);
- Si vuole più controllo e indipendenza tra vari metodi di visita alternativi di un oggetto molto complesso; senza sovraccaricare la classe principale.

Diagramma UML



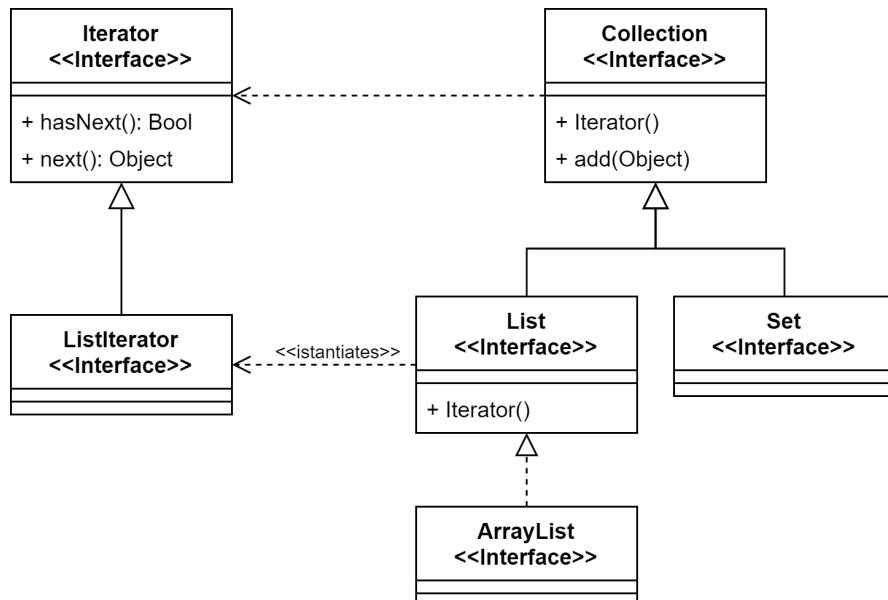
Partecipanti

- **Iterator:** definisce un'interfaccia per attraversare l'insieme degli elementi di un composite/oggetto e accedere ai singoli elementi;
- **ConcreteIterator:** implementa l'interfaccia **Iterator** tenendo traccia della posizione corrente nel composite e calcolando quale sia l'elemento successivo nell'attraversamento corrente;
- **Composite:** definisce un'interfaccia per creare un oggetto **Iterator**;
- **ConcreteComposite:** implementa l'interfaccia di creazione dell'**Iterator** e ritorna un'istanza appropriata di **ConcreteIterator**.

Esempio

Un esempio di uso del pattern può essere trovato in linguaggi di programmazione come Java; come ad esempio le interfacce **Collection**, **List**, **Set**. Il linguaggio espone l'interfaccia **java.util.Iterator**. Ad esempio, **iterator()** di **Collection** è un factory method che restituisce l'iteratore opportuno.

Il diagramma UML di **java.util.Iterator**:



```
1 List list = Arrays.asList(new String[] {"have", "a", "nice", "day", ":"});
2 Iterator i = list.iterator();
3
4 while (i.hasNext()){
5     System.out.println(i.next());
6 }
```

Visitor

Scopo

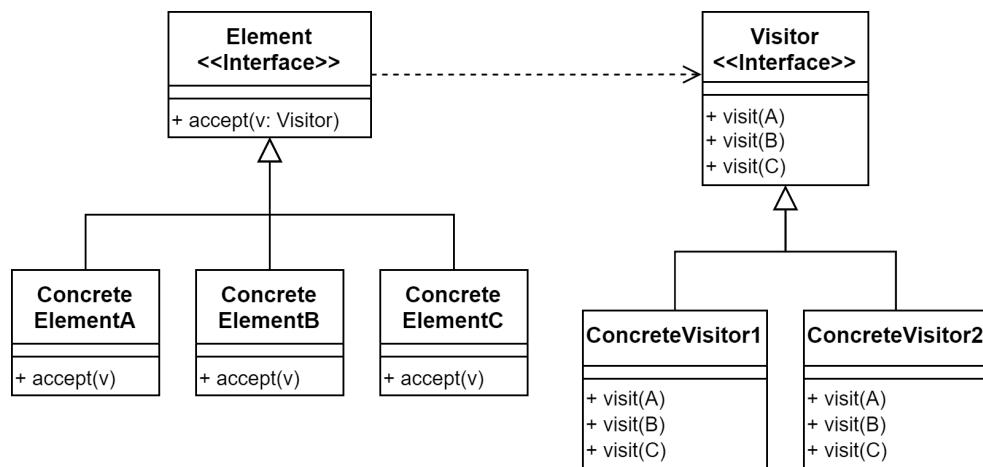
Permette di separare un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter rendere possibile l'aggiunta di nuove operazioni e comportamenti senza dover modificare la struttura stessa dell'oggetto su cui opera.

Ciò significa che il visitor permette di aggiungere nuove funzioni “virtuali” a una famiglia di classi, senza necessità di modificare le classi stesse.

Casi d'uso

- Una gerarchia di oggetti è costituita da molte classi con interfacce diverse ed è necessario che l'algoritmo esegua su ogni oggetto un'operazione differente a seconda della classe concreta dell'oggetto stesso;
- È necessario eseguire molteplici operazioni indipendenti sugli oggetti di una gerarchia composta, ma non si vuole sovraccaricare le interfacce delle loro classi. Riunendo le operazioni correlate in ogni visitor è possibile inserirle nei programmi solo dove necessario;

Diagramma UML



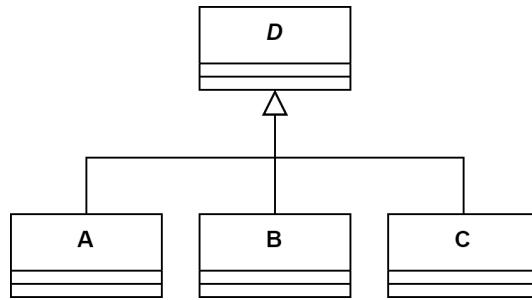
Partecipanti

- **Visitor:** definisce un'interfaccia e dichiara un metodo `visit()` per ciascun elemento concreto appartenente alla gerarchia di oggetti in modo tale che ciascuno di essi possa invocare il metodo appropriato passando un riferimento a sé (`this`) come parametro;
- **ConcreteVisitor:** implementa l'operazione `visit()` perché agisca come desiderato per la rispettiva classe;

- **Element:** fornisce un’interfaccia che dichiara l’operazione `accept()` utilizzata per accettare un `Visitor` passato come parametro;
- **ConcreteElement:** implementa `accept()` per la rispettiva classe concreta.

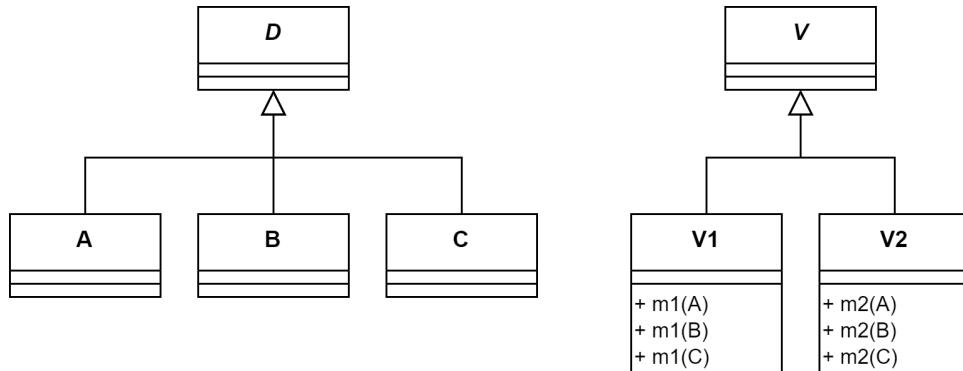
Esempio

Si pensi di avere la seguente gerarchia. È comodo aggiungere nuove classi, basta estendere la classe D. Aggiungere operazioni a tutte le classi è più complesso e richiede la modifica di tutte le sottoclassi. Il visitor propone un’approccio alternativo.



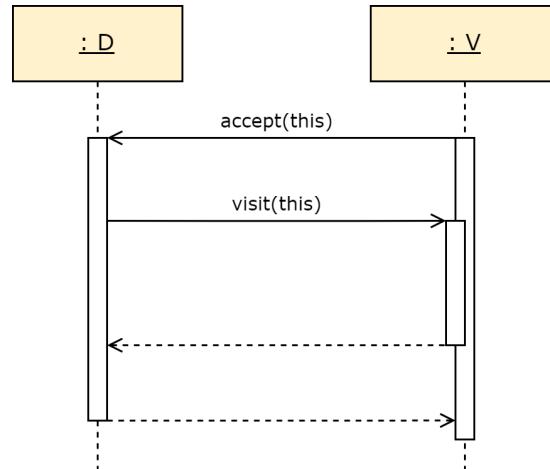
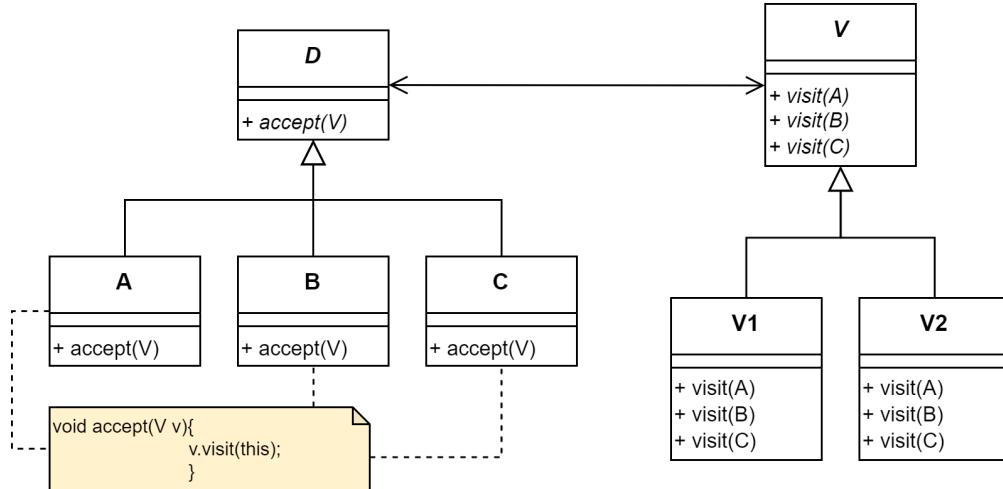
Si crea una sottoclasse di V per ciascuna operazione.

Ogni sottoclasse contiene l’implementazione dell’operazione per tutte le sottoclassi della gerarchia indotta da D. Tali funzioni vengono sovraccaricate grazie al diverso tipo di oggetto passato come argomento (un metodo per ciascuna sottoclasse di D).



Il visitor si basa sulla metafora del visitatore: il visitatore chiede di essere ospitato (invocando `accept()`), mentre il “padrone di casa” accetta (invocando il metodo `visit()`).

Applicando il pattern avremmo il seguente diagramma UML:



Il pattern fa uso di una tecnica chiamata “double-dispatching”, in italiano “doppio intiro”, ovvero:

- V invoca `accept()` di D (in realtà il metodo sovraccaricato della sottoclasse);
- D esegue `accept()`, cioè invoca `visit()` di V (in realtà il metodo sovraccaricato della sottoclasse).

Chain of Responsibility

Scopo

Lo scopo è di evitare l'accoppiamento fra mittente di una richiesta e il destinatario. Il pattern presenta più oggetti concatenati (handler) che possono soddisfare la richiesta.

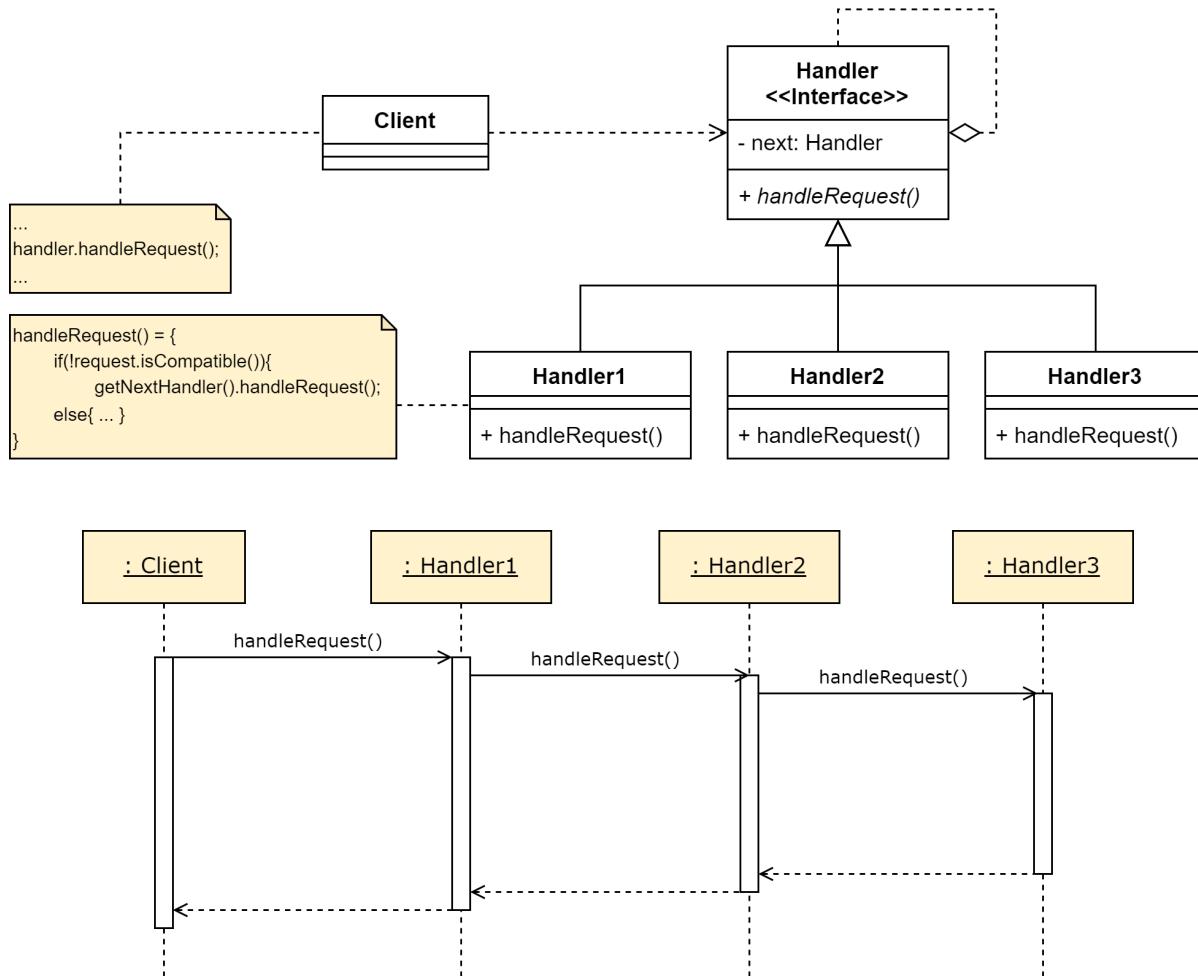
Ciascun oggetto contiene la logica che definisce le tipologie di comandi che può gestire; i comandi non gestiti sono passati all'oggetto successivo facente parte della catena.

