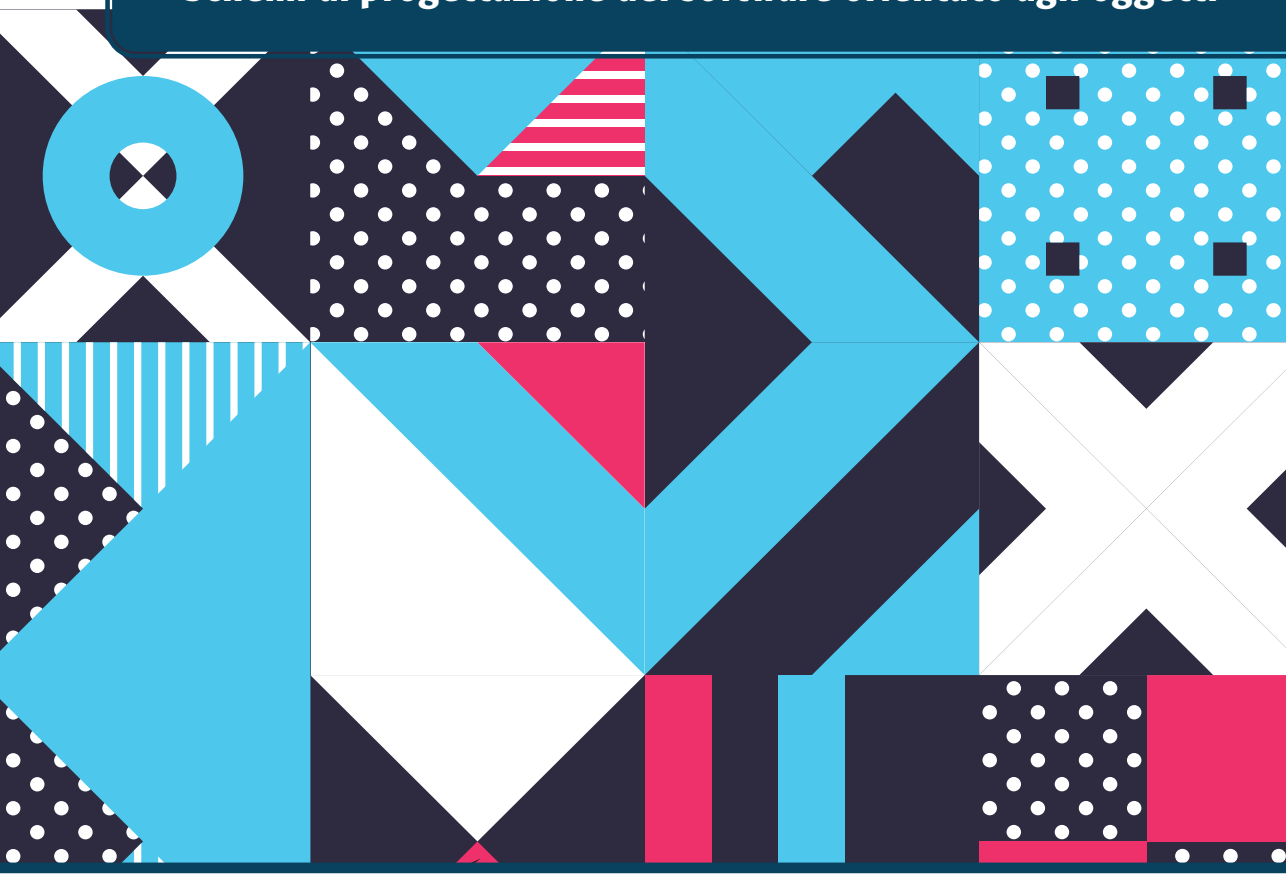
- Antonio Pelleriti -

Design patterns

Schemi di progettazione del software orientato agli oggetti



Introduzione ai design patterns e al software design >>

Patterns di creazione, strutturali e comportamentali >>

Esempi pratici in linguaggio C# >>

Diagrammi in formato UML >>

*pro DigitalLifeStyle

*pro
Digital Life Style

Design patterns

Schemi di progettazione del software orientato agli oggetti

Antonio Pelleriti

EDIZIONI
LSWR

Design patterns | Schemi di progettazione del software orientato agli oggetti.

Autore: Antonio Pelleriti

Collana: *pro DigitalLifeStyle

Publisher: Marco Aleotti
Progetto grafico: Roberta Venturieri
Immagine di copertina: © Vanzyst | Shutterstock

ISBN: 978-88-6895-884-8
eISBN: 978-88-6895-885-5

EDIZIONI LSWR
Via G. Spadolini, 7
20141 Milano (MI)
Tel. 02 881841
www.edizionilswr.it

Printed in Italy

Finito di stampare nel mese di settembre 2020 presso "Rotomail Italia" S.p.A., Vignate (MI)

(*) Edizioni LSWR è un marchio di La Tribuna Srl. La Tribuna Srl fa parte di LSWR GROUP.

