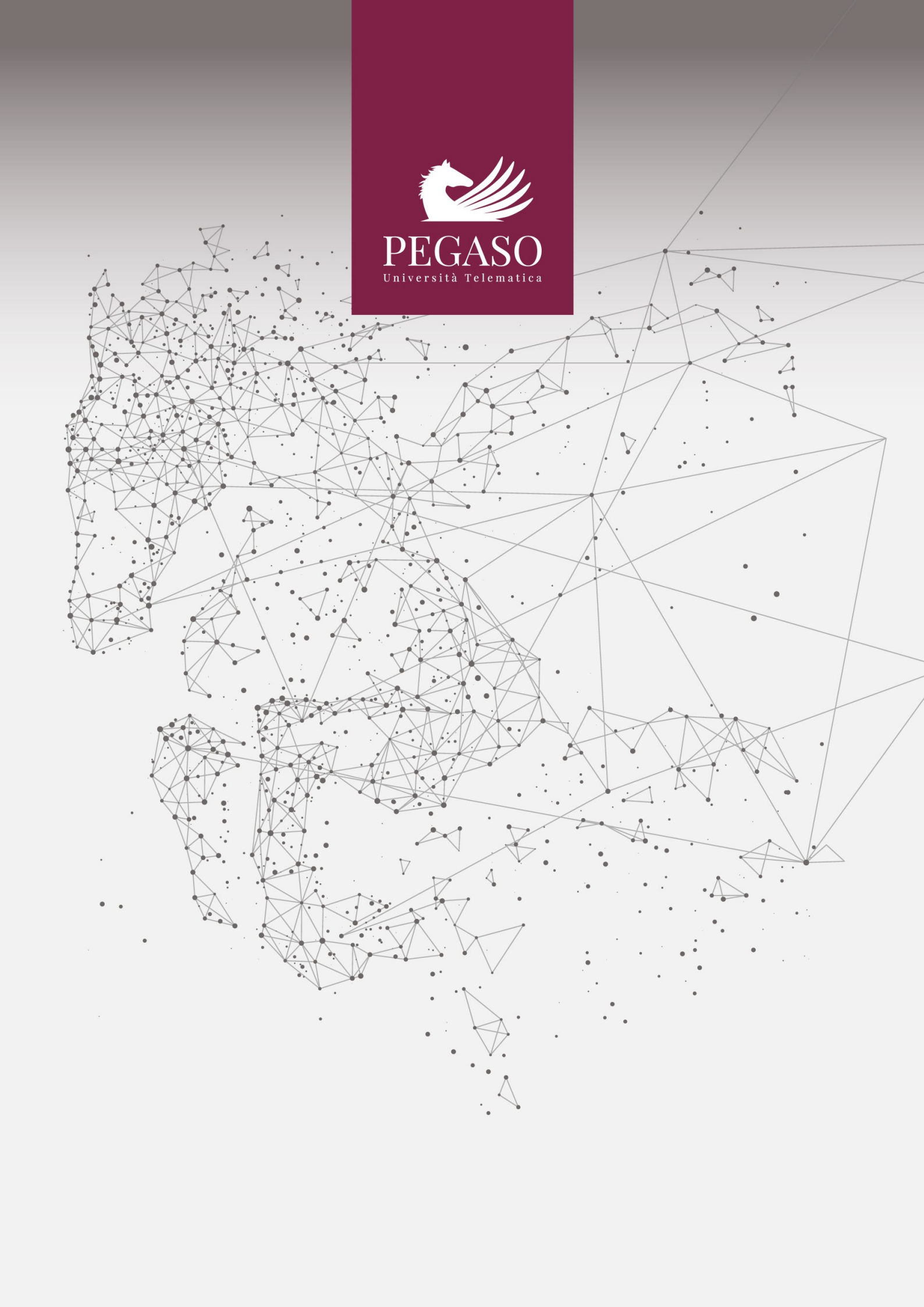

PEGASO
Università Telematica

Indice

1. Origine e definizione

Nel contesto dello sviluppo software moderno, i design pattern rappresentano uno strumento essenziale per progettare sistemi complessi in modo ordinato, efficiente e riusabile. Comprendere i design pattern significa, innanzitutto, riconoscerli come soluzioni standardizzate a problemi di progettazione ricorrenti. Ogni pattern incarna una strategia già collaudata per affrontare una determinata difficoltà architetturale o logica, facilitando la creazione di strutture software più robuste, manutenibili e facilmente comprensibili da parte di team anche numerosi e distribuiti.

Il principale vantaggio nell'impiego dei pattern risiede nella loro capacità di favorire il riuso del codice, contribuendo allo sviluppo di soluzioni sintetiche ed eleganti. Non si tratta di semplici "scorciatoie", ma di vere e proprie astrazioni progettuali, nate da anni di esperienza collettiva e formalizzate in modo tale da poter essere riutilizzate efficacemente in un'ampia varietà di contesti. I pattern promuovono anche la manutenibilità del software, offrendo schemi di progetto noti e ben documentati che possono essere compresi e modificati più agevolmente rispetto a soluzioni ad hoc.

Un ulteriore vantaggio consiste nella possibilità di standardizzare le soluzioni progettuali, rendendo il codice non solo più leggibile, ma anche più facilmente comunicabile tra sviluppatori. I pattern funzionano infatti come un linguaggio comune: una volta che un certo pattern è stato identificato all'interno di un progetto, i membri del team possono comprenderne l'intento e la struttura senza bisogno di lunghi commenti o spiegazioni. Questo favorisce anche una documentazione più efficace e mirata.