Se la richiesta arriva a fine catena, non è gestita. La catena è modificabile a run-time.

Casi d'uso

- Si vuole evitare l'accoppiamento tra sender e receiver di una richiesta;
- Si vuole permettere l'esistenza di più di un receiver capace di gestire una richiesta.

Diagramma UML



Partecipanti

- **Handler:** rappresenta la classe astratta/interfaccia che offre il metodo `handleRequest()` che verrà utilizzato dal codice del client (mittente) per inoltrare le richieste;
- **ConcreteHandler:** rappresenta l'effettiva implementazione della gestione dei comandi/richieste per un oggetto.

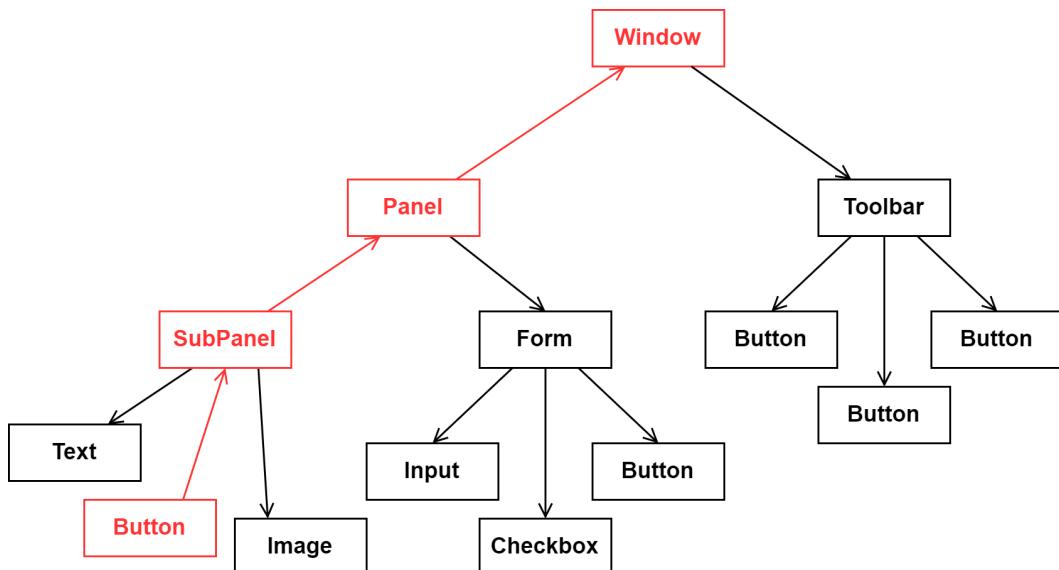
Esempio

L'uso del pattern è molto comune nella gestione di eventi in uno stack di componenti grafiche per interfacce utenti.

Per esempio, quando un utente clicca un pulsante, l'evento viene propagato attraverso la catena di responsabilità di elementi della gerarchia della GUI, iniziando in questo caso dalla classe `Button`.

Tale richiesta (evento) viene propagata attraverso i suoi container (ad esempio dei form o dei panel), fino ad arrivare “in cima” alla finestra principale dell'applicazione.

L'evento è processato dal primo elemento della catena capace di gestirlo.



È cruciale che tutte le classi `Handler` implementino la stessa interfaccia.

Interpreter

Scopo

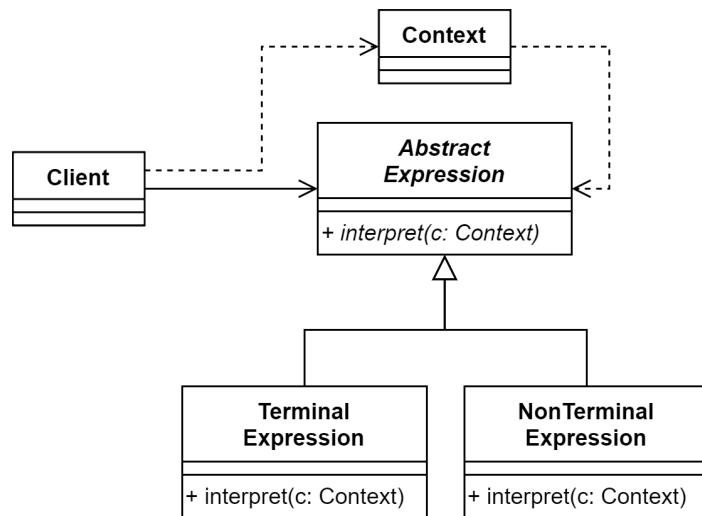
Definire una rappresentazione della grammatica di un linguaggio insieme ad un componente interprete che utilizza tale rappresentazione per l'interpretazione delle espressioni la cui sintassi rispetta il linguaggio.

L'interprete specifica come valutare sentenze in un linguaggio. L'idea base è quella di avere una classe per ciascun simbolo (terminale o non terminale). L'albero di sintassi di una sentenza nel linguaggio trattato è un'istanza di un composite pattern ed è usata per valutare (interpretare) la sentenza per il client.

Casi d'uso

- Implementazione di linguaggi d'interrogazione di database;
- Implementazione di linguaggi informatici specializzati; ad esempio per descrivere protocolli di comunicazione;
- Implementazione di linguaggi di programmazione.

Diagramma UML



Partecipanti

- **AbstractExpression:** definisce un'interfaccia per interpretare un'espressione tramite il metodo `interpret(context)`;
- **TerminalExpression:** non ha figli ed interpreta una espressione direttamente;
- **NonTerminalExpression:** contiene una lista di espressioni figlie a cui inoltra le sotto-richieste di interpretazione.

Esempio

Si pensi alla seguente grammatica per espressioni aritmetiche in Backus-Normal form (BNF):

```
1 expression ::= plus | minus | variable | number
2 plus ::= expression expression '+'
3 minus ::= expression expression '-'
4 variable ::= 'a' | 'b' | 'c' | ... | 'z'
5 digit = '0' | '1' | ... | '9'
6 number ::= digit | digit number
```

Produce espressioni in notazione polacca inversa, come:

```
a b +
a b c + -
a b + c a - -
```

Tale grammatica può essere implementata in C# con l'uso dell'interpreter-pattern nel seguente modo:

```
1 class Program
2 {
3     static void Main()
4     {
5         var context = new Context();
6         var input = new MyExpression();
7
8         var expression = new OrExpression
9         {
10             Left = new EqualsExpression
11             {
12                 Left = input,
13                 Right = new MyExpression { Value = "4" }
14             },
15             Right = new EqualsExpression
16             {
17                 Left = input,
18                 Right = new MyExpression { Value = "four" }
19             }
20         };
21
22         input.Value = "four";
23         expression.Interpret(context);
24         // Output: "true"
25         Console.WriteLine(context.Result.Pop());
26
27         input.Value = "44";
28         expression.Interpret(context);
29         // Output: "false"
30         Console.WriteLine(context.Result.Pop());
31     }
32 }
```

```

1 class Context
2 {
3     public Stack<string> Result = new Stack<string>();
4 }
5
6 interface Expression
7 {
8     void Interpret(Context context);
9 }
10
11 abstract class OperatorExpression : Expression
12 {
13     public Expression Left { private get; set; }
14     public Expression Right { private get; set; }
15
16     public void Interpret(Context context){
17         Left.Interpret(context);
18         string leftValue = context.Result.Pop();
19
20         Right.Interpret(context);
21         string rightValue = context.Result.Pop();
22
23         DoInterpret(context, leftValue, rightValue);
24     }
25
26     protected abstract void DoInterpret(Context context, string leftValue,
27                                         string rightValue);
28 }
29
30 class EqualsExpression : OperatorExpression
31 {
32     protected override void DoInterpret(Context context, string leftValue,
33                                         string rightValue){
34         context.Result.Push(leftValue == rightValue ? "true" : "false");
35     }
36 }
37
38 class OrExpression : OperatorExpression
39 {
40     protected override void DoInterpret(Context context, string leftValue,
41                                         string rightValue){
42         context.Result.Push(leftValue == "true" || rightValue == "true" ?
43                             "true" : "false");
44     }
45 }
46
47 class MyExpression : Expression
48 {
49     public string Value { private get; set; }
50
51     public void Interpret(Context context){
52         context.Result.Push(Value);
53     }
54 }

```

Per completezza si fornisce anche il codice del parser:

```
1  private static Expr parseToken(String token, ArrayDeque<Expr> stack) {
2      Expr left, right;
3      switch(token) {
4          case "+":
5              // It's necessary to remove first the right operand from the stack
6              right = stack.pop();
7              // ...and then the left one
8              left = stack.pop();
9              return Expr.plus(left, right);
10         case "-":
11             right = stack.pop();
12             left = stack.pop();
13             return Expr.minus(left, right);
14         default:
15             return Expr.variable(token);
16     }
17 }
18 public static Expr parse(String expression) {
19     ArrayDeque<Expr> stack = new ArrayDeque<Expr>();
20     for (String token : expression.split(" ")) {
21         stack.push(parseToken(token, stack));
22     }
23     return stack.pop();
24 }
```

Un esempio di utilizzo:

```
1  public static void main(final String[] args) {
2      Expr expr = parse("w x z - +");
3      Map<String, Integer> context = Map.of("w", 5, "x", 10, "z", 42);
4      int result = expr.interpret(context);
5      System.out.println(result);           // -27
6 }
```

Riassunto

Segue la lista dei 23 pattern con un brevissimo riassunto per ciascuno di essi:

- **Strutturali**

1. **Adapter (adattatore)**: adattare l'interfaccia di una classe già pronta (solitamente il cui codice non è accessibile) all'interfaccia voluta dal client.
2. **Façade (facciata)**: fornire un'interfaccia più semplice di sottosistemi che presentano interfacce molto complesse e diverse/sparse tra loro.
3. **Composite (composto)**: permettere di trattare un gruppo di oggetti come se fossero l'istanza di un oggetto singolo; ovvero dare la possibilità di manipolare oggetti singoli e composizioni/gruppi di una data gerarchia in modo uniforme.
4. **Decorator (decoratore)**: aggiungere nuove funzionalità ad oggetti già esistenti a run-time, superando le limitazioni di un approccio statico (alternativa all'uso dell'ereditarietà).
5. **Bridge (ponte)**: separare l'interfaccia di una classe dalla sua implementazione, rendendo possibile, tramite l'ereditarietà, di farle evolvere in modo indipendente.
6. **Proxy (proxy)**: fornire un surrogato per un oggetto, al fine di controllarne/aricchirne l'accesso.
7. **Flyweight (peso piuma)**: separare la parte variabile (dello stato) di una classe dalla parte costante.

- **Creazionali**

8. **Factory method (metodo fabbrica)**: delegare l'istanziazione di una classe a una propria sottoclasse.
9. **Abstract factory (fabbrica astratta)**: fornire un'interfaccia per creare famiglie di oggetti connessi e/o dipendenti tra loro, permettendo al client di non specificare i nomi delle classi concrete.
10. **Singleton (singoletto)**: fornire la certezza che una classe abbia un'unica istanza al run-time e fornire un punto di accesso globale ad essa.
11. **Prototype (prototipo)**: rendere possibile la creazione di nuovi oggetti clonando un oggetto iniziale, detto prototipo.
12. **Builder (costruttore)**: separare la costruzione di un oggetto complesso dalla sua rappresentazione.

- **Comportamentali**

13. **Template method (metodo sagoma)**: definire la struttura generale di un algoritmo all'interno di un metodo, lasciandone alcune parti non specificate.
14. **Strategy (strategia)**: encapsulare un algoritmo all'interno di un oggetto, in maniera tale da risultare funzionale in quelle situazioni in cui sia necessario modificare gli algoritmi utilizzati a run-time.

15. **State (stato):** permettere ad un oggetto di cambiare il proprio comportamento a run-time in funzione dello specifico stato in cui si trova.
16. **Command (comando):** permettere di isolare/incapsulare la porzione di codice che effettua un'azione (eventualmente molto complessa) dal codice che ne richiede l'esecuzione, incapsulandola in un oggetto.
17. **Observer (osservatore):** osservare lo stato di un oggetto tramite degli oggetti (osservatori) che vengono notificati ed aggiornati ad ogni variazione.
18. **Mediator (mediatore):** modificare agilmente le politiche di interazioni tra oggetti, poiché le entità coinvolte devono far riferimento a un solo oggetto centrale (il mediatore).
19. **Memento (ricordo):** lo scopo è quello di catturare/estrarre lo stato interno di un oggetto, e di memorizzarlo, in modo da poterlo ripristinare in seguito, senza violarne l'incapsulamento.
20. **Iterator (iteratore):** fornire un metodo di accesso sequenziale agli elementi di un oggetto composto senza esporre la struttura di quest'ultimo, come ad esempio scorrere una lista o visitare un albero.
21. **Visitor (visitatore):** separare un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter rendere possibile l'aggiunta di nuove operazioni e comportamenti senza dover modificare la struttura stessa dell'oggetto su cui opera.
22. **Chain of responsibility (catena di responsabilità):** evitare l'accoppiamento fra mittente di una richiesta e il destinatario. Il pattern presenta più oggetti concatenati (handler) che possono soddisfare la richiesta.
23. **Interpreter (interprete):** definire una rappresentazione della grammatica di un linguaggio insieme ad un componente interprete che utilizza tale rappresentazione per l'interpretazione delle espressioni la cui sintassi rispetta il linguaggio.

6.1 8 Problemi evitabili con i pattern

Vediamo ora 8 problemi evitabili grazie all'uso dei pattern:

Problema 1: Creazione di un oggetto specificando esplicitamente la classe. Si ha dipendenza dall'implementazione anziché da una interfaccia. Le implementazioni però cambiano; meglio creare oggetti "indirettamente", rimanendo indipendenti dalle specifiche implementazioni; qualora possibile.