Indice

INTRODUZIONE XIII
A chi si rivolge il libro XIV
Struttura del libro XIV
Esempi ed errata corrige XV
L'autore XVI

PARTE 1: SOFTWARE DESIGN E DESIGN PATTERN 1

1. I DESIGN PATTERN 3
Introduzione ai design pattern 4
Che cos'è un design pattern 5
Che cosa non è un design pattern 6
A che cosa servono i design pattern 6
Classificazione dei design pattern 7
Scelta dei design pattern 13

2. PROGRAMMAZIONE ORIENTATA AGLI OGGETTI 15
Fondamenti di OOP 16
Oggetti e classi 17
Gerarchie di classi 21
Interfacce 22
Classi astratte 24
I principi fondamentali di OOP 25

3. PRINCIPI DI SOFTWARE DESIGN 33
Code Reuse 34
Program to an Interface 35
Composition over Inheritance 36
Encapsulate what Varies 37
Loose Coupling 38
High Cohesion 39
Principi SOLID 39

PARTE 2: PATTERN CREAZIONALI 51

4. ABSTRACT FACTORY 53
Motivazioni di utilizzo 53
Struttura 54
Implementazione 55
Applicabilità 60
Conseguenze 60
Pattern correlati 61

5. BUILDER 63
Motivazioni di utilizzo 63
Struttura 64
Implementazione 66
Applicabilità 71
Conseguenze 71
Pattern correlati 71

6. FACTORY METHOD 73
Motivazioni di utilizzo 73
Struttura 74
Implementazione 75
Applicabilità 81
Conseguenze 81
Pattern correlati 82

7. PROTOTYPE 83
Motivazioni di utilizzo 83
Struttura 84
Implementazione 85
Applicabilità 94
Conseguenze 95
Pattern correlati 95

8. SINGLETON 97
Motivazioni di utilizzo 97
Struttura 98
Implementazione 99
Conseguenze 100
Pattern correlati 100

PARTE 3: PATTERN STRUTTURALI

9. ADAPTER 103
Motivazioni di utilizzo 103
Struttura 104
Implementazione 107
Applicabilità 112
Conseguenze 113
Pattern correlati 113

10. BRIDGE 115
Motivazioni di utilizzo 115
Struttura 119
Implementazione 121
Applicabilità 123
Conseguenze 124
Pattern correlati 124

11. COMPOSITE 125
Motivazioni di utilizzo 125
Struttura 126
Implementazione 127
Applicabilità 132
Conseguenze 132
Pattern correlati 132

12. DECORATOR 133
Motivazioni di utilizzo 133
Struttura 134
Implementazione 136
Applicabilità 142
Conseguenze 142
Pattern correlati 143

13. FAÇADE 145
Motivazioni di utilizzo 145
Struttura 146
Implementazione 149
Applicabilità 153
Conseguenze 154
Pattern correlati 154

14. FLYWEIGHT 155
Motivazioni di utilizzo 155
Struttura 156

Implementazione 157
Applicabilità 162
Conseguenze 163
Pattern correlati 164

15. PROXY 165
Motivazioni di utilizzo 165
Struttura 166
Implementazione 168
Applicabilità 176
Conseguenze 177
Pattern correlati 177

PARTE 4: PATTERN COMPORTAMENTALI 179

16. CHAIN OF RESPONSIBILITY 181
Motivazioni di utilizzo 181
Struttura 182
Implementazione 184
Applicabilità 189
Conseguenze 189
Pattern correlati 189

17. COMMAND 191
Motivazioni di utilizzo 191
Struttura 192
Implementazione 194
Applicabilità 199
Conseguenze 199
Pattern correlati 199

18. INTERPRETER 201
Motivazioni di utilizzo 201
Struttura 202
Implementazione 203
Applicabilità 209
Conseguenze 209
Pattern correlati 209

19. ITERATOR 211
Motivazioni di utilizzo 211
Struttura 212
Implementazione 214
Applicabilità 219

Conseguenze 220
Pattern correlati 221

20. MEDIATOR 223
Motivazioni di utilizzo 223
Struttura 224
Implementazione 226
Applicabilità 229
Conseguenze 229
Pattern correlati 230

21. MEMENTO 231
Motivazioni di utilizzo 231
Struttura 232
Implementazione 234
Applicabilità 238
Conseguenze 239
Pattern correlati 239

22. OBSERVER 241
Motivazioni di utilizzo 241
Struttura 242
Implementazione 244
Applicabilità 248
Conseguenze 249
Pattern correlati 250

23. STATE 251
Motivazioni di utilizzo 251
Struttura 252
Implementazione 254
Applicabilità 258
Conseguenze 258
Pattern correlati 259

24. STRATEGY 261
Motivazioni di utilizzo 261
Struttura 262
Implementazione 263
Applicabilità 266
Conseguenze 266
Pattern correlati 268

25. TEMPLATE METHOD 269
Motivazioni di utilizzo 269

Struttura 270
Implementazione 272
Applicabilità 275
Conseguenze 276
Pattern correlati 277

26. VISITOR 279
Motivazioni di utilizzo 279
Struttura 280
Implementazione 283
Applicabilità 287
Conseguenze 287
Pattern correlati 288

APPENDICI

A. INTRODUZIONE AL LINGUAGGIO UML 289
Introduzione a UML 290
Diagrammi UML 291
Diagrammi delle classi 292
Diagrammi di sequenza 302

B. ANTIPATTERN 309

INDICE ANALITICO 315

A mia figlia Matilda, il mio unico capolavoro

Introduzione

Una delle espressioni maggiormente utilizzate e che sarebbe meglio tenere a mente (anche) nel mondo dello sviluppo software è **don't reinvent the wheel** (la riporto in inglese perché deriva dal mondo anglosassone, ma possiamo tranquillamente tradurla senza perderne il significato: non reinventare la ruota).
Il suo significato è che, in alcuni casi, soluzioni tecniche universalmente accettate e funzionanti vengono stranamente o volontariamente ignorate a favore di altre soluzioni progettate e sviluppate da zero, sprecando tempo, lavoro e quindi denaro.
Evitare la duplicazione di "cose" esistenti e riutilizzarle permette un risparmio notevole di tempo di progettazione, di sviluppo, di test, che si traduce in efficienza a tutto tondo: meno costi, meno ritardi, rilasci puntuali, più guadagno!
I design pattern, argomento del testo, rappresentano uno dei metodi per evitare di reinventare la ruota, riutilizzando soluzioni ben note, corrette e collaudate.
In particolare un catalogo di 23 di questi design pattern è stato formalizzato, più di 20 anni fa, da un team di quattro esperti di progettazione software, noto come la Gang of Four. Questi pattern, nonostante nell'ambito informatico due decenni siano più di un'era geologica, sono ancora utilizzati e adottati come best practice fondamentali.
Essi d'altronde oltre che dall'esperienza degli autori e di altri esperti nascono dall'applicazione di altre tecniche, strategie, principi o comunque costituiscono soluzioni che rispettano concetti fondamentali di sviluppo e progettazione.
Se siete interessati ai design pattern, probabilmente conoscete già il paradigma di programmazione orientato agli oggetti; ma non basta saper creare una gerarchia di classi sfruttando l'ereditarietà, le interfacce, l'incapsulamento per poter affermare che si sta sviluppando a oggetti in maniera corretta.
Quindi nella prima parte, oltre a un capitolo sullo sviluppo object oriented, per rinfrescare o comunque introdurre anche i meno esperti all'argomento, ho previsto anche un capitolo dedicato ai principi di progettazione e sviluppo su cui si fondano sia il paradigma OOP sia i design pattern.