Per approfondire lo studio dei pattern esiste una vasta letteratura, ma il testo considerato fondamentale in questo ambito è "Design Patterns: Elements of Reusable Object-Oriented Software", scritto da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, noto anche come "il catalogo dei GoF" (Gang of Four). Pubblicato da Addison-Wesley, il volume propone una raccolta sistematica dei pattern di design più ricorrenti, presentandoli in modo strutturato e mettendo in evidenza le relazioni concettuali tra di essi. Questo catalogo costituisce una risorsa fondamentale per ogni sviluppatore che voglia approcciare in modo solido e metodico la progettazione orientata agli oggetti.

Il testo utilizza la notazione OMT (Object Modeling Technique), successivamente confluita nello standard UML (Unified Modeling Language), e fornisce esempi di codice in linguaggio C++. Tuttavia, la validità dei concetti esposti trascende il linguaggio di programmazione: i pattern descritti sono linguaggio-indipendenti e applicabili in maniera efficace in ambienti di sviluppo diversi, come Java, Python, C# e molti altri. L'adozione dei pattern favorisce quindi un approccio progettuale rigoroso, capace di coniugare astrazione e concretezza, teoria ed esperienza.

Un design pattern può essere definito come una soluzione progettuale astratta e riusabile per un problema ricorrente nel contesto dello sviluppo software. Si tratta, in altre parole, di un modello concettuale che descrive come risolvere un determinato problema di progettazione all'interno di un contesto specifico, sfruttando un'architettura già testata in molteplici situazioni reali. I pattern non sono frammenti di codice pronti all'uso, bensì schemi di progettazione che guidano il programmatore nella creazione di strutture software più solide, flessibili e comprensibili.

Gli sviluppatori esperti, soprattutto nel campo della programmazione orientata agli oggetti, tendono ad affrontare nuovi progetti applicando strategie già utilizzate con successo in passato. Queste strategie, una volta formalizzate, diventano pattern riconosciuti e condivisi nella comunità. In questo modo, l'esperienza dei progettisti viene trasformata in conoscenza strutturata che può essere tramandata, studiata e applicata in contesti sempre diversi. La natura riusabile di un pattern consente infatti di estendere la sua validità a più domini applicativi, indipendentemente dalle specifiche tecnologie utilizzate.

L'adozione dei design pattern consente di proporre soluzioni sintetiche, eleganti ed efficienti, ma ciò non significa che siano sempre di immediata comprensione. Al contrario, in molti casi, un pattern può apparire inizialmente controintuitivo, soprattutto per chi affronta un determinato problema per la prima volta. Questo avviene perché i pattern non riflettono necessariamente l'approccio ingenuo e diretto alla risoluzione di un problema, bensì propongono soluzioni raffinate, spesso basate su principi di astrazione e responsabilità ben definiti. Tuttavia, una volta compresi, i pattern permettono di affrontare situazioni complesse con maggiore consapevolezza e precisione, riducendo tempi di sviluppo e margini di errore.

Un ulteriore vantaggio dei pattern consiste nella loro capacità di offrire alternative progettuali ben documentate, tra cui il programmatore può scegliere in base alle esigenze specifiche del progetto. Questo processo di scelta consapevole non solo migliora la qualità architetturale del sistema, ma stimola anche la crescita professionale del progettista, che impara a confrontarsi con diverse strategie e compromessi.

Infine, i design pattern contribuiscono significativamente alla manutenibilità e alla documentazione del software. Utilizzando un pattern noto, il progettista fornisce automaticamente un'indicazione chiara della logica seguita, riducendo la necessità di spiegazioni verbose. Chiunque legga o debba modificare il codice può, riconoscendo il pattern impiegato, comprenderne rapidamente le intenzioni e i meccanismi, rendendo il lavoro collaborativo più efficace e meno soggetto a fraintendimenti.

Il concetto di design pattern può essere ulteriormente chiarito distinguendo tra due tipologie principali: i pattern orizzontali e i pattern verticali. I primi si riferiscono a soluzioni applicabili in modo trasversale a diversi sistemi software, soprattutto in relazione a problematiche generali come la gestione della concorrenza, la creazione e la comunicazione tra oggetti, o la gestione della persistenza. I pattern verticali, invece, si applicano a domini applicativi specifici, come ad esempio lo sviluppo di applicazioni web,

la costruzione di interfacce grafiche utente (GUI), o la progettazione di software embedded. Questa classificazione permette di comprendere meglio l'ambito e il contesto di applicabilità di ciascun pattern, fornendo una guida utile nella scelta dello strumento più adatto a risolvere un determinato problema progettuale.

Una delle definizioni più celebri e influenti del concetto di pattern è stata proposta dall'architetto Christopher Alexander nel 1977. Pur operando nel campo dell'architettura, Alexander ha formulato una visione talmente generale da risultare perfettamente applicabile anche al mondo del software engineering. Secondo la sua definizione, "ogni pattern descrive un problema che si ripete più e più volte nel nostro ambiente, descrive poi il nucleo della soluzione del problema, in modo tale che si possa usare la soluzione un milione di volte, senza mai applicarla nella medesima maniera". Questa affermazione racchiude uno dei principi fondamentali dell'approccio basato sui pattern: la riusabilità senza rigidità.

L'idea che si possa applicare una stessa soluzione in molteplici contesti, ogni volta in modo diverso, è di cruciale importanza nello sviluppo software. Infatti, un pattern non è mai una rigida prescrizione operativa, bensì una guida strutturata e flessibile che consente di adattare la logica della soluzione alle caratteristiche specifiche del problema in esame. L'obiettivo è quindi quello di identificare l'essenza della soluzione, ovvero il suo nucleo invariabile, che può essere riutilizzato senza imporre un'unica forma di implementazione.