Pattern utili: Abstract Factory, Factory Method, Prototype.

Problema 2: Dipendenza da una particolare operazione; ci si lega a un modo preciso di soddisfare una richiesta.

Pattern utili: Command, Chain of Responsibility.

Problema 3: Dipendenza dalla piattaforma HW/SW; si progetta il sistema massimizzando l'indipendenza dalla piattaforma.

Pattern utili: Abstract Factory, Bridge.

Problema 4: Dipendenza dalle rappresentazioni interne di un oggetto o dalle implementazioni. Se il client sfrutta il modo in cui un oggetto è implementato, memorizzato, rappresentato o localizzato: modifiche all'oggetto potrebbero comportare modifiche al client. Si deve avere information hiding.

Pattern utili: Abstract Factory, Bridge, Memento, Proxy.

Problema 5: Dipendenza dagli algoritmi: vengono spesso modificati/estesi/ottimizzati/rimpiazzati durante sviluppo e riuso. Gli oggetti che vi dipendono verranno modificati. Si devono isolare gli algoritmi.

Pattern utili: Builder, Iterator, Strategy, Template Method, Visitor.

Problema 6: Accoppiamento stretto: classi fortemente interdipendenti portano a SW difficile da comprendere, modificare, manutenere, estendere, riusare, portare... Meglio progettare con accoppiamento lasco.

Pattern utili: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer.

Si noti come questi primi 6 problemi riguardano dipendenze e accoppiamento; quindi i pattern aiutano a risolvere problemi su dipendenze e accoppiamento.

Problema 7: Estendere funzionalità tramite sottoclassi: non sempre è semplice, richiede conoscenza profonda della sopraclasse, si ha alto accoppiamento con essa e si rischia di avere esplosione combinatoria. Inoltre, è un approccio statico.

In certi casi una alternativa è l'utilizzo della composizione e delega. Molti pattern "disciplinano" la creazione di sottoclassi.

Pattern utili: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy, State, Template Method, Visitor, ...

Problema 8: Impossibilità di modificare alcune classi, per mancanza del sorgente, o per eventuali effetti domino su sottoclassi, etc. Si pensi all'utilizzo dei framework.

Pattern utili: Adapter, Decorator, Visitor.

Tabella riassuntiva:

	Strutturali	Creazionali	Comportamentali				
Adapter	Proxy Bridge Decorator Composite Facade	Flyweight Factory M. Abstract F.	Builder Prototype Singleton	Strategy State Template M.	Observer Command Mediator Memento	Interpreter	
1: Creaz. specificando classe			●				
2: Dip. particolare operazione			●		●		●
3: Dip. piattaforma	●			●			
4: Dip. interno/impl.	●	●		●			●
5: Dip. algoritmi				●	●	●	●
6: Accoppiamento	●	●	●		●	●	●
7: + funz. con sottoclassi	●	●	●		●	●	●
8: Impossibile modifica classi	●	●					●

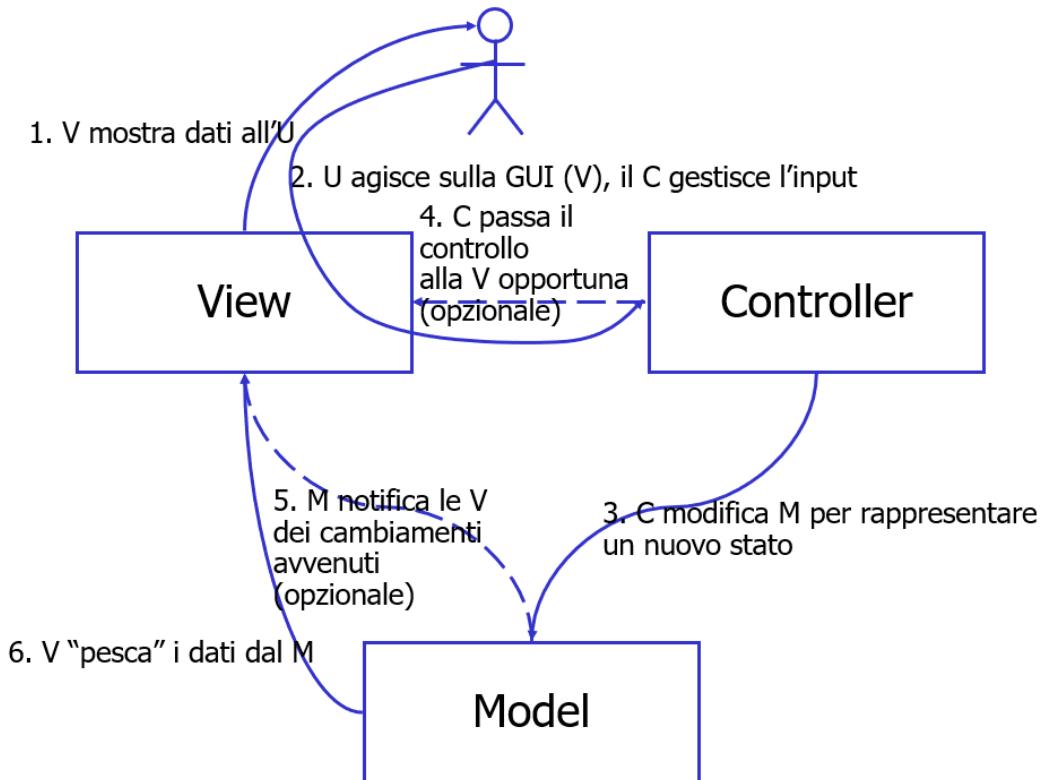
6.2 Cenni ai Pattern Architetturali

I pattern architetturali operano ad un livello diverso (e più ampio) rispetto ai design pattern, ed esprimono schemi di base per impostare l'organizzazione strutturale di un sistema software. In questi schemi si descrivono sottosistemi predefiniti insieme con i ruoli che essi assumono e le relazioni reciproche. Esempio: MVC pattern.

6.2.1 Model View Controller - MVC

Triade di classi/packages per costruzione di una GUI:

- Model: oggetto dell'applicazione, stato interno, logica di dominio
- View: rappresentazione a video - interfaccia utente
- Controller: gestore dell'input dell'utente



Si possono avere più VIEW; una per ogni "vista" dell'utente sull'applicazione. Se cambia la GUI, MODEL (che rappresenta la logica di dominio) viene riusato tale e quale, in quanto disaccoppiato da VIEW e CONTROLLER (che sono a sua volta disaccoppiati).

Design Pattern in MVC

- **Observer:** le view possono essere observer sul model;
- **Composite:** le view possono essere annidate; il composite consente di gestire una view composita come una view atomica;
- **Strategy (o State):** gerarchia di controller, per associare a una view il controller giusto fra una gerarchia di controller;
- **Decorator:** per modificare dinamicamente la/le view;
- **Command:** per rappresentare i comandi utente;
- **Factory Method:** gerarchia non solo di controller, ma anche di view.

7 Refactoring

Con refactoring si indica “una tecnica strutturata per modificare la struttura interna di porzioni di codice senza modificarne il comportamento esterno, al fine di migliorare alcune caratteristiche non funzionali del software quali leggibilità, manutenibilità, riusabilità, estensibilità, etc. nonché la riduzione della sua complessità”. Tale tecnica può avvenire attraverso l’introduzione a posteriori di design pattern.

Benché il concetto generale di refactoring possa essere applicato in qualsiasi contesto di sviluppo software, nelle metodologie agili e nell’extreme programming il termine è usato prevalentemente nel contesto della programmazione orientata agli oggetti.

L’azione di refactoring mira a eliminare il problema (per esempio portando a fattor comune il codice duplicato) attraverso una serie di ”micro-passi” il più possibile semplici. Il requisito di semplicità delle singole modifiche ha due giustificazioni:

- Ridurre il rischio di introdurre errori con la modifica;
- Rendere ipotizzabile l’esecuzione automatica della modifica stessa da parte di strumenti integrati negli IDE.

Gran parte della letteratura sul refactoring descrive tipi di micro-modifiche di uso comune che, combinate in sequenza, possono portare a ristrutturazioni anche radicali del software. Molte delle azioni di refactoring proposte in letteratura sono implementate da IDE moderni.

Il refactoring è un elemento integrante di molti processi di sviluppo fortemente basati su test automatici; per esempio, lo sviluppo basato su test (TDD) prevede una fase (obbligatoria ed esplicita) di refactoring al termine di ogni ciclo di modifica. Fra i due concetti esiste infatti un legame molto stretto: rieseguire eventuali test automatici al termine di ogni micromodifica fornisce infatti un più alto grado di confidenza che non siano stati introdotti errori; questo consente di prendere in considerazione anche modifiche particolarmente pericolose (come lo spostamento di codice fra classi o la modifica delle relazioni di ereditarietà).

I test possono essere eseguiti con framework come JUnit (in ambiente JAVA). JUnit è un framework open source utilizzato per effettuare testing unitario.

- I test unitari: test che verificano la correttezza direttamente del codice, in ogni sua piccola parte.
- I test funzionali: test che verificano che il software nel suo insieme funzioni correttamente.

I refactoring (o passi di refactoring) sono divisi in 6 gruppi:

1. Composizione di metodi (9)
2. Spostamenti fra oggetti (8)
3. Organizzazione dei dati (16)
4. Semplificazione di espressioni condizionali (8)
5. Semplificazione di invocazioni di metodi (15)

6. Gestione della generalizzazione (12)

Inoltre ci sono 4 “big-refactoring”, per un totale di 72.

Vediamo ora diverse tecniche di refactoring (piccoli passi di refactoring), a partire dalla categoria relativa alla composizione di metodi. Verranno illustrati tutti i 72 refactoring, alcuni in modo dettagliato, altri meno.

7.1 Composizione di Metodi

Questa categoria raggruppa 9 refactoring e mira ad arrivare a metodi ”ben fatti”; di qualità.

1. Extract Method
2. Inline Method
3. Inline Temp
4. Replace Temp with Query
5. Introduce Explaining variable
6. Split Temporary variable
7. Remove Assignments to parameters
8. Replace Method with Method Object
9. Substitute Algorithm

7.1.1 Extract Method

Si utilizza quando una porzione di codice raggruppata insieme viene divisa, estraendone porzione della logica in un nuovo metodo separato. Questo semplifica il primo metodo, rendendolo più leggibile. Si riduce la duplicazione del codice qualora questo nuovo metodo sia utile altrove.

```
void stampaRendiconto(double ammontare) {  
    stampaIntestazione();  
    // stampa dettagli  
    System.out.println("nome:" + nome);  
    System.out.println("ammontare:" + ammontare);  
}  
  
void stampaRendiconto(double ammontare) {  
    stampaIntestazione();  
    stampaDettagli(ammontare);  
}  
  
void stampaDettagli(double ammontare) {  
    System.out.println("nome:" + nome);  
    System.out.println("ammontare:" + ammontare);  
}
```



7.1.2 Inline Method

Quando il metodo il cui corpo è chiaro quanto il nome, posso sostituire l'invocazione al metodo direttamente con il suo corpo. È l'inverso dell'extract method. Ad esempio:

```
int getPuntualita() {
    return (piuDiCinqueSpedizioniInRitardo())?2:1;
}

boolean piuDiCinqueSpedizioniInRitardo() {
    return _numeroDiSpedizioniInRitardo > 5;
}
```



```
int getPuntualita() {
    return (_numeroDiSpedizioniInRitardo>5)?2:1;
}
```

7.1.3 Inline Temp

Elimino una variabile temporanea a cui è assegnato un unico valore.

```
double prezzoBase = ordine.prezzoBase();
System.out.println(prezzoBase);
return (prezzoBase > 100);
```



```
System.out.println(ordine.prezzoBase());
return (ordine.prezzoBase() > 100);
```

7.1.4 Replace Temp with Query

Nel caso in cui si memorizzi il risultato di un'espressione in una variabile locale che poi non viene utilizzata (var. temporale), si può muovere l'intera espressione in un metodo da cui ritornare il valore risultato. Si interroga il metodo invece che usare una variabile temporanea.

Esempio:

<pre>double calculateTotal() { double basePrice = quantity * itemPrice; if (basePrice > 1000) { return basePrice * 0.95; } else { return basePrice * 0.98; } }</pre>	<pre>double calculateTotal() { if (basePrice() > 1000) { return basePrice() * 0.95; } else { return basePrice() * 0.98; } } double basePrice() { return quantity * itemPrice; }</pre>
---	---

7.1.5 Introduce Explaining Variable

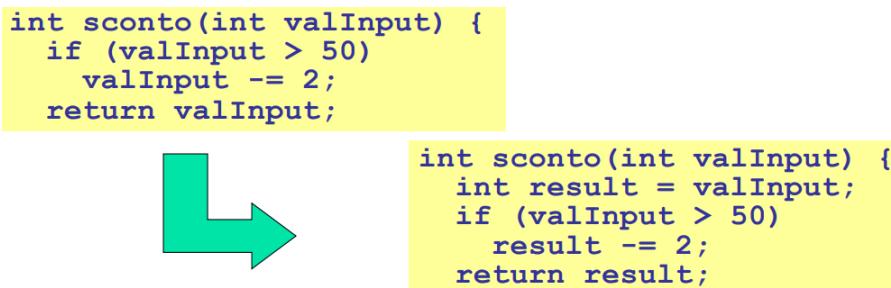
Se ho un'espressione complicata, posso mettere il risultato dell'espressione in una variabile temporanea con nome esplicativo (inverso di Inline Temp).

7.1.6 Split Temporary Variable

Se ho una variabile temporanea a cui vengono assegnati due valori scorrelati (la variabile temporanea viene utilizzata per "più cose") allora creo due variabili temporanee separate; una per ciascun assegnamento.