Come vedremo, ogni pattern si applica a una particolare categoria di problemi e fornisce la soluzione o l'approccio giusto per risolverli.
Naturalmente il miglior modo per utilizzare i design pattern, una volta studiati e assimilati (anche grazie a questo libro...), sarà quello di analizzare il problema, riconoscere a questo punto quali pattern permettono di risolverlo, e infine applicarli per implementare la soluzione.

A chi si rivolge il libro

Questo libro intende rivolgersi a lettori che abbiano un minimo di esperienza nella programmazione orientata agli oggetti e che desiderino migliorare il proprio approccio sia alla progettazione del software sia allo sviluppo vero e proprio.
Anche chi è alle prime armi comunque può trarre beneficio, in maniera da partire subito con il piede giusto e imparare ad accostarsi correttamente allo sviluppo.
Lo scopo del libro è affrontare con la maggior precisione e approfondimento possibile i concetti trattati, in modo tale da diventare un riferimento completo sui design pattern per programmatori di ogni livello e con ogni linguaggio. Infatti, anche se è stato utilizzato C# per gli esempi di implementazione, ho cercato di creare esempi che non facciano uso di caratteristiche esclusive di tale linguaggio, e che quindi siano comprensibili a colpo d'occhio anche a chi sviluppa in Java, C++ o qualunque altro linguaggio orientato agli oggetti.
Ogni capitolo dedicato ai design pattern contiene inoltre diverse figure con diagrammi UML, quindi presuppone che il lettore sia in grado di comprenderli almeno in minima parte. In ogni caso, l'Appendice A contiene una rapida introduzione a UML, che spiega i concetti utilizzati nel libro.

Struttura del libro

Ho scritto questo libro nel modo in cui mi piacerebbe leggere un libro sui design pattern, quindi spero che questo mio modo di pensare e strutturarlo incontri anche il gusto dei lettori, e soprattutto che possa raggiungere l'obiettivo di trattare e spiegare i design pattern in modo semplice ma allo stesso tempo preciso ed esauriente, anche a chi non ha mai sentito parlare dell'argomento, e di comprenderne ogni sfaccettatura e i concetti fondamentali su cui si basano e da cui derivano.
Il libro è suddiviso in quattro parti:

Parte 1 - Software design e design pattern

- **Capitolo 1 - I design pattern**: è un'introduzione ai design pattern, che spiega che cosa sono, che cosa non sono, a che cosa servono, e contiene poi una loro classificazione nelle tre categorie che costituiscono le successive tre parti del libro;

- **Capitolo 2 - Programmazione orientata agli oggetti**: introduzione al paradigma orientato agli oggetti, che ne spiega gli elementi fondamentali, necessari a capire e applicare i design pattern;
- **Capitolo 3 - Principi di software design**: espone i concetti fondamentali di software design, le best practice, le strategie e i principi su cui si basano i design pattern e che la loro corretta applicazione permette di rispettare.

Le parti successive sono dedicate alle tre categorie di design pattern.
Parte 2 - Pattern creazionali
Parte 3 - Pattern strutturali
Parte 4 - Pattern comportamentali

All'interno di ogni parte a ciascun pattern è dedicato un capitolo specifico, che si apre con l'enunciazione dell'intento di quel pattern, per passare poi a descrivere i problemi e le motivazioni che portano al suo utilizzo; ne viene mostrata poi la struttura mediante l'ausilio di diagrammi UML, con la descrizione dettagliata dei partecipanti a tale struttura (classi, interfacce e così via); si passa quindi a implementare un esempio pratico di applicazione del pattern. Due paragrafi evidenziano sia gli scenari di applicabilità del pattern, sia i benefici, ma anche eventuali controindicazioni che nascono dalla sua applicazione. Infine, per ogni pattern si elencano quelli correlati per somiglianza, per scenario di applicazione, o perché possono essere combinati.

L'**Appendice A - Introduzione al linguaggio UML** è un'introduzione allo Unified Modeling Language, in particolare ai diagrammi delle classi e ai diagrammi di sequenza, utilizzati nei capitoli dedicati ai design pattern per spiegarne struttura e funzionamento.
L'**Appendice B - Antipattern** è un breve riepilogo degli antipattern, cioè di quelle tecniche o pratiche che sembrano soluzioni corrette come i pattern, ma che alla fine non lo sono, anzi sono controproducenti o addirittura dannose.

Esempi ed errata corrige

Ogni capitolo del libro è dedicato a un singolo design pattern, contiene esempi pratici di implementazione che potete scrivere e compilare autonomamente, nel vostro linguaggio di programmazione abituale.
Per comodità comunque gli esempi in C# completi sono disponibili sul mio sito internet, alla pagina *https://antoniopelleriti.it/libro-design-patterns*.
Sebbene il libro sia stato letto e riletto, qualche errore può sempre sfuggire! Potrete trovare comunque eventuali correzioni sempre sul sito dedicato.