Il valore universale della definizione di Alexander dimostra come i pattern non siano legati a un particolare dominio disciplinare, ma rappresentino piuttosto un modo di pensare i problemi e le soluzioni, basato sull'osservazione dell'esperienza, sull'astrazione e sulla formalizzazione. Nel contesto del software, questa impostazione consente di costruire sistemi più coerenti, robusti ed evolvibili, facendo tesoro delle esperienze già maturate da altri progettisti in situazioni analoghe.

Per comprendere appieno il significato di design pattern e la sua applicabilità in ambiti progettuali diversi, è utile analizzare un esempio concreto tratto proprio dal lavoro di Christopher Alexander. Uno dei pattern da lui proposti, noto con il nome di "Corridoio corto", illustra chiaramente l'approccio metodologico alla base del concetto di pattern e ne evidenzia la capacità di risolvere problemi ricorrenti attraverso soluzioni collaudate e adattabili.

Il nome stesso del pattern, “Corridoio corto”, ne riassume l’essenza: si tratta di una soluzione architettonica pensata per affrontare l’effetto negativo che i corridoi lunghi, diritti e monotoni hanno sul benessere psicologico delle persone. Il contesto in cui questo pattern si colloca è quello dell’architettura moderna, in particolare degli edifici pubblici e ospedalieri, dove l’esigenza di razionalizzare gli spazi ha spesso condotto alla costruzione di corridoi lunghi e impersonali. Alexander descrive questi ambienti come uno degli esempi peggiori di progettazione, evidenziando come essi siano spesso bui, angusti e alienanti, privi di qualsiasi elemento estetico o funzionale che possa renderli accoglienti o umanizzati.

Il problema è analizzato da Alexander non solo in termini architettonici ma anche psicologici e sociali. Egli osserva come corridoi troppo lunghi – oltre i 16 o 17 metri – vengano percepiti come luoghi di disagio, soprattutto in contesti delicati come quello ospedaliero. In questi ambienti, dove i pazienti sono spesso vulnerabili e in cerca di rassicurazione, il design degli spazi assume un ruolo fondamentale. I corridoi stretti e infiniti con porte chiuse e poca luce contribuiscono a generare ansia e disorientamento. Alexander supporta la sua tesi citando studi clinici e osservazioni comportamentali, che evidenziano l’effetto negativo di tali ambienti sulla percezione e sullo stato emotivo delle persone.

Questo esempio dimostra come un pattern non sia semplicemente una ricetta tecnica, ma piuttosto una risposta profonda a un bisogno umano ricorrente, identificato attraverso l’esperienza e formalizzato in una proposta progettuale. L’importanza di dare un nome al pattern – “Corridoio corto” – sta

proprio nella possibilità di riconoscerlo, condividerlo e applicarlo in futuro in contesti analoghi, migliorando così la qualità delle soluzioni progettuali in modo sistematico.

Nel passaggio successivo, Alexander fornirà la soluzione al problema, illustrando come progettare corridoi che evitino gli effetti negativi descritti. Questo sarà il contenuto della prossima sezione, dove emergerà chiaramente la forza risolutiva dei pattern e la loro capacità di trasformare un'esperienza negativa in una linea guida per il miglioramento. Il valore di questo esempio risiede anche nella sua portabilità concettuale: sebbene legato al mondo dell'architettura, il pattern può essere considerato un archetipo anche per la progettazione software, dove ambienti ostili e strutture complesse possono essere "umanizzati" tramite l'adozione di soluzioni semplici, efficaci e orientate all'esperienza dell'utente.

La soluzione proposta da Christopher Alexander per affrontare il problema dei corridoi lunghi e alienanti è tanto semplice quanto profondamente efficace. Consiste nel ridurre la lunghezza dei corridoi, ma soprattutto nel trasformarne la percezione, rendendoli quanto più possibile simili a spazi abitati e accoglienti, quasi fossero stanze attraversabili piuttosto che semplici passaggi funzionali.

L'idea di fondo è quella di rompere la monotonia e l'anonimato del corridoio, conferendogli carattere, calore e varietà visiva. Questo può essere ottenuto tramite una serie di accorgimenti progettuali, come l'uso di pavimenti in legno o tappeti, che contribuiscono a una percezione più domestica e rilassante dello spazio. L'aggiunta di elementi di arredo, come librerie, quadri, specchi o piccoli mobili, aiuta a personalizzare l'ambiente, rendendolo meno asettico e più familiare. In questo modo, il corridoio cessa di essere un semplice luogo di transito per diventare parte integrante e vissuta dello spazio architettonico.

Un altro elemento fondamentale è la luce, non solo artificiale ma soprattutto naturale. Alexander sottolinea l'importanza di dotare i corridoi di ampie finestre, preferibilmente disposte lungo un'intera parete, in modo da garantire abbondante illuminazione e un collegamento visivo con l'esterno. La presenza della luce naturale ha un effetto immediato sul benessere psicologico delle persone, riducendo il senso di chiusura e migliorando l'umore, soprattutto in ambienti come ospedali, scuole o uffici.

Questa soluzione, sebbene pensata per il contesto architettonico, contiene principi universali applicabili anche al mondo del software engineering. Nel design di sistemi informatici, spesso si ricade in "corridori lunghi": strutture di codice troppo lineari, ridondanti, monotone o difficili da mantenere. Anche qui, il ricorso a pattern ben progettati permette di suddividere la complessità, aggiungere modularità, migliorare l'esperienza degli sviluppatori e degli utenti. Un pattern di design nel software, come il "Corridoio corto" nell'architettura, rappresenta dunque una forma concreta di saggezza progettuale riutilizzabile, che permette di affrontare problemi complessi in modo elegante, con soluzioni che tengano conto sia della funzionalità sia della qualità complessiva dell'esperienza.