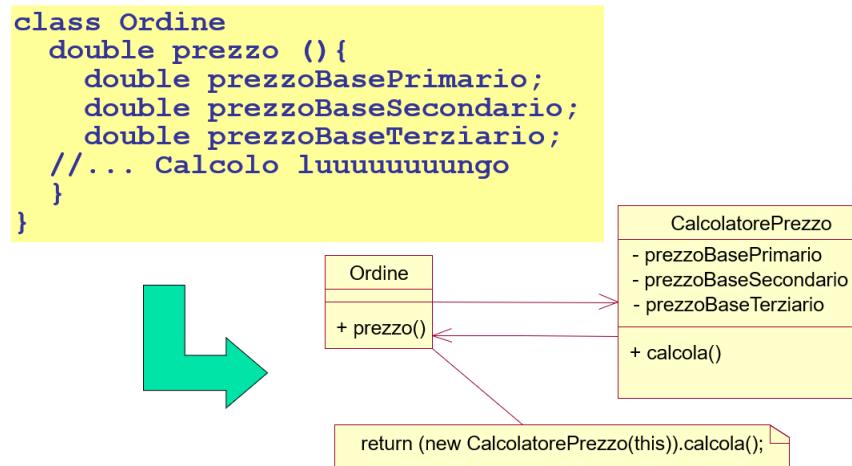
7.1.7 Remove Assignments to Parameters

Se il corpo di un metodo assegna un valore a un parametro, evito l'assegnamento e uso una variabile temporanea, ad esempio:



7.1.8 Replace Method with Method Object

Se ho un metodo lungo che usa parecchie variabili locali (e quindi extract method è difficile e complicato), posso creare una nuova classe che incapsula il metodo e in cui tutte le variabili locali diventano attributi.



7.1.9 Substitute Algorithm

Rimpiazzare un algoritmo complicato con uno semplice.

7.2 Spostamenti fra Oggetti

Vediamo ora la categoria di refactoring relativa allo spostamento fra oggetti.

1. Move Method
2. Move Field
3. Extract Class
4. Inline Class
5. Hide Delegate
6. Remove Middle Man
7. Introduce Foreign Method
8. Introduce Local Extension

7.2.1 Move Method

Quando un metodo è utilizzato più in altre classi che nella classe di appartenenza, forse è posizionato nella classe errata. Si crea un nuovo metodo nella classe che lo utilizza di più e si sposta il codice. Le invocazioni al vecchio metodo ora saranno un'invocazione al nuovo metodo.

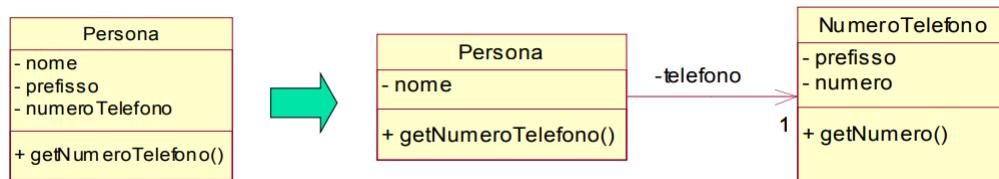
7.2.2 Move Field

Quando si ha un attributo in una classe errata, perché ad esempio viene utilizzato più nelle altre classi che in quella in cui è attualmente; lo sposto nella classe corretta.

7.2.3 Extract Class

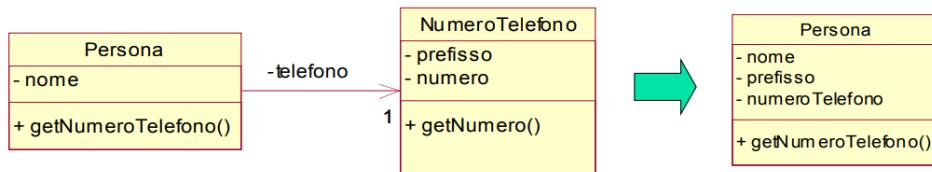
Quando ho una classe che fa due cose distinte, creo una nuova classe e vi sposto gli attributi e metodi opportuni.

- Move Field
- Move Method



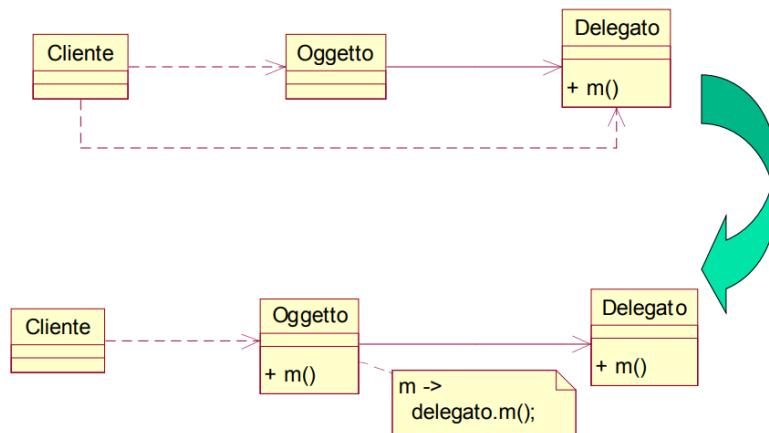
7.2.4 Inline Class

Si ha una classe che non fa molto, sposto tutti i suoi attributi e metodi e la rimuovo (inverso della extract class).



7.2.5 Hide Delegate

Si ha un cliente che utilizza un Oggetto e anche un Delegato dell’Oggetto: aggiungo un metodo a Oggetto per nascondere il Delegato.



Ovvero, evito che il cliente utilizzi direttamente i metodi del Delegato; ma lo costringo a passare sempre tramite l’Oggetto.

7.2.6 Remove Middle Man

Inverso di hide delegate. Il prezzo di hide delegate è di ”ingrassare” l’interfaccia di Oggetto; se ho tanti metodi, Oggetto fa essenzialmente da middleman; può essere meglio che il cliente chiami direttamente Delegato.

7.2.7 Introduce Foreign Method

Un cliente sta usando una classe Servitore e gli farebbe comodo che Servitore avesse un metodo in più, ma non la può modificare. Creo un metodo estraneo in cliente con primo argomento un’istanza di Servitore.

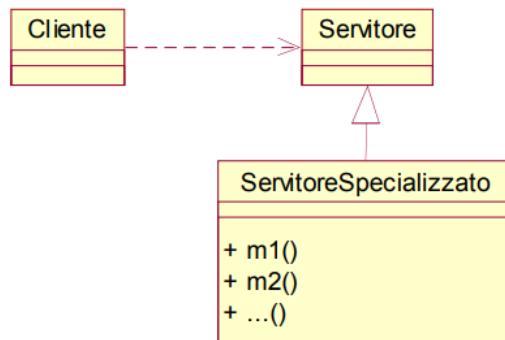
```
Date inizio2 = new Date(fine1.getYear(),  
                        fine1.getMonth(),  
                        fine1.getDay() + 1);
```



```
Date inizio2 = giornoDopo(fine1);  
...  
private static Date giornoDopo(Date arg) {  
    return new Date(arg.getYear(),  
                    arg.getMonth(),  
                    arg.getDay() + 1);  
}
```

7.2.8 Introduce Local Extension

Cliente sta usando una classe servitore e gli vorrebbe aggiungere vari metodi, ma non la può modificare. Creo una nuova classe con i nuovi metodi (come sottoclasse o come wrapper). Con più di 1-2 metodi, introduce foreign method diventa scomodo e introduce local extension è una scelta più appropriata.



7.3 Organizzazione dei Dati

Vediamo ora la terza categoria: Organizzazione dei dati.

1. Self Encapsulate Field
2. Replace Data Value with Object
3. Change Value to Reference
4. Change Reference to Value
5. Replace Array with objects
6. Duplicate Observed Data
7. Change Unidirectional Association to Bidirectional
8. Change Bidirectional Association to Unidirectional
9. Replace Magic Number with Symbolic Constant
10. Encapsulate Field
11. Encapsulate Collection
12. Replace Record with Data Class
13. Replace Type Code with Class
14. Replace Type Code with Subclasses
15. Replace Type Code with State/Strategy
16. Replace Subclass with Fields

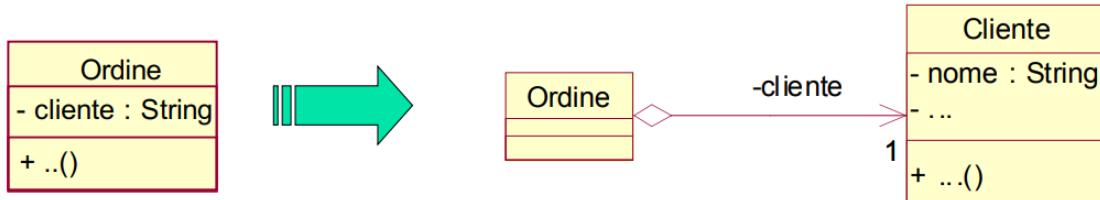
7.3.1 Self Encapsulate Field

Creo metodi `getXXX` e `setXXX` per accedere a un attributo `XXX`. Esistono due scuole di pensiero:

- Accedere direttamente all'interno della classe (più leggibile)
- Accedere sempre tramite get/set (sottoclassi possono sovrascrivere, lazy initialization, etc.)

7.3.2 Replace Data Value with Object

Una classe ha un attributo di un certo tipo (spesso un tipo base) a cui aggiunge comportamento/dati. Si crea per l'attributo, una classe apposita in cui metto comportamenti/dati aggiuntivi.



7.3.3 Change Value to Reference

Si ha una classe con molte istanze uguali, che però rappresentano la stessa entità del mondo; le rimpiazzo con una singola istanza.

7.3.4 Change Reference to Value

Faccio fatica a mantenere un'unica istanza per tutte le entità di un certo tipo uguali fra loro, le rimpiazzo con più istanze. Inverso del precedente.

7.3.5 Replace Array with Objects

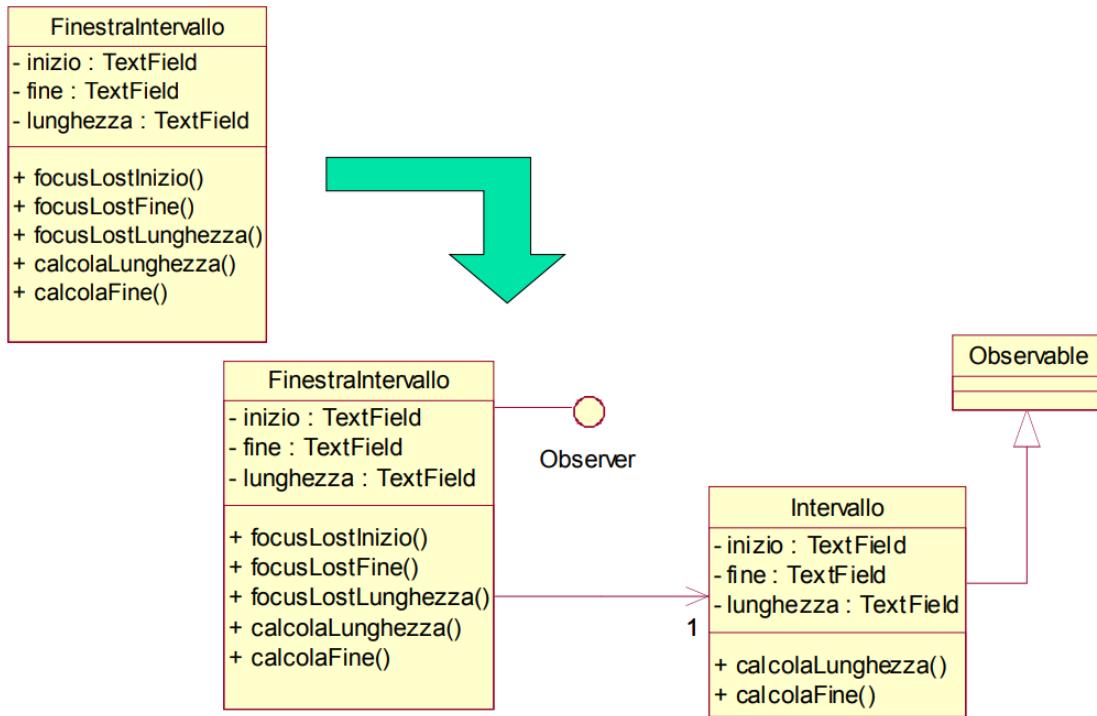
Un array contiene dati concettualmente diversi; trasformo l'array in un oggetto con un attributo per ogni elemento dell'array.

```
String[] indirizzo = new String[4];
indirizzo[0] = "Roma";
indirizzo[1] = "17";
...
```

```
Indirizzo indirizzo = new Indirizzo();
indirizzo.setVia("Roma");
indirizzo.setNumero("17");
...
```

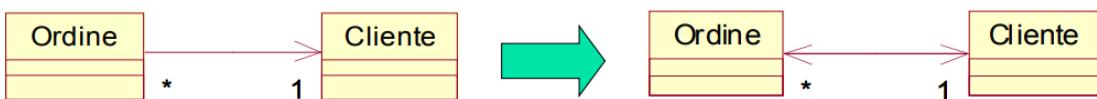
7.3.6 Duplicate Observer Data

Dati di dominio sono contenuti solo nelle classi della GUI, e le classi di dominio vi accedono. Copio i dati di dominio in un oggetto di dominio. Uso il pattern Observer per sincronizzare.



7.3.7 Change Unidirectional Association to Bidirectional

Ho due classi, una delle quali fa riferimento all'altra ma non viceversa. Ho bisogno anche del riferimento nell'altro senso. Aggiungo un back-pointer.



7.3.8 Change Bidirectional Association to Unidirectional

Inverso del precedente, le associazioni bidirezionali hanno dei costi:

- Una maggiore complessità di gestione;
- Interdipendenze fra classi, accoppiamento alto;
- Fonte di errori.

Quindi da evitare se non necessarie.

7.3.9 Replace Magic Number with Symbolic Costant

Esempio:

```
double energiaPotenziale(double m, double h) {  
    return m * 9.81 * h;  
}
```



```
double energiaPotenziale(double m, double h) {  
    return m * COSTANTE_GRAVITAZIONALE * h;  
}  
static final double COSTANTE_GRAVITAZIONALE = 9.81;
```

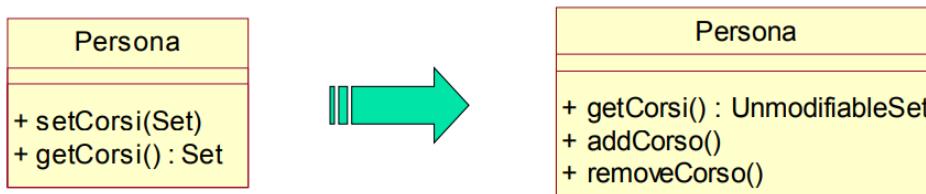
Sostanzialmente, il refactoring consiglia di utilizzare le costanti e non i letterali. Inoltre, si va a creare un unico punto di modifica nel codice; qualora si dovesse modificare il "magic number".

7.3.10 Encapsulate Field

Attributo public: renderlo private e aggiungere i metodi get/set; poi controllare se ha senso il move-method. La differenza con il self-encapsulate field è che qui l'attributo è public, non private.