Per eventuali segnalazioni di errori e imprecisioni, e proporre magari suggerimenti per le prossime edizioni, scrivetemi o seguite la pagina ufficiale Facebook: *https://fb.me/libro.design.pattern*.

L'autore

Antonio Pelleriti è ingegnere informatico e si occupa da diversi anni di progettazione e sviluppo software, su varie piattaforme.
In particolare il .NET Framework e le tecnologie ad esso correlate sono i suoi principali interessi fin dal rilascio della prima beta della piattaforma Microsoft, quindi da oltre 15 anni. In tale ambito un importante riconoscimento è stata la nomina a **Microsoft MVP** per .NET e in seguito per Visual Studio and Development Technologies.
Autore di numerose pubblicazioni per riviste di programmazione e di guide tascabili dedicate al mondo .NET, per Edizioni FAG ha già pubblicato il libro *Silverlight 4, guida alla programmazione* e, per Edizioni LSWR, quattro diverse edizioni di *Programmare con C#, guida completa*.
Il suo sito personale su cui pubblica articoli e pillole di programmazione legate al mondo dello sviluppo software, in particolare a .NET e C#, è *www.antoniopelleriti.it*.

Parte 1

Software design e design pattern

1

I design pattern

Un design pattern rappresenta un problema comune di progettazione e implementazione, magari ripetitivo, e allo stesso tempo la soluzione e l'approccio con cui affrontarlo, che sarà quindi riutilizzabile più volte senza partire da zero.

Se siete sviluppatori o progettisti esperti, quante volte vi è capitato di sedere davanti al vostro computer, per iniziare la programmazione di un'applicazione, e pensare che, almeno in parte avete già affrontato lo stesso problema? Oppure quante volte, anche se il codice scritto funziona, vi rendete conto che c'è sicuramente un modo migliore, più efficace, più elegante per ottenere lo stesso risultato?
Il termine design pattern indica uno schema che identifica come progettare e implementare un'applicazione o una sua parte, e quindi gli oggetti che la compongono, le loro caratteristiche e le loro eventuali interazioni, in maniera indipendente da linguaggi di programmazione o da tecnologie.
Naturalmente non c'è nulla di pronto all'uso, ogni design pattern andrà valutato, scelto, personalizzato e adattato al proprio scenario, implementando poi l'applicazione della soluzione al problema iniziale.

Introduzione ai design pattern

La parola pattern è difficile da tradurre in italiano, o perlomeno ha diversi significati: modello, schema, esempio. Design pattern quindi indica uno schema o un modello architetturale o di progettazione.
Il termine fu introdotto dall'architetto (non software!) **Christopher Alexander** in un suo celebre saggio intitolato *A pattern Language* (1977), in cui egli dice che "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 riusare la soluzione un milione di volte, senza mai applicarla alla stessa maniera".
Per dirla breve, un design pattern è una soluzione progettuale a un problema ricorrente. Anche se il saggio di Alexander lo usa riferendosi all'architettura e alla costruzione di edifici e ponti, il concetto di pattern è applicabile anche all'ambiente dello sviluppo software, nel quale i design pattern sono stati introdotti invece nel 1995, da un gruppo di quattro esperti di progettazione software (Gamma, Helm, Johnson e Vlissides) noto come **Gang of Four** (spesso abbreviato in GoF), vale a dire la Banda dei Quattro.

NOTA

Nemmeno i membri della GoF sanno come si sia originato l'epiteto Gang of Four. Il termine è forse un'allusione al gruppo di uomini politici cinesi noto come banda dei quattro, e suggerisce un parallelo tra l'influenza esercitata da questi ultimi sulla storia cinese contemporanea e quella invece esercitata sull'ingegneria del software proprio dal concetto di design pattern.

Il frutto della loro esperienza è un catalogo di "soli" 23 design pattern, che presenteremo nel prosieguo del testo in maniera da poterli riutilizzare e applicare a problemi che si presentano tutti i giorni, avendo così a disposizione una soluzione pronta all'uso, elegante ed efficiente, nata proprio grazie all'esperienza di chi l'ha pensata e progettata prima di noi.
Nella progettazione software i design pattern sono inevitabilmente associati al paradigma *orientato agli oggetti*, facendo quindi uso, come si vedrà, di concetti tipici di tale paradigma, come oggetti, classi e interfacce, le relazioni e interazioni fra di essi, e le loro caratteristiche, come l'ereditarietà, il polimorfismo e così via. I concetti però sono generali, quindi applicabili eventualmente anche a ogni altra tipologia o paradigma di programmazione.
In effetti molti dei concetti fondamentali del paradigma Object Oriented possono essere considerati dei design pattern di per sé. Per esempio l'ereditarietà permette di implementare una gerarchia di classi, specializzando il comportamento di oggetti con

nuove funzionalità e caratteristiche da aggiungere a quelle definite in una o più classi madre. L'incapsulamento può essere considerato un pattern di progettazione con cui si può nascondere lo stato e il comportamento interno di un oggetto e utilizzare solo la sua interfaccia pubblica.
Probabilmente, l'introduzione dei design pattern rappresenta uno dei più grandi balzi in avanti mai fatti nell'ambito della progettazione e della programmazione orientata agli oggetti.

Che cos'è un design pattern

Ogni design pattern viene definito da almeno quattro elementi fondamentali.

- **Nome,** costituito da una o più parole che identifica e rappresenta il pattern stesso.
- **Problema,** che descrive la situazione alla quale il design pattern può essere applicato o in genere le motivazioni di un suo utilizzo.
- **Soluzione,** descritta da un insieme di classi e le interazioni fra di esse, rappresentate anche tramite diagrammi, che permetteranno di affrontare e risolvere il problema, senza scendere in dettagli di implementazione.
- **Conseguenze** e altri risultati che derivano dall'applicazione del design pattern e che quindi possono anche influenzare la scelta del pattern stesso o come implementarlo nel proprio linguaggio di programmazione.