Questo esempio evidenzia inoltre l'essenza stessa di un pattern: non una formula rigida, ma un modello flessibile, un nucleo di soluzione che può essere adattato, reinterpretato e declinato a seconda del contesto. È proprio questa capacità di trasformarsi pur mantenendo la propria identità logica a rendere i pattern strumenti così potenti, sia nell'architettura degli spazi reali che nella progettazione di sistemi informatici.

Ogni pattern di design è descritto seguendo una struttura ben definita, pensata per essere facilmente compresa, condivisa e riutilizzata in diversi contesti progettuali. Questa struttura consente ai progettisti di identificare rapidamente la natura del pattern, i problemi che risolve, e le implicazioni della sua applicazione. I principali elementi che compongono la descrizione di un pattern sono:

- **Nome.** È l'etichetta identificativa del pattern. Deve essere semplice, evocativo e facilmente memorizzabile, poiché viene utilizzato nelle discussioni tecniche per fare riferimento a un'intera soluzione progettuale. Il nome del pattern rappresenta in sintesi il problema risolto e diventa parte del vocabolario comune dei progettisti. Ad esempio, nomi come Observer, Strategy, Factory Method evocano immediatamente il tipo di meccanismo che il pattern implementa.

- **Problema.** Definisce il contesto in cui il pattern è applicabile, specificando le condizioni e i vincoli che giustificano l'uso di quella particolare soluzione. In questa sezione viene descritto il problema ricorrente che ha spinto alla formulazione del pattern, spesso accompagnato da esempi reali o scenari d'uso. Il riconoscimento del problema è fondamentale per scegliere consapevolmente quale pattern applicare.
- **Soluzione.** Fornisce una descrizione astratta della struttura progettuale proposta dal pattern. Si tratta di una combinazione di classi, oggetti, responsabilità e relazioni tra elementi, senza legarsi a un linguaggio di programmazione specifico. La soluzione non è una ricetta di codice già pronto, ma piuttosto un'architettura generale da modellare secondo le esigenze del sistema. Può includere diagrammi UML, relazioni tra classi (associazioni, aggregazioni, composizioni), e principi di interazione tra oggetti.
- Conseguenze. Analizza gli effetti derivanti dall'applicazione del pattern, sia in termini positivi (vantaggi) che di costi o vincoli. Tra gli aspetti considerati ci sono:
    - Il miglioramento della manutenibilità, flessibilità e riusabilità del codice.
    - I possibili sovraccarichi computazionali o aumento della complessità.
    - I vincoli temporali o legati al linguaggio di programmazione scelto.
    - Le implicazioni sull'estendibilità futura del sistema, cioè quanto facilmente sarà possibile aggiungere nuove funzionalità o modificare quelle esistenti.

Questa struttura sistematica consente non solo di documentare efficacemente i pattern, ma anche di confrontarli tra loro e di valutarne l'adeguatezza in relazione ai requisiti specifici di un progetto. Utilizzare un pattern in modo consapevole significa saper riconoscere il problema, selezionare il pattern giusto, adattarne la soluzione al contesto e comprendere le implicazioni della scelta effettuata.

Un primo esempio utile per comprendere l'importanza dei design pattern è rappresentato dal pattern Iterator. Questo pattern consente di accedere sequenzialmente agli elementi di una collezione senza esporne la struttura interna. È largamente adottato nei linguaggi orientati agli oggetti, come Java, e rappresenta un caso classico in cui l'astrazione consente di separare l'accesso ai dati dalla loro rappresentazione.

In Java, ad esempio, è possibile utilizzare l'interfaccia ListIterator per scorrere gli elementi di una lista in modo sicuro e indipendente dalla sua implementazione concreta. Un uso tipico prevede la creazione di un iteratore tramite list.listIterator() e l'uso del metodo hasNext() per verificare la presenza di un elemento successivo, seguito da next() per accedere all'elemento. Questo approccio consente di eseguire operazioni sugli elementi della lista in modo ordinato, senza conoscere se la lista sia implementata come array, lista concatenata, vettore o altro.

```
ListIterator iterator = list.listIterator();
while (iterator.hasNext()) {
Object current = iterator.next();
…
}
```

In contrapposizione, un approccio tradizionale e meno astratto comporterebbe l'accesso diretto ai collegamenti interni della struttura dati. Nel caso di una lista concatenata, questo significa inizializzare un riferimento al primo nodo della lista (tipicamente head) e scorrerla manualmente accedendo al campo next di ciascun nodo per muoversi al nodo successivo. A ogni iterazione, si estrae il valore del nodo corrente (di solito dal campo data o value) e si aggiorna il riferimento al nodo successivo.

```
Link currentLink = list.head;
while (currentLink != null) {
Object current = currentLink.Data;
currentLink = currentLink.next;
…
}
```

Questo metodo, sebbene funzionante, presenta una serie di criticità: innanzitutto, viola il principio di incapsulamento, poiché espone dettagli interni della struttura dati che dovrebbero rimanere nascosti. Inoltre, rende il codice più fragile, perché eventuali modifiche alla struttura della lista (ad esempio, cambiamenti nei nomi dei campi o nella modalità di collegamento tra i nodi) richiederebbero modifiche puntuali in ogni porzione di codice che accede direttamente ai nodi. Infine, aumenta il rischio di errori come l'interruzione accidentale della lista o la creazione di cicli infiniti, rendendo più complessa la manutenzione del software.

L'introduzione del pattern Iterator rappresenta quindi una soluzione elegante ed efficace, che migliora la qualità del codice, ne favorisce il riuso e lo rende più facilmente testabile e manutenibile. In particolare, l'incapsulamento delle regole di iterazione all'interno di un oggetto iteratore consente anche la definizione di percorsi di visita personalizzati (ad esempio, iterazioni in ordine inverso o filtrate), senza dover modificare la struttura dati sottostante. Questo dimostra come i pattern possano offrire soluzioni standard ma flessibili a problemi comuni della progettazione software.