7.3.11 Encapsulate Collection

Un metodo di una classe restituisce una collection, ma il cliente potrebbe modificarla. Rendo immutabile la collection restituita e aggiungo metodi add/remove.

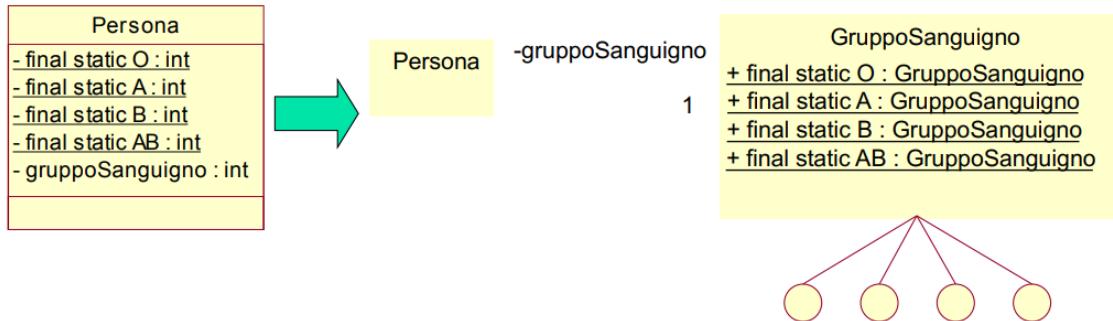


7.3.12 Replace Record with Data Class

Trasformo un record in una classe, con un attributo private per ogni campo del record e metodi set/get per ogni attributo.

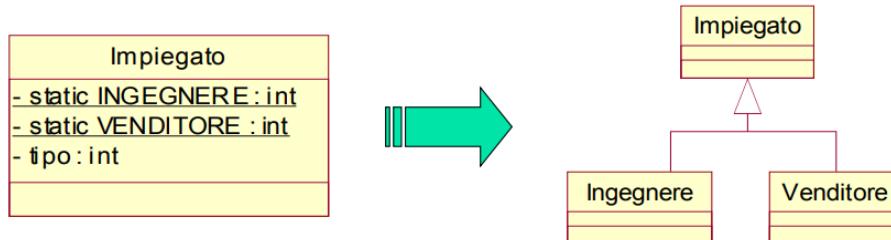
7.3.13 Replace Type Code with Class

Attributo di tipo numerico: lo trasformo in un attributo d'istanza di una nuova classe; I type code numerici non hanno controllo di tipo. Si possono usare gli ENUM (vedi Java, C#, etc.).



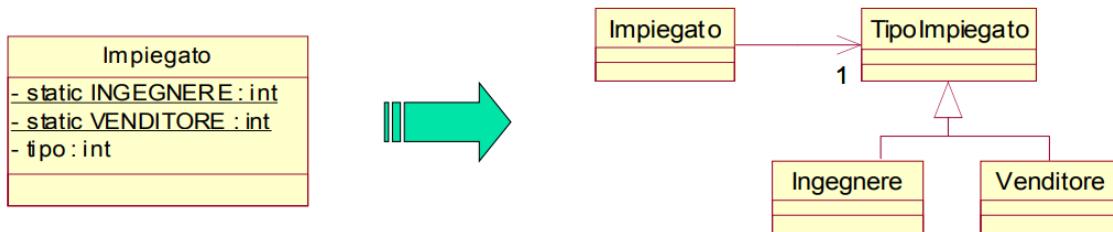
7.3.14 Replace Type Code with Subclasses

Simile al precedente, ma qui il type-code influenza il comportamento (sono presenti if e switch): si usa il polimorfismo.



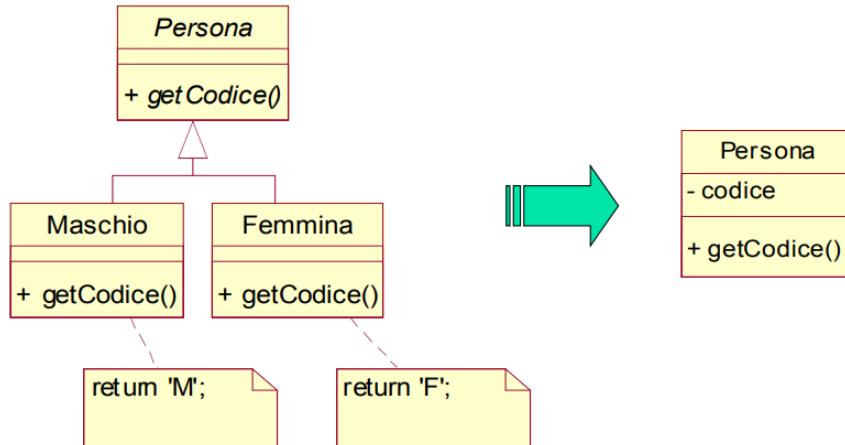
7.3.15 Replace Type Code with State/Strategy

Simile al precedente, ma da usare se il type-code cambia durante la vita dell'oggetto. Si usa il pattern State o Strategy; esempio di refactoring to pattern:



7.3.16 Replace Subclass with Fields

Ho sottoclassi che differiscono solo per metodi che restituiscono dati costanti: aggiungo un attributo nella superclasse ed elimino le sottoclassi.



7.4 Espressioni Condizionali

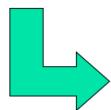
Vediamo ora la categoria semplificazione di espressioni condizionali.

1. Decompose Conditional
2. Consolidate Conditional Expression
3. Consolidate Duplicate Conditional Fragments
4. Remove Control Flag
5. Replace Nested Conditional with Guard Clauses
6. Replace Conditional with Polymorphism
7. Introduce Null Object
8. Introduce Assertion

7.4.1 Decompose Conditional

Si hanno if/then/else complicati: si estraggono metodi per la condizione, rami dell'if e dell'else; ad esempio:

```
if (data.primaDi(INIZIO_ESTATE) ||
    data.dopoDi(FINE_ESTATE))
    costo=qta*_tariffaInverno*_tariffaServizioInverno;
else
    costo = qta * _tariffaEstate;
```

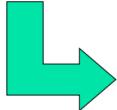


```
if (!inEstate(data))
    costo = costoInvernale(qta);
else
    costo = costoEstivo(qta);
... def metodi privati...
```

7.4.2 Consolidate Conditional Expression

Si ha una sequenza di condizioni con lo stesso risultato, le combino in un'unica espressione condizionale e la estraggo in un metodo.

```
double quantitaDisabilita() {
    if (_anzianita < 2) return 0;
    if (_mesi > 12) return 0;
    if (_partTime) return 0;
    // calcola la quantita' di disabilita'
}
```

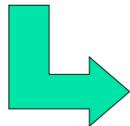


```
double quantitaDisabilita() {
    if (nonHaDiritto()) return 0;
    // calcola la quantita' di disabilita'
}
... def. Metodo/i...
```

7.4.3 Consolidate Duplicate Conditional Fragments

Lo stesso frammento di codice in tutti i rami di un if; si sposta al di fuori, esempio:

```
if (affareSpeciale()) {
    tot = prezzo * 0.95;
    spedisci();
} else {
    tot = prezzo * 0.98;
    spedisci();
}
```



```
if (affareSpeciale())
    tot = prezzo * 0.95;
else
    tot = prezzo * 0.98;
spedisci();
```

7.4.4 Remove Control Flag

Si ha una variabile di controllo; si utilizza break o return, ad esempio:

```
void m (String[] persone)
boolean trovato = false;
for (int i = 0; i < persone.length & !trovato; i++) {
    if(persone[i].equals("Meni")) {
        attenzione();
        trovato = true;
    }
    if(persone[i].equals("Toni")) {
        attenzione();
        trovato = true;
    }
}
```

The diagram illustrates the refactoring of a loop that uses a boolean variable to track a condition. On the left, the original code has a large green arrow pointing to the right. On the right, the transformed code shows two separate paths. The first path, which corresponds to the first if block, contains a blue box labeled "break;" with an arrow pointing from the original "trovato = true;" line. The second path, which corresponds to the second if block, also contains a blue box labeled "break;" with an arrow pointing from the original "trovato = true;" line. Both paths converge back to the original loop structure.

```
void m (String[] persone)
for (int i = 0; i < persone.length; i++)
    if(persone[i].equals("Meni")) {
        attenzione();
        break;
    }
    if(persone[i].equals("Toni")) {
        attenzione();
        break;
    }
}
```

7.4.5 Replace Nested Conditional with Guard Causes

Si hanno if/then/else con "pesi" diversi ai due rami; si usa if+return (guard clause), esempio:

```
double getAmmontare ()
double res = 0;
if (_morto) res = ammontareMorto();
else {
    if (_separato) res = ammontareSeparato();
    else{
        if (_pensionato) res = ammontarePensionato();
        else res = ammontareNormale();
    }
}
return res;
```

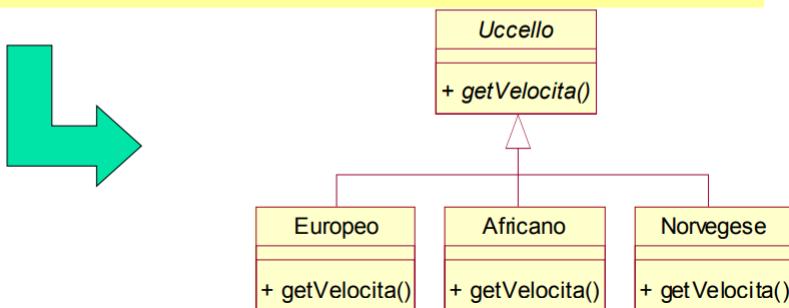
The diagram illustrates the refactoring of nested if/else statements into guard clauses. On the left, the original code has a large green arrow pointing to the right. On the right, the transformed code shows a single return statement that branches based on the conditions. Each branch is preceded by its respective if condition and followed by a return statement. The code is enclosed in a blue box.

```
double getAmmontare ()
    if (_morto) return ammontareMorto();
    if (_separato) return ammontareSeparato();
    if (_pensionato) return ammontarePensionato();
    return ammontareNormale();
```

7.4.6 Replace Conditional with Polymorphism

Istruzione condizionale con comportamento diverso a seconda del ramo; sposto ogni ramo in un metodo sovrascrivente in una sottoclasse e rendo astratto il metodo originale nella superclasse.

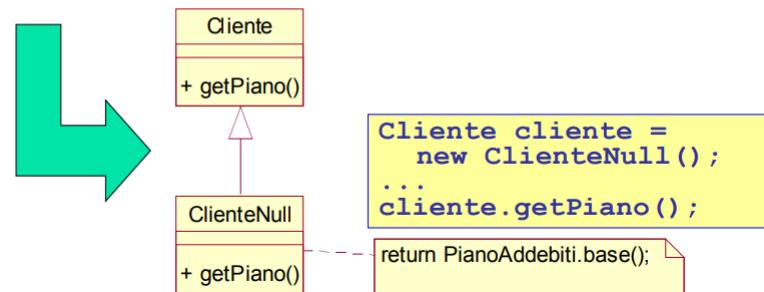
```
double getVelocita() {
    switch (_tipo) {
        case EUROPA: return getVelocitaBase();
        case AFRICA:
            return getVelocitaBase() - getCarico() -
                getNumeroNociDiCocco();
        case NORVEGESE:
            return (_inverno ? 0 : getVelocitaBase());
    }
    throw new RuntimeException("Troppo veloce?");
}
```



7.4.7 Introduce Null Object

Ho parecchi test per vedere se istanze di una certa classe sono in realtà null. Sostituisco il valore null con un oggetto null;

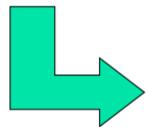
```
Cliente cliente;
...
if (cliente == null) piano = PianoAddebiti.base();
else piano = cliente.getPiano();
```



7.4.8 Introduce Assertion

Una parte di codice fa qualche assunzione sullo stato del programma: rendo esplicita l'assunzione con un'asserzione (assert)

```
// x deve essere positivo  
if (x >= 0) return Math.sqrt(x);
```



```
assert x >= 0;  
return Math.sqrt(x);  
  
assert x >= 0: "Errore: x negativo";  
return Math.sqrt(x);
```

7.5 Invocazione di Metodi

Vediamo ora la quinta categoria: Semplificazione di invocazioni di metodi.

- Rename Method
- Add Parameter
- Remove Parameter
- Separate Query from Modifier
- Parametrize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method
- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Encapsulate Downcast
- Replace Error Code with Exception
- Replace Exception with Test

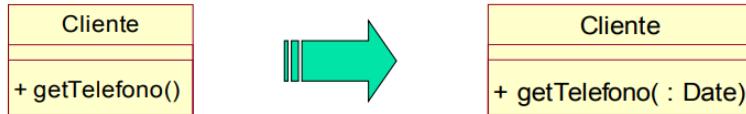
7.5.1 Rename Method

È importante eseguire una fase di valutazione e correzione dei nomi utilizzati nel sorgente. Un buon codice deve essere facilmente leggibile da altri programmati. È buona norma adottare convenzioni (tra developer dello stesso progetto) sulla nomina dei metodi e delle variabili.

7.5.2 Add Parameter

Un metodo ha bisogno di più dati; aggiungo un parametro. Esistono alternative spesso migliori come:

- Controllare che il metodo sia adeguato alla classe in cui è collocato
- Usare/aggiungere metodi di altri oggetti, "disponibili" al metodo
- Usare l'introduce parameter object



7.5.3 Remove Parameter

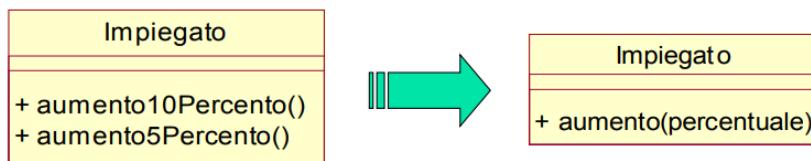
Se c'è un parametro non più usato, lo rimuovo; lasciarlo implica leggibilità inferiore.

7.5.4 Separate Query from Modifier

Ho un singolo metodo che restituisce un valore e modifica un oggetto: creo due metodi; uno get e uno set. I metodi che restituiscono un valore non dovrebbero avere side effect osservabili (get successivi danno valori diversi - ad esempio la memoization è un effetto collaterale non osservabile).

7.5.5 Parametrize Method

Ho molti metodi che fanno cose simili, che differiscono solo per un valore nel corpo: Creo un unico metodo con un parametro per quel valore.