Oltre a questi, o comunque per descriverne ancora meglio alcuni dettagli, per ogni design pattern potranno essere esposte e spiegate altre caratteristiche, che dipendono dal pattern in esame, per esempio quali sono i design pattern correlati, l'applicabilità a particolari situazioni o scenari, esempi di implementazione con codice sorgente (in questo testo sarà usato spesso C#, ma vedrete che tradurre nel vostro linguaggio preferito non sarà un problema, una volta assimilati i concetti del pattern e assumendo che siate abbastanza esperti nel linguaggio che avete scelto) e diagrammi UML.

> **NOTA**
>
> L'UML è un linguaggio grafico per la progettazione di software object oriented. Vedi l'appendice A per una panoramica rapida dei tipi di diagrammi utilizzati in questo testo, e in generale di quelli usati con i design pattern, in modo tale che, se siete proprio a digiuno di UML, possiate acquisire subito familiarità e orientarvi con i diagrammi.

Che cosa non è un design pattern

Dopo aver chiarito cos'è un design pattern, per togliere eventuali dubbi, possiamo anche dire che cosa non è un design pattern e che cosa non ci si deve aspettare.
Un design pattern non è un algoritmo che descrive i passi per risolvere un preciso problema, né un progetto di classi pronto all'uso, da implementare in maniera immediata nel proprio linguaggio a oggetti; per esempio non troverete un'implementazione di una struttura dati o cose del genere.
I design pattern non si riferiscono a specifici domini o ambiti applicativi e non rappresentano sottosistemi di intere applicazioni, pronti da integrare magari in un sistema esistente.
Uno stesso design pattern, applicato più volte a problemi o in scenari diversi, potrebbe portare a soluzioni e quindi a codice anche completamente differenti.

A che cosa servono i design pattern

Ogni pattern si applica a un particolare problema e fornisce la soluzione o l'approccio giusto per risolverlo, in quanto è stato già affrontato e risolto da altri sviluppatori e progettisti. Il modo migliore o più efficace per utilizzare i design pattern, una volta studiati e assimilati, sarà quello di analizzare il problema, riconoscere a questo punto quali pattern permettono di risolverlo, e infine applicarli per implementare la soluzione.
Qualsiasi attività, e quindi anche scrivere un programma in un qualsiasi linguaggio, è più facile se ci si organizza e se si sfruttano schemi e piani ben collaudati. Per lo stesso motivo sarà anche più facile eseguirne la manutenzione. L'utilizzo dei design pattern permette di sfruttare l'esperienza di diverse persone che hanno affrontato problematiche comuni e di sviluppare software in maniera elegante ed efficiente.
I design pattern sono un modo per **riutilizzare** codice e software già realizzati in precedenza, da noi o da altri, e per condividerlo così con altri programmatori e in altre applicazioni. In parole povere si può dire che essi forniscono la soluzione pronta all'uso per problemi simili ad altri problemi già affrontati e risolti in precedenza.
Probabilmente avrete risolto problemi, anche complessi, senza far uso di alcun design pattern, almeno non consapevolmente. Ma le vostre soluzioni forse non avranno lo stesso indice di efficienza, completezza ed eleganza che noterete quando invece riuscirete a farne uso.
In poche parole, i design pattern sono le soluzioni a problemi di programmazione che renderanno automatica l'applicazione delle corrette tecniche di progettazione.
Conoscere i design pattern inoltre aumenterà le vostre abilità di **comunicazione**, con i membri del vostro team, con altri sviluppatori, nella vostra documentazione e così via. Se per esempio avete già in mente di usare un design pattern e di applicarlo per

risolvere un problema, vi basterà citarne il nome, anziché parlare di classi, oggetti, relazioni e altre cose del genere: se dite o sentite dire "qui usiamo il pattern Strategy" e conoscete di che si tratta, è già chiaro quale soluzione viene utilizzata, non serve altro, o quasi.
Altro vantaggio dello studio dei design pattern, anche se non sempre poi venissero applicati, è quello di suggerire l'adozione di tecniche e **best practice** generali, che risulteranno essere in ogni caso di notevole utilità, soprattutto in ambito object oriented.
Infatti i design pattern nascono da questo paradigma e ne applicano i principi fondamentali, come l'incapsulamento, l'ereditarietà e il polimorfismo (che verranno anche per questo approfonditi nel prossimo capitolo), insieme ad altri principi generali di software design, e così facendo danno anche un'indicazione di come l'applicazione di questi e altri principi e/o strategie basate sul paradigma OOP permetta di progettare e sviluppare meglio.

Classificazione dei design pattern

La Gang of Four ha elaborato un catalogo di 23 design pattern fondamentali, che costituiscono i mattoncini per risolvere buona parte dei più comuni problemi di progettazione e programmazione, quindi sono considerati quasi uno standard de facto (anche se non sono gli unici design pattern, e anzi chiunque può inventarsene di nuovi!).
Naturalmente esistono design pattern che si dimostreranno più utili di altri e in più occasioni, mentre altri saranno applicabili solo in casi più specifici o particolari.
I design pattern sono classificati in base a due diversi criteri, l'**ambito**, che specifica se il pattern si applica alle *classi* o agli *oggetti* (le istanze di una classe), e lo **scopo**, che invece si riferisce all'obiettivo che il pattern si prefigge di ottenere.