Lo schema di classificazione e descrizione dei pattern proposto da Gamma et al. (nel celebre testo Design Patterns) prevede una serie di voci standard che aiutano a comprendere, confrontare e applicare correttamente ciascun pattern. Le voci principali sono:

- **Nome del pattern e classificazione:** identifica l'essenza del pattern e lo colloca all'interno di una tipologia (creazionale, strutturale, comportamentale).
- **Intento:** spiega in sintesi cosa fa il pattern e qual è il principio logico alla base della sua efficacia.
- **Altri nomi del pattern:** eventuali sinonimi o nomi alternativi con cui il pattern è conosciuto nella letteratura o nella pratica.
- **Motivazione:** presenta uno scenario concreto che mostra il problema che il pattern risolve e come la sua applicazione porta benefici.
- **Applicabilità:** descrive le condizioni e i contesti in cui è opportuno usare il pattern.
- **Struttura:** rappresentazione grafica (tipicamente in UML) delle classi coinvolte e delle loro relazioni, utile per visualizzare la soluzione proposta.
- **Partecipanti:** elenco delle classi e degli oggetti che partecipano all'implementazione del pattern, con indicazione delle rispettive responsabilità.
- **Collaborazioni:** descrizione di come i partecipanti interagiscono tra loro per soddisfare le responsabilità definite e realizzare il comportamento complessivo del pattern.
- **Conseguenze:** questa sezione illustra come il pattern raggiunge gli obiettivi prefissati, evidenziando vantaggi, svantaggi ed eventuali effetti collaterali derivanti dal suo utilizzo. Qui si analizzano anche quali parti del pattern possono essere adattate o modificate liberamente senza comprometterne la validità.
- **Implementazione:** fornisce suggerimenti pratici e tecniche per realizzare il pattern nel codice, affrontando inoltre eventuali problematiche specifiche legate al linguaggio di programmazione utilizzato. Questo aiuta a evitare errori comuni e a sfruttare al meglio le caratteristiche del linguaggio.
- **Codice d'esempio:** presenta frammenti di codice che illustrano concretamente come implementare il pattern, facilitando la comprensione e la sua applicazione diretta nei progetti.
- **Usi conosciuti:** vengono indicati esempi reali di applicazione del pattern in sistemi software noti, dimostrando la sua efficacia e diffusione nel mondo reale.
- **Pattern correlati:** descrive le differenze rispetto ad altri pattern simili e suggerisce con quali altri pattern è opportuno combinarlo per ottenere soluzioni più complete ed efficaci.

2. Catalogo dei pattern

I pattern di design rappresentano soluzioni collaudate a problemi comuni che emergono durante la progettazione software. Il loro studio e utilizzo facilitano la scrittura di codice più robusto, riusabile e manutenibile, fornendo un linguaggio comune tra i progettisti. Di seguito viene presentato il catalogo dei principali pattern di design, in ordine alfabetico, con una breve descrizione delle loro caratteristiche e applicazioni più tipiche.

- **Abstract Factory:** fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le classi concrete. È utile quando un sistema deve essere indipendente dalla creazione, composizione e rappresentazione degli oggetti che utilizza, permettendo di sostituire facilmente l'intera famiglia di prodotti.
- **Adapter:** converte l'interfaccia di una classe in un'altra interfaccia richiesta dal client, consentendo la collaborazione tra classi con interfacce incompatibili. È ideale per integrare componenti legacy o sistemi esterni senza modificarli.
- **Bridge:** disaccoppia un'astrazione dalla sua implementazione, così che entrambe possano variare indipendentemente. Utile quando una gerarchia di classi deve essere estesa in due dimensioni indipendenti (ad esempio, interfaccia e implementazione).
- **Builder:** separa la costruzione di un oggetto complesso dalla sua rappresentazione, permettendo di creare diverse rappresentazioni dello stesso prodotto. È particolarmente indicato per la creazione di oggetti con molti parametri o con configurazioni complesse.
- **Chain of Responsibility:** permette di evitare l'accoppiamento diretto tra mittente e destinatario di una richiesta, passando la richiesta lungo una catena di oggetti che possono decidere se gestirla o passarla oltre. È efficace per implementare sistemi di gestione eventi o filtri dinamici.
- **Command:** incapsula una richiesta come un oggetto, permettendo di parametrizzare client con richieste diverse, di supportare operazioni come undo, logging o bufferizzazione delle richieste. Facilita la separazione tra emittente e esecutore di comandi.
- **Composite:** consente di comporre oggetti in strutture ad albero per rappresentare gerarchie "parte-tutto", permettendo di trattare in modo uniforme singoli oggetti e composizioni di oggetti. Utile per implementare strutture gerarchiche come GUI o documenti complessi.
- **Decorator:** aggiunge responsabilità addizionali a un oggetto in modo dinamico, offrendo un'alternativa flessibile all'ereditarietà per estendere le funzionalità. È adatto per arricchire oggetti senza modificare il loro codice.