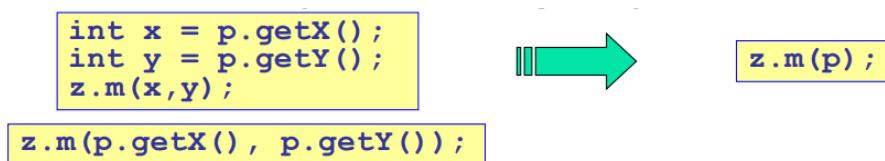
7.5.6 Replace Parameter with Explicit Methods

È l'inverso del precedente; ho un metodo che esegue codice diverso a seconda del valore di un parametro: Creo un metodo separato per ogni valore del parametro. Aumenta la chiarezza del codice, quando sensato, in casi ad esempio binari.

Ad esempio: `interruttore.on()` vs `interruttore.set(true)` o `interruttore.set(ON)`.

7.5.7 Preserve Whole Object

Ottengo (metodi get) vari valori da un oggetto e li passo tutti come parametri: meglio passare tutto l'oggetto come parametro (eventualmente `this`).



7.5.8 Replace Parameter with Method

Ho un oggetto che invoca un metodo `m` e ne passa il risultato come parametro a un altro metodo `m'`: rimuovo il parametro e lascio che sia il metodo `m'` a invocare il metodo `m`.

Si hanno lunghe liste di parametri; diminuisce la leggibilità.

Se un metodo può prendersi la responsabilità di ottenere un valore, non serve passarglielo come parametro.

7.5.9 Introduce Parameter Object

Ho un gruppo di parametri che "stanno bene insieme". Li rimpiazzo con un oggetto; ad esempio: `m(inizio: Date, fine: Date)` diventa `m(intervallo: Intervallo)`.

Si ha il beneficio di limitare lunghe liste di argomenti (che inducono ad un calo della leggibilità). Il raggruppamento aiuta a vedere comportamento che sta bene nella nuova classe.

7.5.10 Remove Setting Method

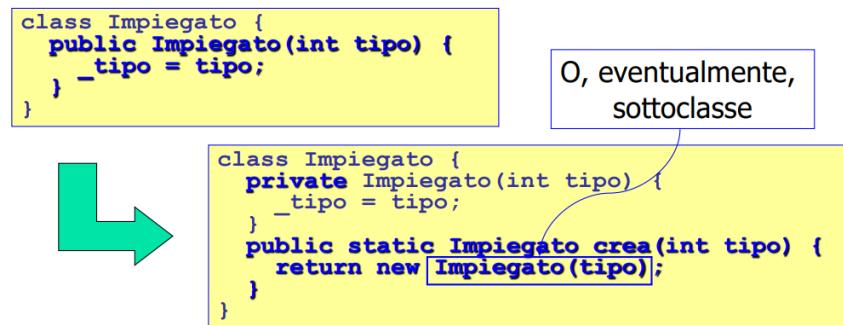
Ho un attributo `x` a cui va assegnato un valore solo alla costruzione. Rimuovo `setX` e se possibile rendo `x` final. In questo caso il metodo setter non verrebbe mai utilizzato.

7.5.11 Hide Method

Ho un metodo pubblico che non viene usato da nessun'altra classe; la soluzione è renderlo privato. L'ambiente di programmazione potrebbe/dovrebbe evidenziare i metodi privati e pubblici non utilizzati.

7.5.12 Replace Constructor with Factory Method

Sostituisco un costruttore con un factory method



7.5.13 Encapsulate Downcast

Ho un metodo che restituisce un oggetto che deve essere downcasted al tipo corretto. Sposto il downcast dentro il metodo. Questo cambia la firma del metodo, che restituirà un tipo diverso. Ma così facendo, il downcast non deve essere esplicito laddove viene utilizzato il risultato del metodo (punti multipli) ma solo nel codice del metodo.

7.5.14 Replace Error Code with Exception

Ho un metodo che restituisce un codice speciale per indicare un errore: va introdotta un'eccezione.

```
int preleva(int ammontare) {  
    if (ammontare > _saldo)  
        return -1;  
    else {  
        _saldo -= ammontare;  
        return 0;  
    }  
}  
  
void preleva(int ammontare) throws EccezioneSaldo{  
    if (ammontare > _saldo)  
        throw new EccezioneSaldo();  
    else  
        _saldo -= ammontare;  
}
```

7.5.15 Replace Exception with Test

Inverso del precedente; se la responsabilità è del chiamante, modifico il chiamante per fargli fare il test (progetto per contratto).

```
double getIesimo(int i) {  
    try {  
        return _array[i];  
    } catch (ArrayIndexOutOfBoundsException e) {  
        return 0;  
    }  
}  
  
double getIesimo(int i) {  
    if(i >= _array.length || i < 0)  
        return 0;  
    return _array[i];  
}
```

7.6 Gestione della Generalizzazione

Vediamo ora i refactoring della sesta categoria: gestione della generalizzazione.

1. Pull Up Field
2. Pull Up Method
3. Pull Up Constructor Body
4. Push Down Method
5. Push Down Field
6. Extract Subclass
7. Extract Superclass
8. Extract Interface
9. Collapse Hierarchy
10. Form Template Method
11. Replace Inheritance with Delegation
12. Replace Delegation with Inheritance

7.6.1 Pull Up Field

Si hanno due sottoclassi con lo stesso attributo; sposto l'attributo nella superclasse.

7.6.2 Pull Up Method

Ho due metodi con risultati identici in due sottoclassi; sposto il metodo nella superclasse.

7.6.3 Pull Up Constructor Body

Ho due costruttori quasi uguali in due sottoclassi. Creo un costruttore nella superclasse e lo chiamo (**super**) dalle sottoclassi. È diverso dal pull-up method perché i nomi dei costruttori sono diversi. Un'alternativa è "Replace Constructor with Factory Method" + "Pull up Method".

7.6.4 Push Down Method

Ho un comportamento non pertinente a una superclasse. Sposto nella/e sottoclasse/i. Inverso del Pull Up Method.

7.6.5 Push Down Field

Ho un attributo usato solo in alcune sottoclassi; lo sposto in quelle sottoclassi dove viene utilizzato. Inverso di Pull Up Field.

7.6.6 Extract Subclass

Ho una classe con qualche caratteristica usata solo in alcune istanze (classe con coesione a istanza mista - la peggiore). Creo una sottoclasse per quel sottoinsieme di caratteristiche.

7.6.7 Extract Superclass

Ho due classi con caratteristiche simili: Creo una superclasse e vi sposto le caratteristiche comuni. Eventualmente rinomino, rendo astratto, sovrascrivo...

7.6.8 Extract Interface

Ho molti clienti che usano solo una parte dell'interfaccia di una classe, oppure: due classi che hanno una parte dell'interfaccia in comune. Estraggo la parte comune in un'interfaccia.

7.6.9 Collapse Hierarchy

Ho una superclasse e una sottoclasse molto simili, le fondo assieme. È l'inverso di extract superclass/subclass. Spesso l'uso di tale refactor è una conseguenza del precedente uso di push up/down field/method.

7.6.10 Form Template Method

Ho in due sottoclassi due metodi che eseguono nello stesso ordine passi simili anche se differenti; costruisco nella superclasse un template method con metodi hook sovrascritti nelle sottoclassi.

7.6.11 Replace Inheritance with Delegation

Ho una sottoclasse che usa solo parte dell'interfaccia della superclasse, oppure sarebbe meglio se non ereditasse certi dati. Creo un attributo per la superclasse, modifco i metodi per far delegare e rimuovo la relazione di eredità.

7.6.12 Replace Delegation with Inheritance

Ho "troppa" delegazione banale (tanti metodi semplici che delegano e basta). Rendo il delegante una sottoclasse del delegato; inverso del precedente.

7.7 Bad Smells

Quando si fa refactoring? quando il codice presenta delle "smells"; vediamo una lista di bad smells. Una "smell" è un indizio che probabilmente una azione di refactoring è necessaria.

Definizione [Wikipedia]: Nell'ingegneria del software, e in particolare nel contesto dello sviluppo agile e dell'extreme programming, l'espressione code smell (letteralmente "puzza del codice") viene usata per indicare una serie di caratteristiche che il codice sorgente può avere e che sono generalmente riconosciute come probabili indicazioni di un difetto di programmazione. I code smell non sono (e non rivelano) "bug", cioè veri e propri errori, bensì debolezze di progettazione che riducono la qualità del software, a prescindere dall'effettiva correttezza del suo funzionamento. Il code smell spesso è correlato alla presenza di debito tecnico e la sua individuazione è un comune metodo euristico usato dai programmatore come guida per l'attività di refactoring.

7.7.1 Duplicated Code

Si ha lo stesso codice in:

- 2+ rami di un condizionale
- 2+ metodi della stessa classe
- 2+ sottoclassi sorelle
- 2+ classi non correlate

7.7.2 Long Method

Un metodo troppo lungo, troppe linee di codice. Tante variabili locali e/o parametri. Il 99% delle volte si risolve con 1.1 Extract Method.

7.7.3 Large Class

Si ha una classe con:

- tante variabili (attributi d'istanza)
- tanto codice
- large GUI Class

7.7.4 Long Parameter List

L'object oriented programming è diverso dalla programmazione procedurale dove bisogna passare tutto come parametri (alternativa unica sono le var. globali). Se ho troppi parametri è probabilmente necessario del refactoring: lasciamo che sia il destinatario a prelevarsi le informazioni che gli servono.

7.7.5 Divergent Change

Idealmente dovrebbe esserci una relazione 1:1 fra tipo di cambiamento (modifica del codice) e classe da modificare. Se diverse variazioni richiedono cambiamenti in punti diversi di una classe probabilmente più classi sarebbero meglio di una.

7.7.6 Shotgun Surgery

Concettualmente simile al precedente, ma una variazione richiede tante piccole modifiche in più punti del codice/più classi. Probabilmente è necessario del refactoring sulle classi.

7.7.7 Parallel Inheritance Hierarchies

Caso particolare del shotgun-surgery.

7.7.8 Feature Envy

Si ha un metodo che sembra più interessato a un'altra classe che a quella in cui è collocato. Tendenzialmente si distinguono due casi:

- oggetto dell'invidia: dati di un'altra classe (get)
- oggetto dell'invidia: dati in molte altre classi

Non sempre si deve mettere il metodo nella classe con i dati su cui si lavora: visitor, strategy... anche se concettualmente "mettere insieme le cose che cambiano insieme".

7.7.9 Data Clumps

Si hanno "grumi/blocchi" di dati (gruppi di attributi/parametri) che tendono a stare assieme; meglio metterli in una classe apposita.

7.7.10 Primitive Obsession

Non avere l'ossessione di fare piccole classi inutili per dati semplici/troppo semplici.

7.7.11 Switch Statements

Codice duplicato in switch statements invece che uso del polimorfismo.

7.7.12 Lazy Class

Classe che non sta facendo molto, quasi inutile. Forse è necessario del refactoring.

7.7.13 Speculative Generality

Speculando che in un futuro possa servire, si crea/generalizza troppo le classi/parametri, innondando il codice di "cose inutili". Il codice è eccessivamente generico.

7.7.14 Temporary Field

Oggetto con attributo d'istanza non sempre settato (coesione a istanza mista).

7.7.15 Message Chains

Lunghe catene di messaggi `getXXX` con eventualmente variabili temporanee. Si pensi alla legge di Demeter.

7.7.16 Middle Man

Troppi metodi pubblici di una classe delegano ad un'altra classe. Si può rimuovere il middle man o sostituire le deleghe con l'ereditarietà.

7.7.17 Inappropriate Intimacy

Classi troppo intime tra loro.

7.7.18 Alternative Classes with Different Interfaces

Classi che dovrebbero avere la stessa interfaccia ma non ce l'hanno.

7.7.19 Incomplete Library Class

Libreria a cui vorreste aggiungere metodi/servizi/funzionalità.

7.7.20 Data Class

Classi "stupide" con solo metodi get e set. Classi rappresentati dati ("data class").

7.7.21 Refused Bequest

Eredità rifiutata: sottoclassi che eredita metodi e attributi ma non li necessita/vuole.

7.7.22 Comments

Troppi commenti che spiegano il codice sono segno di scarsa "naturale" leggibilità del codice. Invece di commentare troppo il codice, lo si dovrebbe rendere più esplicativo. I commenti sono come il deodorante.

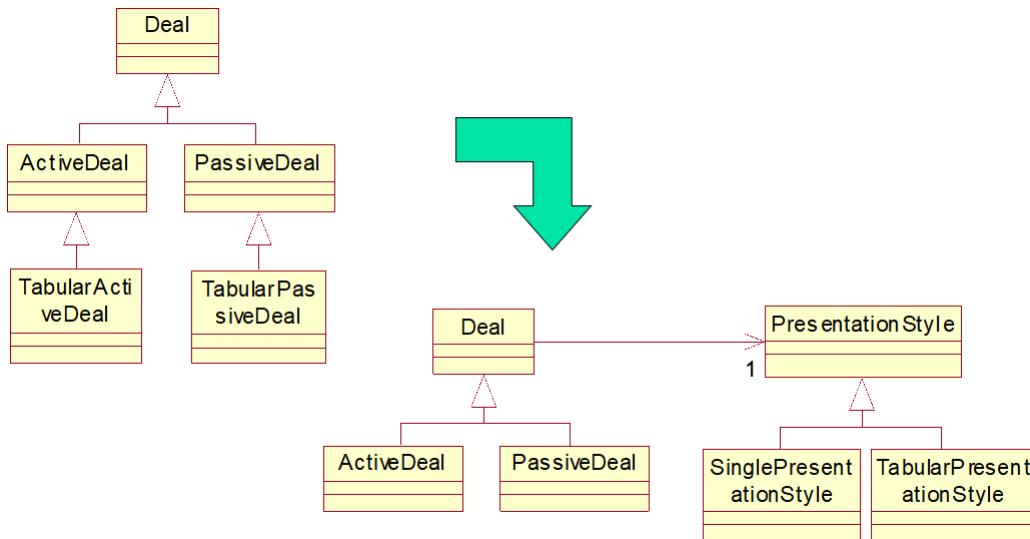
7.8 Big Refactorings

Questi refactoring sono molto complessi. Vedremo degli esempi esplicativi.