- **Creazionali (o di creazione)**: sono i pattern che definiscono come viene *creato* un oggetto.
- **Strutturali (o di struttura)**: sono i pattern che si riferiscono alla *struttura* di una classe/oggetto o di una composizione di classi/oggetti.
- **Comportamentali (o di comportamento)**: sono i pattern che servono a progettare e implementare processi e algoritmi o in generale le azioni eseguite all'interno di un programma, e quindi il modo in cui classi e oggetti si *comportano* e interagiscono fra di loro, e come vengono distribuite le responsabilità dei compiti da svolgere.

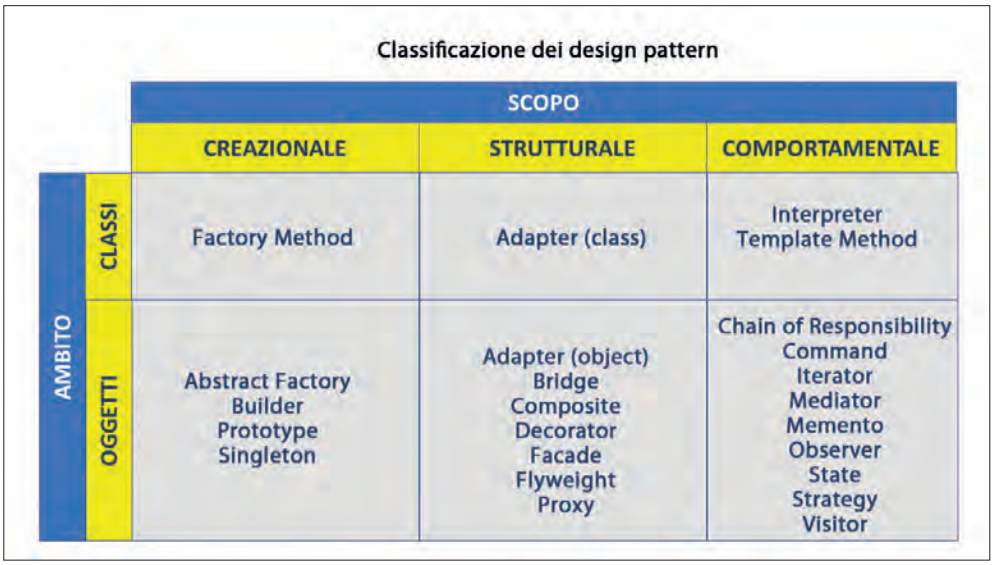


Figura 1.1 - Classificazione dei 23 design pattern della Gang of Four.

La Figura 1.1 suddivide i 23 design pattern a seconda dell'ambito e dello scopo. Si noti che la maggior parte dei pattern si trova nell'ambito *Oggetti*.
Naturalmente questa non è l'unica classificazione possibile e, come abbiamo detto, questi non sono gli unici design pattern esistenti. Altra cosa da notare, alcuni design pattern sono spesso utilizzati insieme, e altri invece possono essere considerati alternativi.
Nei paragrafi seguenti saranno ora elencati e descritti brevemente i 23 design pattern, in maniera da avere da subito una panoramica più chiara sul loro funzionamento. L'elenco può anche tornare utile come riferimento rapido, quando si avrà necessità di ricercare un particolare pattern.

NOTA

Per ogni design pattern è stata riportata fra parentesi una traduzione del nome, più che altro per comprendere meglio il significato e la scelta del nome, anche se difficilmente (se non mai) si utilizzerà il nome italiano nell'ambito lavorativo!

Nel seguito del libro tutti i design pattern verranno analizzati in dettaglio, spiegando a che cosa serve ognuno di essi, come e quando utilizzarlo, facendo uso di diagrammi UML ma anche di esempi di codice per metterli realmente in pratica. Inoltre, esistono

pattern "non standard", nel senso che non fanno parte di questi 23 della GoF, perché magari formulati successivamente, ma che si dimostreranno altrettanto utili. Alcuni di questi saranno affrontati nel seguito.
Fra l'altro molti dei design pattern GoF sono già utilizzati e implementati in diversi linguaggi di programmazione orientati agli oggetti e nelle librerie di classi a loro supporto, per esempio C# e Java. Come si vedrà in seguito, meccanismi come la gestione degli eventi, oppure per la gestione dell'interfaccia grafica a finestre, per l'iterazione degli elementi di una collezione, per la gestione di risorse condivise come file e database, sono stati progettati tenendo bene presenti uno o più design pattern.

Pattern creazionali

I pattern creazionali (o di creazione) si riferiscono al processo di creazione di classi e di oggetti. L'astrazione di tale processo, con cui si può evitare per esempio di istanziare direttamente un oggetto tramite il costruttore, permette di ottenere maggiore flessibilità e possibilità di riuso, rendendo il sistema indipendente dal modo in cui i suoi oggetti vengono creati, composti e rappresentati.

- **Abstract Factory** (in italiano "Fabbrica Astratta") fornisce un'interfaccia per creare famiglie di oggetti fra essi correlati, senza istanziarne direttamente le relative classi. Una libreria di classi per l'accesso ai database, per esempio, può contenere diverse classi per le varie operazioni da eseguire. Per ogni tipologia di database da gestire, deve essere creata una famiglia di queste classi e il pattern Abstract Factory permette di gestire la creazione dei vari oggetti che la compongono.
- **Builder** ("Costruttore") separa la costruzione di un oggetto complesso dalla sua rappresentazione, in maniera che il processo di costruzione stesso possa creare diverse rappresentazioni. In parole povere, si occupa di fornire un meccanismo per istanziare oggetti complessi, gestendo la creazione delle singole parti e assemblandole, compito che non potrebbe essere svolto con un costruttore o che comunque sarebbe più complicato o meno efficiente gestire.
- **Factory Method** ("Metodo Fabbrica") fornisce un'interfaccia per creare un oggetto, lasciando però decidere alle sottoclassi il tipo di oggetto da istanziare. Si pensi per esempio alle richieste web che possono essere inviate usando protocolli di comunicazione differenti. Diverse sottoclassi saranno dedicate ai vari protocolli, implementando le operazioni necessarie per ognuno di essi: il pattern si occuperà di istanziare l'oggetto adeguato al tipo di richiesta desiderato.
- **Prototype** ("Prototipo") permette di creare nuovi oggetti a partire da un prototipo esistente. Si pensi a un videogame in cui i mostri che il nostro personaggio

incontra siano istanze di una classe, con attributi di volta in volta differenti. Il pattern si può applicare alla creazione di un numero illimitato di mostri a partire da uno esistente, con una sorta di operazione di clonazione.