- **Facade:** fornisce un'interfaccia unificata e semplificata a un insieme di interfacce di un sottosistema complesso, facilitandone l'uso e riducendo le dipendenze tra i client e il sottosistema.
- **Factory Method:** definisce un'interfaccia per la creazione di un oggetto, delegando alle sottoclassi la decisione su quale classe concreta istanziare. Aiuta a seguire il principio di responsabilità unica e a rendere il codice più estensibile.
- **Flyweight:** utilizza la condivisione per gestire un gran numero di oggetti a granularità fine in modo efficiente, riducendo l'uso di memoria condividendo dati immutabili tra istanze. Utile per sistemi con molti oggetti simili, come editor grafici o videogiochi.
- **Interpreter:** definisce una grammatica per un linguaggio e un interprete che valuta le espressioni del linguaggio. Spesso utilizzato per implementare linguaggi di scripting o parser di linguaggi specifici.
- **Iterator:** fornisce un modo standard per accedere sequenzialmente agli elementi di una struttura dati aggregata senza esporne la rappresentazione interna. Facilita l'iterazione in modo sicuro e indipendente dalla struttura dati concreta.
- **Mediator:** centralizza la comunicazione tra un gruppo di oggetti, riducendo l'accoppiamento diretto tra essi e permettendo di cambiare le modalità di collaborazione senza modificare le classi partecipanti. È utile in sistemi complessi con molte interazioni.
- **Memento:** consente di catturare e salvare lo stato interno di un oggetto senza violarne l'incapsulamento, permettendo di ripristinare tale stato in un secondo momento. Spesso usato per implementare funzionalità di undo/redo.
- **Observer:** definisce una relazione di dipendenza uno-a-molti tra oggetti, in modo che quando un oggetto cambia stato, tutti i suoi osservatori vengano notificati e aggiornati automaticamente. Fondamentale nei sistemi event-driven e nelle interfacce utente.
- **Prototype:** specifica come creare nuovi oggetti copiando un'istanza prototipale, evitando di dover creare nuove istanze da zero. Utile quando la creazione di un oggetto è costosa o complessa.
- **Proxy:** fornisce un surrogato o rappresentante di un altro oggetto per controllare l'accesso ad esso, aggiungendo funzionalità come controllo degli accessi, caching o caricamento ritardato (lazy loading).
- **Singleton:** garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a essa. Spesso usato per gestire risorse condivise o configurazioni globali.
- **State:** permette a un oggetto di modificare il proprio comportamento quando il suo stato interno cambia, dando l'illusione che l'oggetto cambi classe. Favorisce un design più pulito rispetto a lunghe strutture condizionali.

- **Strategy:** definisce una famiglia di algoritmi intercambiabili incapsulati in classi separate, permettendo di variare l'algoritmo usato da un client senza modificarne il codice.
- **Template Method:** definisce lo scheletro di un algoritmo in un metodo, lasciando che alcune fasi siano implementate dalle sottoclassi. Promuove il riuso del codice e la specializzazione controllata.
- **Visitor:** consente di definire nuove operazioni da eseguire su una struttura di oggetti senza modificarne le classi, separando l'algoritmo dalla struttura dati. Utile per estendere funzionalità senza alterare il modello di dati.

Alcuni pattern si ritrovano con grande frequenza nei progetti software a oggetti perché rispondono a esigenze comuni e fondamentali di progettazione, che emergono spesso nei requisiti reali di sistemi complessi, scalabili e manutenibili. Ecco le principali ragioni per cui questi pattern sono così ampiamente adottati:

- **Abstract Factory:** spesso usato in sistemi che devono supportare più famiglie di prodotti o piattaforme (es. GUI multi-piattaforma, sistemi embedded), permette di isolare la creazione degli oggetti dall'implementazione concreta, facilitando la portabilità e la sostituibilità dei componenti.
- **Adapter:** fondamentale in progetti che integrano componenti legacy o librerie esterne con interfacce incompatibili, l'adapter consente di "incollare" sistemi diversi senza dover modificare il codice originale, favorendo la riusabilità e l'interoperabilità.
- **Composite:** molto usato in applicazioni con strutture gerarchiche o ricorsive (es. interfacce grafiche, sistemi di file, documenti strutturati), il composite permette di trattare in modo uniforme singoli oggetti e aggregati, semplificando la gestione e l'estensione delle gerarchie.
- **Decorator:** adottato quando è necessario estendere funzionalità in modo flessibile e dinamico, senza ricorrere all'ereditarietà statica. Questo è frequente in UI, flussi di elaborazione dati, e sistemi di personalizzazione, dove le responsabilità possono cambiare a runtime.
- **Factory Method:** utile in progetti con gerarchie di classi che differiscono nella creazione degli oggetti, favorisce l'estensione delle classi senza modificare il codice cliente, semplificando l'aggiunta di nuove varianti di prodotto.
- **Observer:** essenziale nei sistemi con architettura event-driven o basati su eventi e notifiche (es. GUI, sistemi distribuiti, motori di regole), consente una comunicazione efficiente e disaccoppiata tra oggetti, migliorando la modularità e la scalabilità.
- **Strategy:** adottato quando è necessario variare dinamicamente algoritmi o comportamenti (es. ordinamenti, validazioni, calcoli), il pattern permette di incapsulare le alternative e cambiarle senza modificare il contesto.

- **Template Method:** usato nei casi in cui si vuole definire un algoritmo generale con passi modificabili dalle sottoclassi, favorisce il riuso del codice e la specializzazione controllata, comune in librerie e framework.

È difficile trovare un sistema a oggetti che non impieghi almeno un paio di questi pattern, tanto sono diventati fondamentali per una progettazione modulare, manutenibile ed estendibile.

3. Organizzazione del catalogo

Il catalogo dei pattern di design contiene numerosi modelli, ognuno con caratteristiche e ambiti di applicazione differenti. Per facilitarne lo studio, la memorizzazione e l'applicazione pratica, è fondamentale organizzare i pattern in modo strutturato. L'organizzazione aiuta a riconoscere somiglianze, differenze e relazioni tra i pattern, oltre a favorire la scelta del modello più adatto in base al contesto progettuale.