1. Tease Apart Inheritance
2. Convert Procedural Design to Objects
3. Separate Domain from Presentation
4. Extract Hierarchy

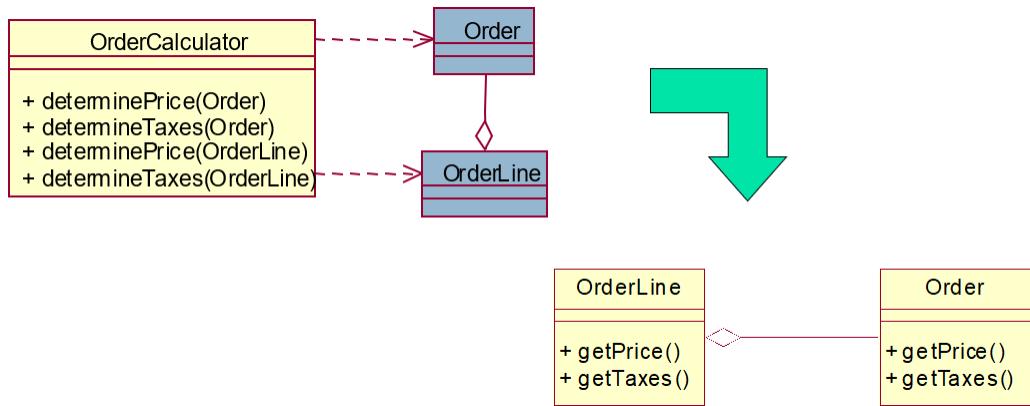
7.8.1 Tease Apart Inheritance

Sbrogliare gerarchie; esempio UML:



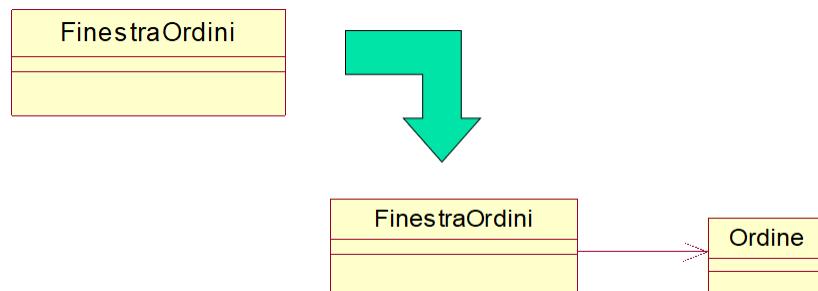
7.8.2 Convert Procedural Design to Objects

Da design procedurale all'uso degli oggetti.



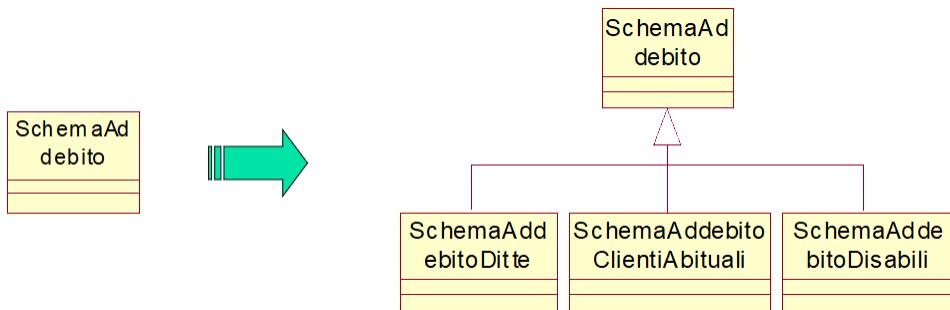
7.8.3 Separate Domain from Presentation

Separare l'interfaccia (GUI) e presentazione (logica di dominio).



7.8.4 Extract Hierarchy

Estrarre una gerarchia.



7.9 JUnit

È un framework open-source per il testing unitario. Permette elevata strutturazione dei test.

7.10 Profiling

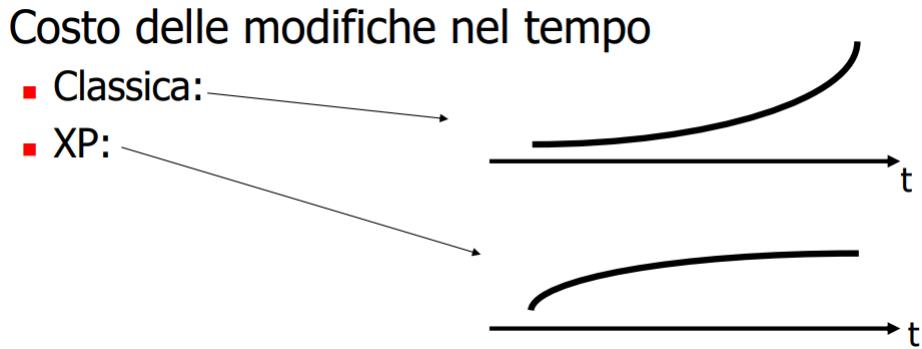
Non è necessario ottimizzare a caso, si perde tempo. La maggioranza del tempo il codice esegue su specifiche porzioni di sorgente (il 90% del tempo nel 10% del codice). Meglio utilizzare un profiler, al fine di migliorare le prestazioni di un software, mirando ad ottimizzare le parti dove il codice esegue per più tempo.

La filosofia di funzionamento di un profiler è a "campionamento". Si esegue il programma e si eseguono degli "snapshot" dello stack dei record di attivazione.

8 eXtreme Programming

Un "nuovo" approccio allo sviluppo di software, che pone il codice "al centro" del focus: tutti devono considerare essenziale la fase di programmazione (e non il 20% del progetto).

L'XP si pone l'obiettivo di ammortizzare i costi delle modifiche nel tempo:



L'XP è basato su 4 valori (principi/concetti astratti) e 12 pratiche (concrete).

8.0.1 4 Valori dell'XP:

- **Comunicazione:** massima informazione, tutti devono comunicare, nessun segreto, la mancanza di comunicazione porta a problemi nel software;
- **Semplicità:** fare le cose semplici ("la cosa più semplice che funziona"). Scrivere codice che "parla";
- **Feedback:** continuo dai clienti, che vengono aggiornati;
- **Coraggio:** si può buttare via codice e ridurne la complessità.

8.0.2 Le 12 Pratiche dell'XP

Seguono le 12 pratiche principali dell'Extreme Programming:

Brevi cicli di rilascio: un rilascio ogni 1-2 mesi; il più piccolo possibile e contenente le funzionalità più importanti (che devono quindi essere ordinate per priorità).

Metafora: ogni progetto si basa su una metafora, una storia sul sistema che chiunque può comprendere e raccontare: aiuta a capire essendo più comprensibile per il cliente.

Semplicità di progetto: il progetto deve essere il più semplice possibile che soddisfi i requisiti; si implementa quello che serve solo quando serve. Il progetto giusto deve far funzionare tutti i test e non avere logica duplicata. Non si programma con il focus sul riuso ma: "fare un lavoro buono e semplice oggi, ed essere confidenti di saper aggiungere nuove funzionalità in futuro".

Planning game: problema delle stime dei tempi; non ci deve essere una lotta tra sviluppatori e manager. Il manager decide cosa il sistema deve fare (priorità) e gli sviluppatori

stimano i tempi. Si chiama "gioco" perché si hanno tre mosse: esplorazione (cosa il sistema può fare), impegno (quali requisiti implementare) e gestione (dirigere lo sviluppo nella pratica).

Testing: scrivere prima i test e poi le funzionalità. I programmatori scrivono test unitari, il cliente definisce test funzionali.

Programmazione a coppie: tutto il codice deve essere scritto con due persone davanti al terminale. Ruoli precisi: uno scrive, l'altro lavora ad alto livello.

Proprietà collettiva: il codice non è di un programmatore in particolare, tutti hanno responsabilità di tutto e tutti devono potere e sapere modificare tutto. Chi vede la possibilità di migliorare il codice deve farlo, sempre.

Standard di codifica: tutti devono programmare con le stesse convenzioni.

Refactoring: bisogna ristrutturare il codice se si pensa di poterlo semplificare; si può fare senza pericolo poiché ci sono i test automatici.

Integrazione continua: il codice viene integrato ogni poche ore. Sempre seguendo la politica dei piccoli passi; si trovano subito incoerenze ed errori. Una sola macchina dedicata all'integrazione.

Settimana di 40 ore: per far funzionare le altre pratiche servono energie e freschezza. Se c'è un problema la sera, meglio risolverlo la mattina dopo piuttosto che di notte. Se si fa straordinario per 2 settimane di seguito si causano problemi al progetto.

Cliente sul posto: un cliente vero che userà il sistema, pagato dal committente, deve essere sempre disponibile (decisione priorità, feedback, risoluzione incongruenze). Scrive i test funzionali.

8.0.3 Sinergia

Molte delle 12 pratiche sono senza senso se isolate; ma efficaci se combinate con le altre. Si ha "extreme programming" se si applicano tutte le pratiche.

8.1 Agile Methods

L'Extreme Programming è un esempio di metodo agile, il più famoso. Ce ne sono altri, ad esempio:

- Agile Unified Process (AUP)
- Dynamic Systems Development Method (DSDM)
- Essential Unified Process (EssUP)
- Open Unified Process (OpenUP)
- Feature Driven Development (FDD)
- Scrum
- Velocity Tracking
- Etc.

I metodi agili danno più valore/focus a (agile manifesto):

- **Individui e interazione** che a processi e strumenti
- **Software funzionante** che a documentazione completa
- **Collaborazione con il cliente** che a negoziazione di un contratto
- **Rispondere ai cambiamenti** che a seguire un piano

I metodi agili cercano di minimizzare i rischi sviluppando SW durante iterazioni brevi (1-4 settimane).

12 Principi Agili:

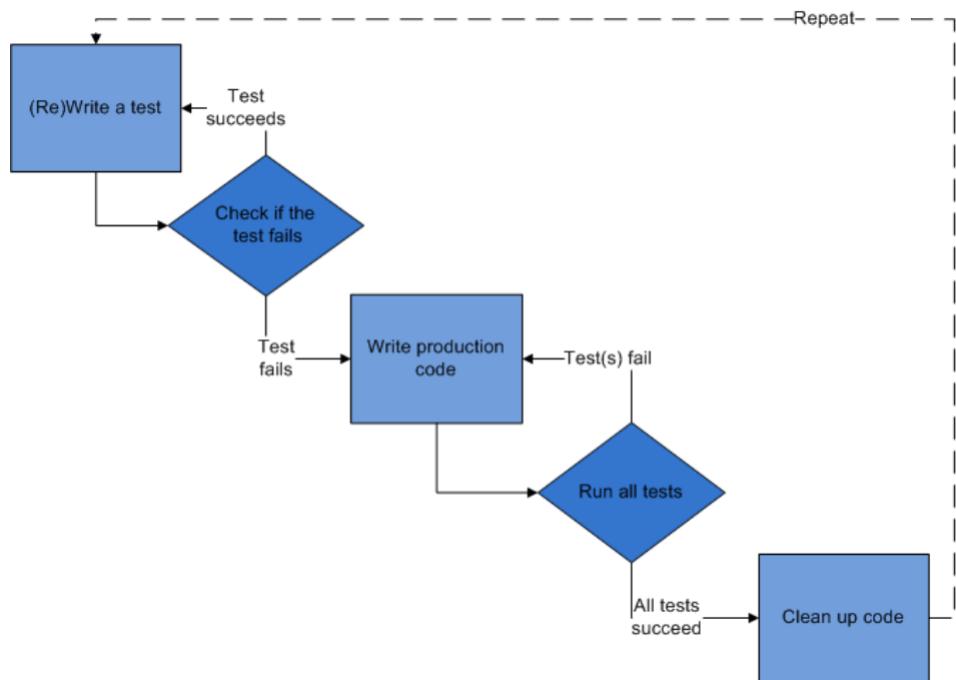
1. Soddisfazione dell'utente con delivery rapide di software utile
2. Accogliere cambiamenti nei requisiti, anche in fasi tardive dello sviluppo
3. Software funzionante (una prima versione) fornito prima possibile
4. Il software funzionante è la misura principale di progresso
5. Sviluppo sostenibile, capace di mantenere un ritmo costante
6. Cooperazione stretta e giornaliera tra team di sviluppo e stakeholders
7. Conversazioni face-to-face come principale forma di interazione con il cliente e tra membri del team
8. I progetti sono costruiti attorno a individui motivati e credibili

9. Attenzione continua all'eccellenza tecnica e al buon design
10. Semplicità di progetto
11. Team che si auto-organizzano
12. Adattamento regolare alle circostanze che cambiano

Tecniche e strumenti:

- Integrazione continua
- Automated unit testing
- Pair programming
- Test driven development TDD
- Design patterns
- Domain driven design DDD
- Refactoring
- Etc.

Test Driven Development Cycle:



8.1.1 Plan-Driven (classical/waterfall/predictive) VS Agile-Methods (adaptive)

Principali differenze:

- Predictive (classical, waterfall)

- Focus on planning the future in detail
- A predictive team can report exactly what features and tasks are planned for the entire length of the development process
- Difficulties in changing direction

- Adaptive (AM, XP)

- Adapting quickly to changing realities
- When the needs of a project change, an adaptive team changes as well
- Difficulties in describing exactly what will happen in the future

- Plan-driven:

- High criticality
- Junior developers
- Requirements do not change often
- Large number of developers
- Culture that demands order

- Agile:

- Low criticality
- Senior developers
- Requirements change often
- Small number of developers
- Culture that thrives on chaos

9 Analisi Orientata agli Oggetti - OOA

In generale, la programmazione Object-Oriented (inteso, il paradigma OO) ha più ripercussioni/conseguenze sul progetto che sull'analisi. Non è sorprendente, l'OO è nato come paradigma di programmazione; mentre la fase di analisi è più "vicina all'utente".

L'analisi a oggetti è un approccio all'analisi dei requisiti di un sistema software fondata sui concetti della programmazione ad oggetti. Tale analisi comprende tipicamente un'analisi del dominio il cui scopo è quello di descrivere in termini orientati agli oggetti le entità che costituiscono il dominio applicativo in cui il sistema deve operare. Questa operazione produce un modello a oggetti del dominio applicativo. Se l'analisi a oggetti è seguita dalla progettazione a oggetti, il modello prodotto in fase di analisi può solitamente essere utilizzato come modello di partenza anche per la fase di progettazione, che può procedere raffinando tale modello e arricchendolo di dettagli di livello più implementativo. Molti strumenti di modellazione a oggetti (per esempio UML) prevedono esplicitamente la creazione di modelli di analisi e di modelli di progetto integrati e correlati l'uno con l'altro.

Spazi del problema e della soluzione

Con spazio del problema ("analisi") ci si riferisce al modello del dominio (concetti, termini, classi concettuali), casi d'uso del sistema; "vicino" al committente.