- **Singleton** ("Singoletto") permette di controllare la creazione di un oggetto, assicurandosi che esista al massimo una singola istanza della sua classe. Tipicamente il pattern è utilizzato con classi di configurazione o di sistema, che gestiscono dati di cui esiste una sola istanza e per cui quindi è spesso conveniente avere un singolo punto di accesso.

Pattern strutturali

I pattern strutturali (o di struttura) si riferiscono alla modalità con cui classi e oggetti sono combinati per ottenere strutture più o meno grandi. L'uso di principi fondamentali della programmazione a oggetti permette di aumentare la flessibilità di un sistema, la sua manutenzione, le eventuali evoluzioni, e in genere sarà consigliabile utilizzare, ove possibile, la composizione anziché l'ereditarietà.

- **Adapter** ("Adattatore") converte l'interfaccia di una classe in un'interfaccia differente. Spesso si utilizza per permettere a una classe di lavorare con dati diversi da quelli per cui era prevista, ma in genere il suo utilizzo è utile per far collaborare oggetti diversi, tramite un'interfaccia comune. Si pensi agli adattatori per le prese di corrente elettrica, con cui si può utilizzare una spina anche in prese non originariamente previste per il suo tipo.
- **Bridge** ("Ponte") separa l'astrazione di una classe dalla sua implementazione, in maniera che esse possano essere modificate in maniera indipendente. Un tipico esempio di utilizzo può essere quello in cui esistano molteplici implementazioni di una classe dedicate a piattaforme differenti, che devono essere utilizzate in maniera trasparente tramite un'interfaccia comune. Per esempio, la gestione di un'interfaccia a finestre è implementata in maniera diversa su diversi sistemi operativi, ma il codice client sarà comune per ognuno di essi.
- **Composite** ("Composto") permette di utilizzare un insieme di oggetti come se fossero un unico elemento. Rimanendo nell'ambito delle interfacce grafiche, il pattern può essere applicato ai controlli per la gestione di gruppi di oggetti, per esempio un pannello o una finestra può contenere diversi pulsanti e caselle di testo, in maniera gerarchica.
- **Decorator** ("Decoratore") consente di aggiungere dinamicamente delle funzionalità a oggetti esistenti, in alternativa alla creazione di sottoclassi. Una stessa classe quindi può essere dotata di funzionalità supplementari, con l'utilizzo di

una sorta di servizio aggiuntivo: per esempio, una finestra può essere dotata di barre di scorrimento, di bordi, di icone e così via.

- **Façade** ("Facciata") permette attraverso un'interfaccia semplificata l'accesso a funzionalità complesse, che fanno uso magari di numerose classi, di framework e di librerie esterne. Si pensi per esempio all'accensione del computer: basta premere un pulsante, ma dietro le quinte ci sono vari sottosistemi da avviare e gestire, come la cpu, la ram, l'hard disk e così via.
- **Flyweight** ("Peso mosca") permette di separare la parte variabile di una classe da quella che può essere riutilizzata, consentendo quindi di condividere un oggetto e risparmiare memoria. Si pensi a un videogame in cui l'eroe spara migliaia di proiettili con caratteristiche più o meno uguali. Anziché creare un'istanza completa per ognuno di essi, alcuni dati possono essere condivisi: ogni proiettile per esempio può usare lo stesso sprite, cioè la stessa immagine, inizializzandola e tenendola in memoria una sola volta.
- **Proxy** ("Procura") fornisce la rappresentazione o il segnaposto di un oggetto per controllare l'accesso alle sue funzionalità. In tal modo si può usare un sostituto o surrogato di un oggetto per agire al posto di un altro. Un possibile esempio di applicazione del pattern è quello in cui l'applicazione utilizza un servizio remoto o una grande mole di dati: per mezzo di un proxy si ha una semplice interfaccia che per esempio può occuparsi della connessione o del caricamento dati quando possibile, conservare i dati in una cache, e alle successive chiamate o quando necessario, utilizzare i dati già disponibili anziché aprire una nuova connessione o ricaricarli tutti.

Pattern comportamentali

I pattern comportamentali (o di comportamento) sono quelli che si riferiscono agli algoritmi, quindi alle azioni svolte da un oggetto, all'assegnazione delle responsabilità fra i vari oggetti e al loro modo di interagire e comunicare, utilizzando spesso l'incapsulamento per definire ed eseguire le operazioni richieste.

- **Chain of Responsibility** ("Catena di responsabilità") diminuisce l'accoppiamento fra un oggetto che effettua una richiesta e quello che si occupa di soddisfarla, inviandola attraverso una catena di oggetti che collaborano fra loro e di cui ognuno può decidere se processarla o meno. La gestione delle eccezioni in diversi linguaggi applica questo pattern: generata un'eccezione in un metodo, si verifica se c'è un gestore adeguato, e in caso negativo viene fatta risalire lungo lo stack delle chiamate, fino a trovarne uno.
- **Command** ("Comando") separa l'oggetto che effettua una richiesta dalla parte di codice che esegue un'azione, rappresentata anch'essa come oggetto. Si pensi

per esempio alle diverse tipologie di query eseguibili su un database, implementabili come classi che nascondono la complessità di ogni operazione.