Esistono diverse modalità per classificare i pattern, che si basano su criteri differenti, ognuno utile per un aspetto specifico della progettazione:

- **Per granularità e livello di astrazione.** Si distinguono pattern che agiscono a livelli diversi, dal più generale e concettuale al più specifico e concreto. Questo aiuta a capire se un pattern fornisce una struttura complessiva o se risolve dettagli implementativi puntuali.
- **Per gruppi di pattern logicamente correlati.** Alcuni pattern condividono concetti o meccanismi simili, e raggrupparli secondo tali affinità facilita l'apprendimento e la comprensione delle alternative e delle varianti disponibili.
- **Per gruppi di pattern usati insieme.** In molti casi, i pattern non sono usati singolarmente ma combinati per ottenere soluzioni più complete e robuste. Identificare queste combinazioni frequenti aiuta a progettare sistemi integrati e coerenti.
- **Per scopo e raggio d'azione.** Questa classificazione tiene conto della funzione principale di ciascun pattern, ad esempio se è orientato alla creazione di oggetti, alla gestione delle responsabilità, o al controllo delle comunicazioni tra oggetti.

Una delle principali modalità di organizzazione riguarda lo scopo dei pattern, cioè il tipo di problema che essi vogliono risolvere.

I pattern possono essere suddivisi in tre grandi famiglie: creazionali, strutturali e comportamentali. I pattern creazionali si concentrano sul processo di creazione degli oggetti, cercando di rendere più flessibile e modulare il modo in cui gli oggetti vengono istanziati. I pattern strutturali, invece, riguardano la composizione di classi e oggetti, cioè come combinare componenti più semplici per costruire strutture più complesse e funzionali. Infine, i pattern comportamentali descrivono le modalità di collaborazione e comunicazione tra oggetti o classi, definendo responsabilità e flussi di controllo che permettono di organizzare in modo efficiente il comportamento di sistemi complessi.

Un'altra importante classificazione si basa sul raggio d'azione del pattern, cioè a quale livello agisce all'interno del sistema. Si distingue tra pattern che operano sulle classi e pattern che operano sugli oggetti.

Nel caso dei pattern creazionali, si possono avere quelli che delegano la creazione degli oggetti alle sottoclassi, sfruttando quindi l'ereditarietà per determinare come un oggetto viene creato, oppure quelli che affidano la creazione ad altri oggetti, privilegiando la composizione e il riuso piuttosto che l'ereditarietà.

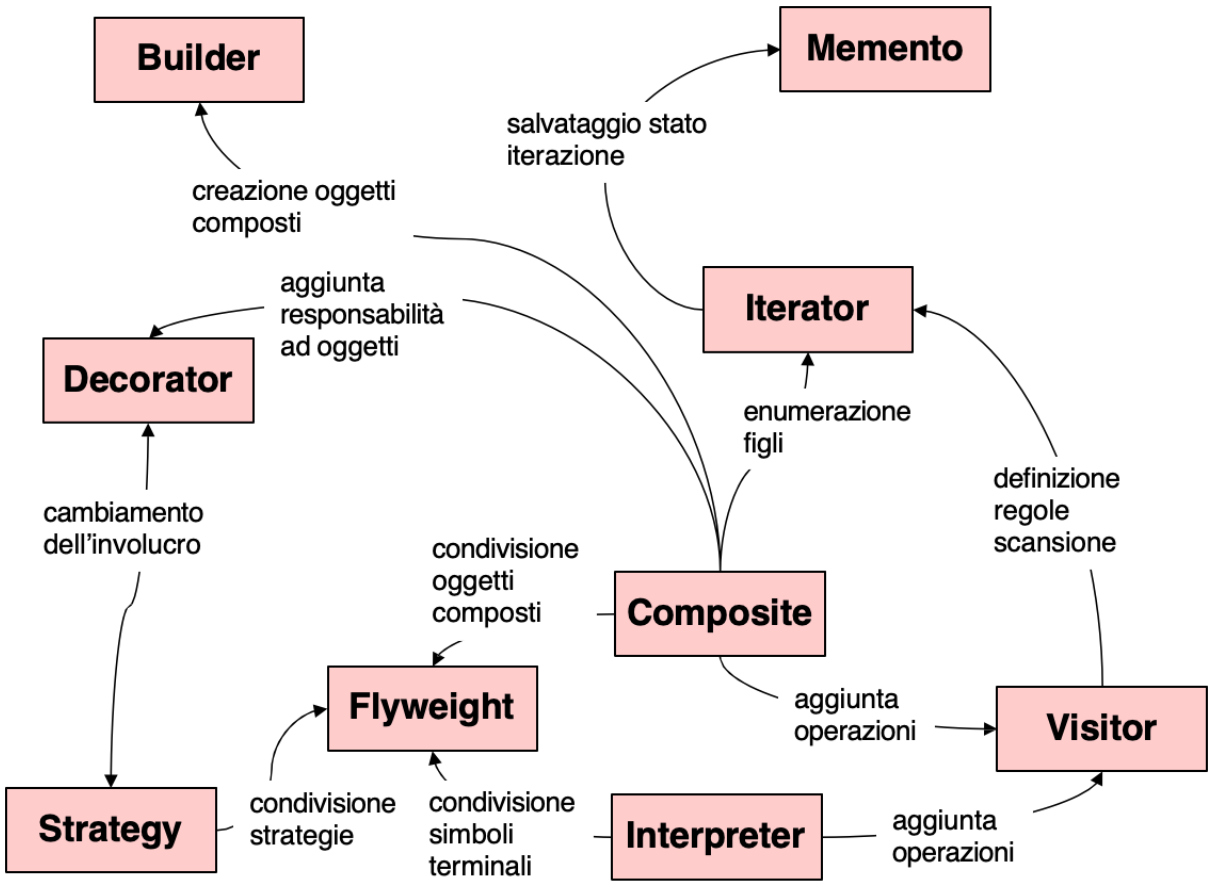
Per i pattern strutturali, la distinzione è tra quelli che usano l'ereditarietà per comporre nuove classi a partire da quelle esistenti e quelli che invece definiscono modi per raggruppare oggetti, mantenendo separate le interfacce dalle implementazioni e facilitando così l'estensibilità del sistema.