Con spazio della soluzione ("progetto") ci si riferisce al modello del sistema (classi software); "vicino" al programmatore.

Nel paradigma object-oriented si ha isomorfismo parziale: il confine tra i due spazi diventa più labile/sfocato.

Diagrammi UML per l'analisi

- Di classe "concettuali": per rappresentare i concetti del dominio
- Dei casi d'uso: per rappresentare le funzionalità del sistema
- Meno frequenti ma utili:
 - Di interazione (sequenza)
 - Di attività: per flussi di operazioni complesse
 - Degli stati: per gli stati di entità del dominio
 - Di package

I diagrammi di classe hanno "due prospettive" (punti di vista diversi):

- **Software:** implementazione (corrispondenza con il codice) o di specifica (sempre vicino al codice, ma solo interfaccia, non implementazione).
- **Concettuale:** ogni classe è un concetto nella mente dell'utente o nel dominio.

Fino ad ora abbiamo analizzato solo la prima; più vicina alla progettazione object-oriented. Quelli concettuali, sono più orientati alla fase di analisi.

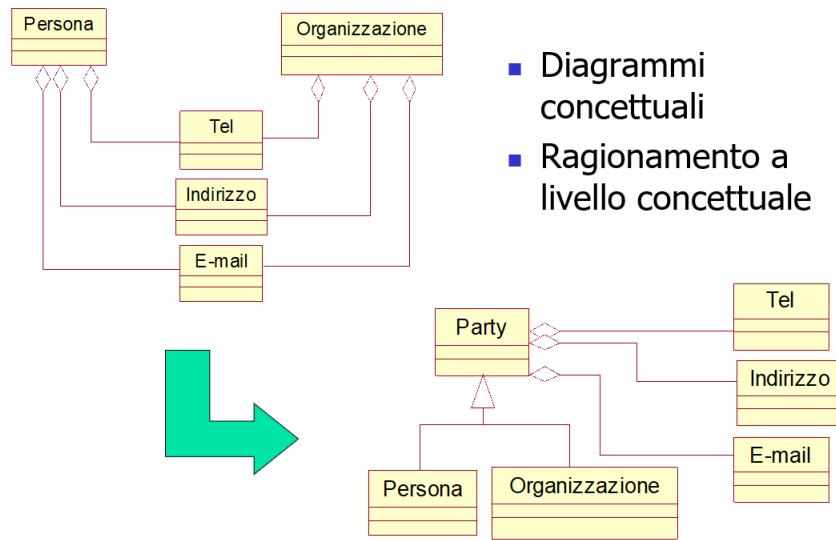
Diagrammi di classe concettuali

Ogni classe è un concetto del dominio. Ogni classe, poi, corrisponderà con molta probabilità a una classe del software: ma qui non si parla di software. Si fa modellazione di **dominio**:

- Studio del problema, non della soluzione
- Modellare/rappresentare/comprendere... il dominio
- Interagire con l'esperto di dominio
- Diagrammi di classe concettuali
- Costruzione di un vocabolario di concetti/terminologia del dominio

Le classi presenti sono quelle che servono a spiegare il dominio, non a "implementarlo". I diagrammi sono astratti, non vaghi: solo nomi di classe e relazioni, senza metodi e attributi.

Esempio:



Casi d'uso

Forniscono un modello funzionale del sistema e un suo possibile utilizzo: consistono in una sequenza di passi che descrivono un'interazione fra utente e sistema. Sono in forma testuale.

Esempio di caso d'uso:

Acquisto di un prodotto

1. Il cliente naviga nel catalogo, seleziona articoli da comprare
 2. Il cliente va alla cassa
 3. Il cliente indica le informazioni di spedizione
 4. Il sistema presenta il conto dettagliato
 5. Il cliente compila modulo per pagare con carta di credito
 6. Il sistema autorizza l'acquisto
 7. Il sistema conferma la vendita
 8. Il sistema invia un email di conferma
- Alternativa: carta di credito non valida
- 6'. Il sistema non autorizza e consente di riprovare
- Alternativa: cliente abituale
- 3a. Il sistema visualizza info di default, precedenti
 - 3b. Il cliente conferma il default o ri-inserisce

UML definisce i **diagrammi** dei casi d'uso: sono una rappresentazione grafica parziale dei casi d'uso; tendenzialmente non basta il diagramma e basta, la versione testuale è importante.

9.1 Pattern di Analisi

I pattern di analisi sono pattern a livello concettuale, da utilizzare durante l'analisi, non il progetto. Non sono pattern di design né pattern architetturali.

Rappresentano soluzioni ricorrenti in situazioni diverse, nella modellazione di domini diversi ("Pattern di dominio"). Vedremo i seguenti:

- Quantità (e Denaro)
- Intervallo
- Accountability (Responsabilità)
 - Party
 - Organization Hierarchy (Gerarchia organizzativa)
 - Accountability (Responsabilità)
- Pattern temporali:
 - Time point, Audit log, Effectivity, Temporal property, Snapshot, Temporal object

9.1.1 Quantity

Rappresenta valori con la loro unità di misura: combina ammontare e unità; fornisce un comportamento: operazioni aritmetiche, confronto, conversioni, etc.

Denaro-Valuta è la quantità più frequente.

Quantita	Unita
- ammontare : Number - unitaDiMisura : Unita	+ final static KM : Unita + final static KG : Unita + ... :
+ +, -, *, / + <, <=, = >, > + convertIn(Unita) + toString() + parse(String) : Quantita	- Unita() + toString()

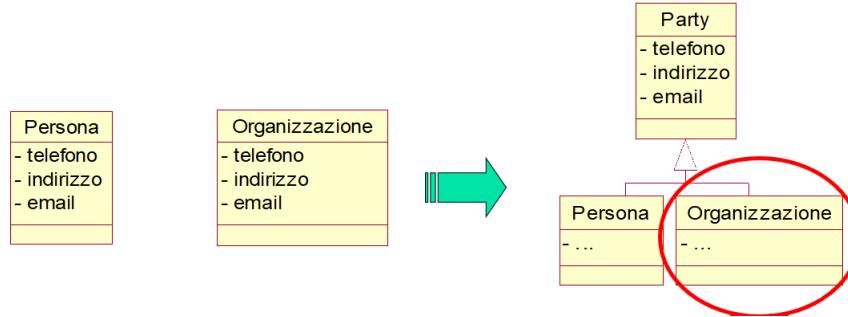
9.1.2 Intervallo (range)

Un intervallo di valori, meglio di due valori separati, uso naturale dei tipi parametrici.

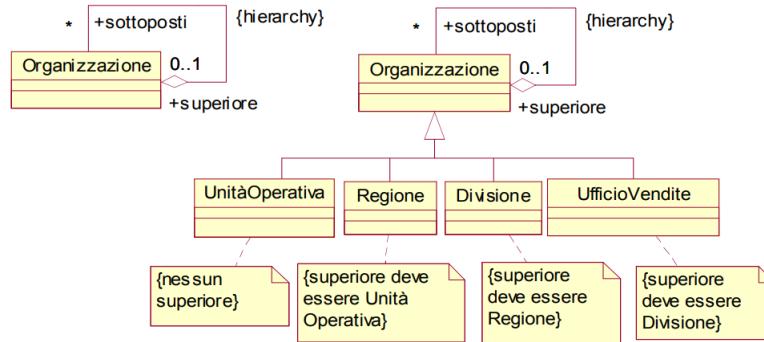
Intervallo
- inizio : Comparable - fine : Comparable
+ contiene(Comparable) : boolean

9.1.3 Accountability

L'essere responsabili, il dovere rendere conto a qualcun altro. Partendo dall'esempio del Party: esempio della rubrica, si generalizza e le trattiamo in modo uniforme.

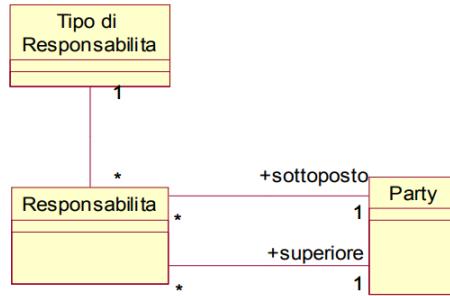


Gerarchia organizzativa: gestione dei cambiamenti più flessibile.



Con la gerarchia organizzativa, la gestione dei vincoli delle gerarchie è scomoda e va bene per gestire solo poche gerarchie (1,2,3... non troppe).

Per situazioni più complesse è meglio un altro pattern, Accountability (Responsabilità): rappresenta un grafo complesso di relazioni fra parti.



Ogni istanza di **Responsabilità** è una relazione fra due **Party**... e il tipo di relazione è indicato da **Tipo di Responsabilità**.

9.1.4 Pattern Temporali

Vengono utilizzati per "cose che cambiano nel tempo". Da usare per rispondere a domande sullo stati dei dati nel passato.

- **Time Point:** rappresenta un istante temporale (time point) a un qualche livello di granularità.
- **Audit Log:** ogni volta che accade un evento interessante si scrive una registrazione contenente che cosa è accaduto e quando.
- **Effectivity:** aggiungo un intervallo (l'altro pattern di analisi) temporale a un oggetto per indicare quando è "in vigore/attivo".
- **Temporal Property:** fornisce un'interfaccia (metodi della classe) regolare e predicibile per le proprietà che variano nel tempo.
- **Snapshot:** vista di un oggetto a un certo istante da cui sono rimossi tutti gli aspetti temporali; lo snapshot stesso ha un timestamp.

- **Temporal Object:** non è una proprietà dell'oggetto ad essere temporale ma l'oggetto stesso che ha varie versioni/revisioni.

10 Sistemi Multi Agente

In questo capitolo viene fornita una breve introduzione/infarinatura ai sistemi multi agente.

Un **agente** è un sistema computerizzato situato in un ambiente, capace di azione autonoma nell'ambiente in questione; con il fine di raggiungere gli obiettivi per cui è stato progettato. È autonomo: capace di effettuare decisioni indipendenti sulle azioni da intraprendere.

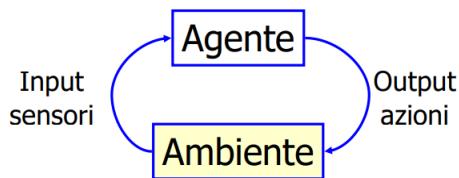
Un **sistema multi agente (SMA)** è un insieme di agenti che interagiscono l'uno con l'altro, tipicamente scambiandosi messaggi per mezzo di qualche infrastruttura di rete.

Due problemi:

- Come costruire un agente autonomo con comportamento autonomo? "Agent design"
- Come costruire un insieme di agenti che interagiscono per raggiungere un obiettivo? "society design"
- Sono due problemi diversi ma correlati.

Interazione con l'ambiente

- Un agente interagisce con l'ambiente;
- Ci sono ambienti di vario tipo;
- Qualsiasi cosa può essere un agente: termostato, robot, sistema operativo, etc.



Tipologie di ambiente:

- **Accessibile (o inaccessibile)**: l'agente può ottenere informazioni complete
- **Deterministico (o non deterministico)**: ogni azione ha un unico effetto conosciuto/determinabile a priori
- **Statico (o dinamico)**: rimane costante fra un'azione e l'altra
- **Discreto (o continuo)**: ha un numero di stati finito

Agenti intelligenti:

- **Reattività**: rispondere velocemente agli stimoli dell'ambiente

- **Proattività:** prendere l'iniziativa per raggiungere i propri obiettivi
- **Abilità sociale:** interagire con altri agenti (negoziazione e cooperazione)

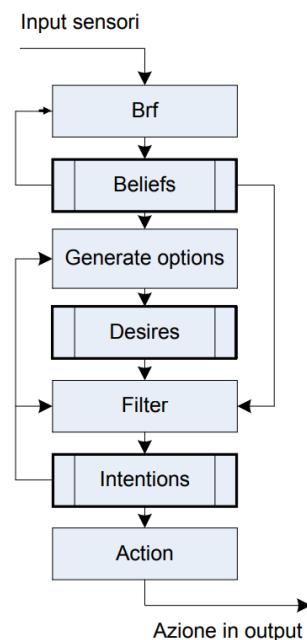
Agenti con stati mentali:

- **Credenze:** (supposizioni) sullo stato dell'ambiente
- **Desideri:** sullo stato dell'ambiente
- **Intenzioni:** di fare qualcosa - azioni
- **Impegni:** a fare qualcosa - obiettivi

La programmazione basata sugli agenti è fondata su tre insiemi: Credenze (beliefs), Desideri (desires) e Intenzioni (intentions) e su quattro funzioni:

- **belief revision function**
 - $\{Beliefs \times Input\} \rightarrow \{Beliefs\}$
- **option generation function**
 - $\{Beliefs \times Intentions\} \rightarrow \{Desires\}$
- **filter (funzione di deliberazione)**
 - $\{Beliefs \times Desires \times Intentions\} \rightarrow \{Intentions\}$
- **action selection function**
 - $\{Intentions\} \rightarrow Action$

- *Belief revision function* valuta le credenze attuali dell'agente e gli input che riceve dall'ambiente, genera un nuovo insieme di credenze. È necessaria affinché l'agente si evolva con l'ambiente che lo circonda.
- *Option generation function* partendo dalle credenze e dalle precedenti intenzioni dell'agente genera un nuovo insieme di desideri. È necessaria per permettere all'agente di modificare i propri obiettivi qualora i precedenti non siano più perseguitibili, interessanti o siano già stati raggiunti.
- *Filter* valuta tutti gli elementi dei tre insiemi, determina un nuovo insieme d'intenzioni dell'agente e rappresenta la vera e propria funzione di deliberazione dell'agente.
- *Action selection function*, determina un'azione da intraprendere partendo dall'insieme corrente d'intenzioni



Strategia dominante

Una strategia s_1 domina un'altra strategia s_2 sse ogni risultato dell'ambiente ottenuto con s_1 è preferito a tutti i risultati ottenibili con s_2 (quindi s_2 non ha mai senso).

La strategia dominante è quella che domina tutte le altre.

Equilibrio di Nash

Due strategie s_1 e s_2 sono in equilibrio di Nash sse:

- Assumendo che l'agente i segua s_1 , j non può fare meglio che seguire s_2 ;
- Assumendo che l'agente j segua s_2 , i non può fare meglio che seguire s_1 .

I due agenti sono "incastrati": a nessuno conviene uscire dall'equilibrio con adesempio una terza strategia che per entrambi porterebbe maggiore utilità. Non sempre esiste un equilibrio di Nash.