- **Interpreter** ("Interprete"), dato un linguaggio, definisce una rappresentazione della sua grammatica e la utilizza per interpretare delle "espressioni". Alcuni problemi sono più facili da risolvere in specifici linguaggi e usando quindi un particolare modo di scriverli. Per esempio, le espressioni regolari permettono di scrivere un'espressione di ricerca di un testo per mezzo di sequenze di simboli, che identificano un insieme di stringhe. Utilizzare le espressioni regolari in un linguaggio di programmazione richiede l'implementazione di un interprete dedicato a tale scopo.
- **Iterator** ("Iteratore") si occupa dei meccanismi di accesso e navigazione degli elementi di una struttura dati, nascondendo i dettagli della struttura interna e la complessità dell'implementazione. Molti linguaggi implementano il pattern per effettuare un'iterazione degli elementi di una collezione, senza che sia necessario conoscere il tipo di oggetti che essa contiene.
- **Mediator** ("Mediatore") si occupa della coordinazione di diversi oggetti che collaborano fra loro, interponendosi nella loro comunicazione e facendo in modo che essi non interagiscano direttamente, agendo quindi da mediatore fra le parti. In tal modo ogni oggetto deve solo preoccuparsi di informare il mediatore, che si occuperà di aggiornare e agire sull'intero sistema. Pensate alla gestione di un aeroporto: gli aeroplani o meglio i piloti colloquiano con la torre di controllo, che coordina e gestisce il traffico aereo, senza che essi debbano comunicare fra di loro.
- **Memento** ("Promemoria") permette di estrarre lo stato interno di un oggetto, in maniera da conservarlo e poterlo ripristinare in un momento successivo. Un tipico esempio è l'operazione di Undo, che permette di ripristinare lo stato di un sistema a come era prima dell'esecuzione di un'operazione.
- **Observer** ("Osservatore") definisce la dipendenza di un oggetto da un insieme di altri oggetti, in maniera tale che, se un oggetto cambia il suo stato, tutti gli oggetti dipendenti, che lo osservano, siano notificati del cambiamento avvenuto. È il tipico meccanismo utilizzato nella gestione degli eventi, in cui un sottoscrittore vuole ricevere una notifica qualora si verifichi un evento in un dato oggetto, e l'oggetto osservato, in cui avviene l'evento, può appunto notificare tutti gli osservatori interessati.
- **State** ("Stato") permette a un oggetto di modificare il proprio comportamento quando cambia il suo stato interno. Per esempio, un televisore può essere spento, in stand-by e acceso. Il suo comportamento dipende dallo stato in cui si trova: solo nello stato acceso è possibile cambiare canale o modificare il volume.

- **Strategy** ("Strategia") definisce una famiglia di algoritmi, incapsulando ognuno di essi, e permette di selezionare dinamicamente quello da utilizzare o comunque modificarlo in maniera indipendente dall'utilizzatore. Per esempio, esistono diversi algoritmi di ordinamento e tramite il pattern Strategy è possibile eseguirne uno diverso ottenendo il risultato voluto in maniera trasparente.
- **Template Method** ("Metodo Schema") permette di definire la struttura generale di un algoritmo, lasciando alle sottoclassi il compito di implementarne alcuni passi in maniera differente, personalizzando anche solo parzialmente il comportamento senza modificare invece le parti comuni predefinite. Un esempio potrebbe essere l'esportazione di file in diversi formati, per cui per esempio la creazione e il salvataggio del file è comune, ma la creazione delle sue parti nei vari formati è differente.
- **Visitor** ("Visitatore") permette di separare un algoritmo dalla struttura composta da oggetti a cui è applicato, in modo da poter aggiungere o modificare le operazioni senza modificare gli oggetti che compongono la struttura stessa. Riprendendo l'esempio di esportazione di dati, il pattern permette di applicare tale operazione a documenti composti magari da diverse parti, rappresentati da istanze di classi diverse, agendo su ognuna di esse in maniera differente ed eseguendo l'operazione di esportazione nel modo più adeguato.

Scelta dei design pattern

La scelta di uno o più design pattern, in base al problema da risolvere, non è sempre semplice, soprattutto se non si ha molta familiarità con l'elenco visto nei precedenti paragrafi e non si conoscono i dettagli di ogni pattern.
Naturalmente l'esperienza ha un ruolo fondamentale. I design pattern sono pratiche di buon design e buona programmazione che si imparano con l'uso e con l'applicazione. Anzi, l'esperienza, soprattutto nella programmazione orientata agli oggetti, potrebbe far scaturire la nascita dei propri pattern, dei propri schemi mentali e pratici, di metodologie che diventeranno sempre più usati se ci si accorge che forniscono soluzioni corrette ed efficaci.
Il significato stesso della parola pattern, nel senso di schema, indica che, se si riconosce uno schema simile in un problema di programmazione, probabilmente si potrà far uso di uno stesso pattern e applicarlo come soluzione.
Il nome e la descrizione, già visti, possono dare una prima indicazione, ma non sono sufficienti per orientarsi!

Nei prossimi capitoli, andando a esaminare a fondo le caratteristiche di ogni pattern, si cercherà di mostrare anche l'approccio corretto per tale scelta. In generale saranno diversi i criteri su cui basare la selezione.

Riepilogo

La conoscenza dei design pattern per analisti e sviluppatori software costituisce un bagaglio culturale che vi permetterà di fare un notevole balzo in avanti. Anche se conoscete benissimo la sintassi di un linguaggio di programmazione orientato agli oggetti, come C# o Java, e sviluppate programmi reali, l'applicazione dei design pattern al lavoro quotidiano migliorerà la produttività e la qualità del vostro codice ma anche del vostro modo di lavorare.