Anche nei pattern comportamentali troviamo due categorie: i pattern che utilizzano l'ereditarietà per descrivere algoritmi e flussi di controllo nelle gerarchie di classi, e quelli che definiscono collaborazioni tra gruppi di oggetti per gestire attività che un singolo oggetto non potrebbe svolgere da solo, aumentando la flessibilità e la modularità del sistema.

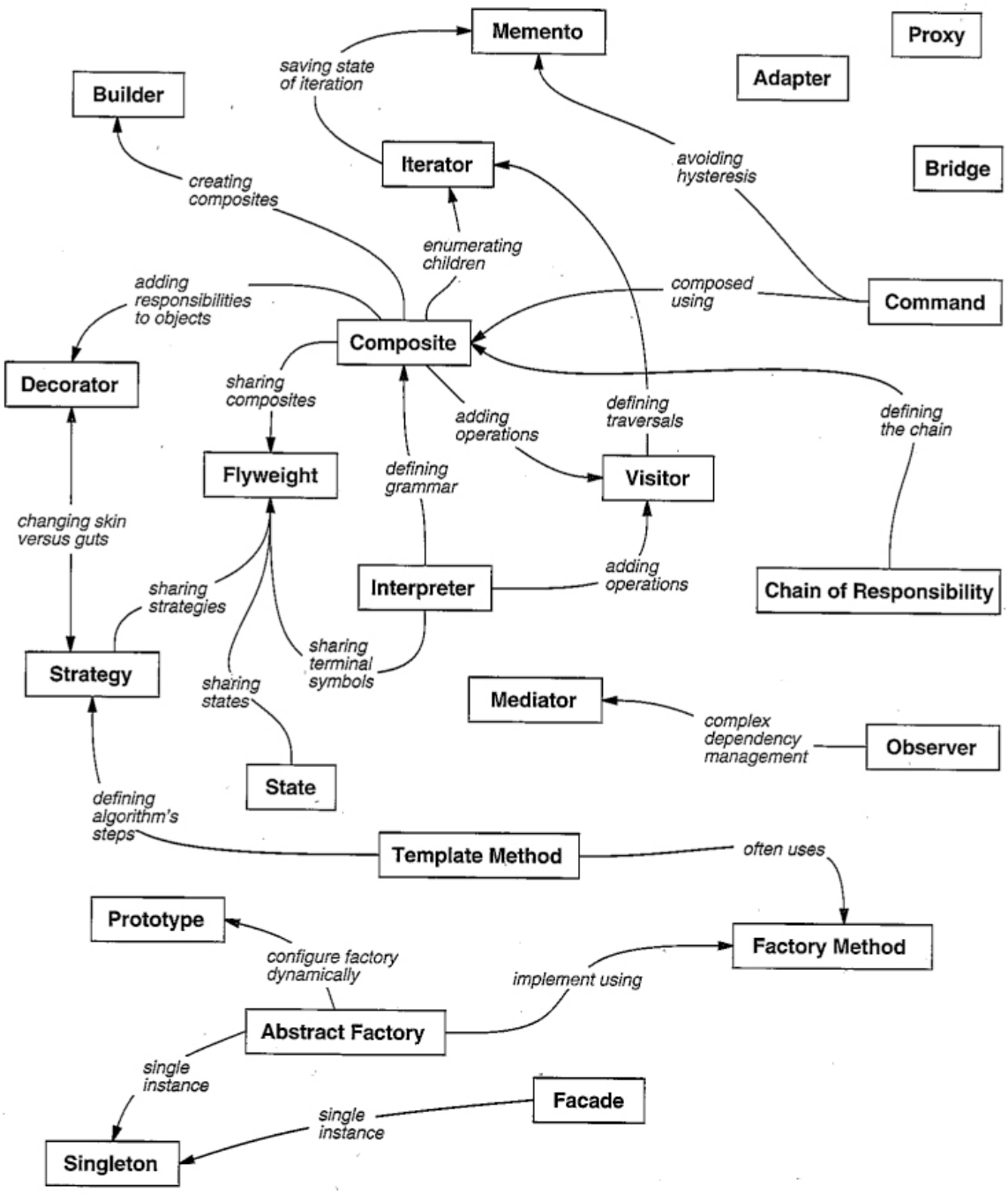
Raggio d'azione	**Creazionale**	**Strutturale**	**Comportamentale**
Classi	Factory Method	Adapter (class)	Interpreter, Template Method
Oggetti	Abstract Factory	Adapter (object)	Chain of Responsibility
	Builder	Bridge	Command
	Prototype	Composite	Iterator
	Singleton	Decorator	Mediator
		Facade	Memento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

Questa duplice organizzazione, basata su scopo e raggio d'azione, aiuta a orientarsi all'interno del catalogo e a selezionare il pattern più adatto a seconda del problema da risolvere e del contesto di sviluppo. Comprendere queste classificazioni permette di cogliere più facilmente le caratteristiche e i vantaggi di ciascun pattern, facilitandone l'applicazione pratica.

Si riporta adesso una porzione della figura 1.1 del testo Gamma et al. con alcune delle relazioni tra pattern.



Di seguito infine si riporta la mappa completa dei pattern.



Design Pattern Relationships

Bibliografia

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2002). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Alexander, C. (1977). A Pattern Language: Towns, Buildings, Construction. Oxford University Press.
- Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). Head First Design Patterns. O'Reilly Media.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern-Oriented Software Architecture: A System of Patterns. Wiley.
- Larman, C. (2004). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall.