Indice

Ingegneria del Software

Francesco Mattone

6 luglio 2024

Capitolo 1

La MDA (Model Driven Architecture) è un processo di modellazione del codice basato sui diagrammi ottenuti tramite UP e UML.

1.1 UML

Linguaggio di modellazione usato in OOP.
Unifica cicli di sviluppo, domini applicativi, processi di sviluppo con una struttura statica e un comportamento dinamico.

Costituito da:

Diagrammi

Sono 13 e modellano la struttura statica e dinamica, sono una vista del modello e lo strumento principale per aggiungere informazioni.

Vincoli

Sono frasi di testo tra {} che definiscono una condizione/regola dell'elemento, che deve essere sempre vera.

Stereotipi

Variazione di un elemento esistente che ha stessa forma ma diverso scopo.

Proprietà

Qualsiasi valore associato ad un elemento di modellazione.
Molti elementi hanno proprietà predefinite.

Etichette

Informazioni aggiuntive da applicare agli elementi di modellazione o proprietà di nuovi elementi.

1.2 UP

- È pilotato dai casi d'uso e dai fattori di rischio.
- È incentrato sull'architettura
- È iterativo ed incrementale

Progetto

Il progetto è diviso in: **fasi**: il completamento di una fase ci fa raggiungere una *milestone*.

- **Avvio**: stabilire la fattibilità dei requisitifondamentali, business case e rischi critici.
- **Elaborazione**: baseline eseguibile, valutazionerischi, determinazione attributi di qualità, casi d'uso evalutazione risorse.
- **Costruzione**: terminare quanto fatto prima.
- **Transizione**: eliminare errori, creare sito utenti,adattare software al sito, manuale utente, consulenza utente,riepilogo.

Ciascuna fase è formata da **iterazioni**: il completamento di un iterazione ci fa raggiungere una *baseline*.

- Pianificazione
- Analisi e progettazione
- Costruzione
- Integrazione e test
- Rilascio

Ogni iterazione si compone di:

- Requisiti
- Analisi
- Progettazione
- Implementazione
- Test

Capitolo 7

7.1 Ciclo di vita

È l'insieme delle attività ed azioni intraprese per realizzare un progetto software.

Cascata

1. **Studio di fattibilità**:

   I progettisti devono capire cosa vogliono i committenti.

   Stima di costi e tempi.

   Suddivisione progetto.

2. **Fase di codifica**:

   Il progetto diventa un'unica entità su cui vengono svolti i test prestazionali.

3. **Fase di manutenzione**:

   Dopo l'entrata in produzione, è la fase più costosa. Ci sono 3 tipi di manutenzione:

   - *Correttiva*: corregge errori nella progettazione.
   - *Adattiva*: aggiornare il sistema a cambiamenti esterni.
   - *Perfettiva*: caratteristiche aggiuntive.

   Accquanto a queste ci sono 3 attività parallele:

   - *Produzione documentazione*
   - *Controllo qualità*
   - *Gestione processo*

Iterativo

È il modello più efficace per la *modifica* dei requisiti. Lo sviluppo avviene ripercorrendo un ciclo iterativo, ciò ci permette di sapere come agire in quella successiva.
L'ordine di implementazione dipende dalle esigenze del progetto.
Le iterazioni terminano una volta implementati tutti i requisiti.

Spirale

È composta da 4 fasi:

1. **Analisi**: Determinazione obiettivi e vincoli.
2. **Progetto**: Valutazione alternative e identificazione rischi.
3. **Sviluppo**: Codifica e test.
4. **Collaudo**: Verifica operativa e pianificazione iterazione successiva.

La spirale è composta da anelli (dati da queste 4 fasi), al termine di ogni anello si avrà un'applicazione sempre più vicina a quella finale.
Tra i vantaggi ci sono: riusabilità del codice, eliminazione progressiva degli errori, integra sviluppo e manutenzione.

7.2 Metodologie agili

Pongono l'accento sulla **codifica**.
Mantengono la fase di analisi in cui definiamo i requisiti.
Passiamo all'implementazione di ogni singolo requisito, si assegna la responsabilità di un requisito ad un programmatore che se ne occuperà fino ai test.

Extreme programming

Con questa metodologia si producono solo i semilatorati strettamente necessari, la produzione non è analizzata o pianificata a priori e il processo è un'insieme di attività da decidere di volta in volta.

Il processo è formato da 4 attività: coding-testing-listening-design.
Si assegnano a coppie di programmatori le *userstory*. Le coppie vengono cambiate durante la progettazione per favorire la collettivizzazione del codice.
I test vengono scritti prima del codice.

Scrum

Si basa su 3 elementi fondamentali ed è diviso in sottoiterazioni di 24 ore:

1. **Sprint**: Brevi iterazioni di 1 mese in cui vengono soddisfatti i requisiti. Alla fine si ottiene un semilavorato funzionante.
2. **BackLog**

   Nel Backlog le funzionalità da soddisfare sono:

   - *Product*: relative al prodotto.
   - *Sprint*: relative allo sprint.
3. **Meeting**

   I meeting possono essere:

   - Daily
   - Sprint planning
   - Sprint review

   Nei meeting entrano in gioco 3 figure: product owner, scrum team, scrum master.

Kanban

È un sistema di gestione del flusso che si basa su 5 punti:

1. **Visualizzare il flusso di lavoro**

   Scomponiamo il processo in fasi distinte e ne controlliamo l'avanzamento sulla **kanban board** grazie ai flussi di lavoro.

   La visualizzazione dei flussi si compone di 7 passi:

   I. Identificare il flusso di valore
   II. Identificare ambito di lavoro
   III. Mappare le fasi del flusso di lavoro su colonne
   IV. Definire i tipi di lavoro e cosa implica il "done"
   V. Decidere il modello di scheda per ogni tipo di lavoro
   VI. Posizionare elementi di lavoro sulle carte
   VII. Tracciare il flusso
2. **Limitare il lavoro in corso**

   Assicurarsi il massimo flusso di un elemento dall'inizio alla fine.

   Ci sono 3 tipi di elementi nell'inventario: materie prime, lavori in corso e prodotti finiti.

3. **Gestire e limitare il flusso**

   Kanbani si concentra sull'assicurare un flusso di lavoro fluido con strumenti per individuare i bottleneck.

4. **Esplicitare le politiche di processo**

   I processi ci dicono cosa fare, le politiche come farlo. Una buona pratica diventa un comportamento di routine ed aiuta i membri a mantenere il flusso.

5. **Opportunità di miglioramento**

   Eliminazione dei vincoli in 5 passi:

   Identificarlo → Sfruttarlo → Sbordinare il lavoro → Elevarlo → Osservarne uno nuovo

Scumban

È un ibrido tra **scum** e **kanban** ma più allineato con quest'ultima.
Non richiede che il product owner controlli il backlog poichè ci possono essere più stakeholder.
Si pianifica il lavoro solo quando è necessario e si evitano le restrizioni temporali degli sprint.
Il flusso è visualizzato sulle kanban card che permettono di dividere il lavoro in più fasi.
Gli elementi nelle colonne hanno priorità scelta dal team scrumban.

Devops

Pratica che unisce il team **Dev** (sviluppo) con quello **Ops** (supporto) in modo da garantire una collaborazione fluida ed un miglior rilascio del software.
I team possono concentrarsi unicamente sulla qualità e sulla velocità di consegna.

Si incoraggia all'eliminazione dei task manuali. Prima si sviluppa il codice a mano e lo si aggiorna nella repository, lo si genera e testa.

IaC

Infrastructure as a Code - è una pratica in cui le infrastrutture sono configurate e gestite usando il codice non manuale.

Microservizi

In questo tipo di architettura una singola app è un'insieme di piccoli servizi e componenti.

Topologie

Il modo in cui DevOps viene applicato dipende dall'organizzazione. Ne esistono 9 topologie.

1. **Collaborazion**: i team collaborano senza problemi.
2. **Fully shared op's responsabilities**: i team OPS sono integrati negli altri.
3. **OPS as Infrastructure as a Service**: hanno ancora un reparto ops.
4. **As External Service**: non avendo un team ops delegano le loro mansioni ai dev.
5. **Teams with an expiry date**: team temporaneo.
6. **Advocacy team**: facilita la collaborazione tra dev e ops ricordando le pratiche.
7. **SRE team**: team di Site Reliability Engineering che controlla che il lavoro dei dev sia conforme agli standard.
8. **Container Driven Collaboration**: incapsulare i deployment e i requisiti in un contenitore. Rimuovere dipendenze tra dev e ops.
9. **Dev & Dba Collaboration**: i ruoli di specializzazione del database di entrambi i team sono integrati.

Capitolo 8

8.1 Software come prodotto

Caratteristiche del software:

- Perminenza design
- Malleabilità
- Manutenzione correttiva e adattiva
- Qualità interna ed esterna

Il rapporto che si crea tra committente e sviluppatore può essere di 4 tipologie:

1. **Autoconsumo**: committente ≡ produttore
2. **Sviluppo interno**: committente = unità organizzativa distinta della stessa azienda
3. **Sviluppo su commessa**: software house sviluppa prodotto dietro compenso. Il cliente può detenere per intero o parzialmente i diritti sul prodotto pagando di meno.
4. **Software Shrink-Wrapped**: software house produce un suo prodotto.

La pianificazione dell'assegnazione delle risorse deve focalizzarsi nel punto minimo della curva non lineare:

$$E_N = \frac{D}{NS} + N\left[P + M(N-1)\right]$$

Dove:

- E_N = Tempo di rilascio
- P = Overhead del signolo programmatore per la pianificazione del suo lavoro
- $M(N-1)$ = Overhead del singolo programmatore per la sincronizzazione con gli altri
- S = Produttività

Costo del software

Si usa il modello **CoCoMo** che distingue tra costi diretti ed indiretti.
Entrano in gioco fattori organizzativi e dimensionali:

- Numero di istruzioni
- Competenze dei programmatori
- Complessità del programma
- Stabilità dei requisiti

La dimensione si determina con due possibili metriche:

- **Metriche dimensionali**
- **Metriche funzionali**

La tecnica di stima dei costi parte da una caratteristica misurabile per predire lo sforzo di sviluppo ed i costi generali.
La metrica dimensionale ha poco senso poichè non tiene conto della qualità del prodotto e del fatto che future modifiche potrebbero portare all'eliminazione di interi moduli.

CoCoMo2 è un modello previsionale per la stima dell'effort per un prodotto a partire dalla dimensione

Diagrammi di Gnatt

Diagramma in cui l'asse delle ascisse indica il tempo e quello delle ordinate l'insieme dei task e sottotask che compongono il progetto. Tutto al fine di una corretta stima dei tempi di realizzazione dei task e delle risorse.

Function Point

Misurano il software quantificando le funzionalità fornite verso l'esterno, basandosi sulla logica del sistema.
La FPA (Function Point Analisis) è un processo semplice ma conciso, con 5 obiettivi:

1. Valutare le richieste del cliente e quanto gli è stato fornito
2. Metrica dimensionale per qualità e produttività
3. Base per studi su prevenzione e stima
4. Misurare software indipendentemente dalla tecnica di sviluppo
5. Fattore di normalizzazione e raffronto con altre aziente

Esso inoltre fa riferimento a 3 principi fondamentali:

1. **Misurare funzionalità richieste ed ottenute dai clienti**
2. **Misurare manutenzione e sviluppo software**
3. **Ottenere conteggio corrente tra pogetti ed organizzazioni**
   (a) Tipo di conteggio:
   - Conteggio per sviluppo di progetto
   - Conteggio per manutenzione evolutiva
   - Conteggio applicativo

   (b) Ambito di conteggio e confini: per confini intendiamo la divisione tra una app e le altre. Per individuare i confini ci basiamo sulle funzionalità che l'utente può capire e descrivere.
   (c) Function Type: componenti richiesti e visibili all'untente, dividiamo in:
   - Funzioni dati
   - Funzioni transazionali

   (d) Conteggio UFP: UFP = Funzioni Dati + Funzioni Transazionali
   (e) Determinare VAF tramite caratteristiche generali del sistema:
   I Valve Adjustment Factor sono composti da 14 caratteristiche dette GSC e sono usati per valutarne la complessità. Per tale valutazione si usa una scala di 6 valori (grado di influenza) ma per alcune GSC questo metodo cambia in quanto si assegna il punteggio in base alla presenza di sottocaratteristiche.
   (f) FP: La loro individuazione avviene solo dopo il completamento dell'app.

Capitolo 9

9.1 Software coeme processo

Vi sono 3 fasi principali:

1. Verifica

Contronto tra il prodotto ottenuto e le specifiche richieste. Abbiamo 3 requisiti da verificare:

1. I casi d'uso verificano i requisiti.
2. Le classi sono in grado di fornire i casi d'uso.
3. Il codice corrisponde alle classi che compongono il progetto.

La verifica va fatta ad ogni iterazione. Includiamo i "check di sanità" della compatibilità del codice e del fatto che compili senza errori.

2. Validazione

Consiste nel coinvolgere i clienti ed avere il loro feedback alla fine di un'interazione.
Per problematiche/incertezze/incompresioni conviene fare più iterazioni brevi.
Il prodotto deve essere **intuitivo** e ciò può essere verificato da utenti esterni e/o esperti di usabilità.

3. Test

Anche la fase di test è divisa in 3 parti:

1. Trovare i bug
2. Convincere il cliente che non ci sono bug
3. Fornire informazioni sull'evoluzione del sistema

I testo sono generalmente divisi in 2 categorie:

- **Test scatola nera**: funzionalità delle entità testate
- **Test scatola bianca**: struttura delle entità testate

Affinchè il software sia percepito come di **alta qualità** dobbiamo:

1. Testare l'intero sistema
2. Verificare che continui a svolgere le funzioni supportate
3. Testare i casi tipici anzichè limite

I test dovranno essere ripetibili, documentati, precisi ed eseguiti su software a configurazione controllata. Uno script può aiutare a eseguire un insieme di test, registrare i test effettuati, confrontare i valori.
Lavorando ad oggetti, entrano in gioco altri concetti da considerare nello svolgimento dei test, tra cui:

- **Minima unità di test da considerare**

  In genere è preferibile pianifirare i test all'inizio dell'iterazione per evitare una progettazione affrettata.

- **Incapsulamento**:

  L'incapsulamento riduce il rischio di errori per comprensione errata delle interazioni tra componenti, ma rende difficile il test.

- **Polimorfismo**:

  L'ereditarietà ci aggiunge un nuovo livello di difficoltà poiché dovremmo testare tutte le funzionalità: è bene utilizzarlo se i vantaggi sono magggiori degli svantaggi.

Revisione e ispezione del codice

Detto walkthrough, è un incontro in cui cerchiamo di identificare i problemi del prodotto.
Sono noti come Formal Technical Review (FTR), si differenziano nei dettagli ma condividono 5 caratteristiche:

1. I partecipanti sono allo stesso livello gerarchico
2. L'artefatto è fornito in anticipo
3. Agenda predefinita
4. Non si discute su come rimediare agli errori
5. La procedura successiva è atta alla risoluzione

Manutezione

Modifiche apportate ad un programma dopo la sua messa in uso.
I tipi di manutenzione sono 4:

- Perfettiva: 50 %
- Adattiva: 25 %
- Correttiva + Correttiva Emergenza: 21 %
- Preventiva: 4 %

Costi

Sono legati soprattutto all'evoluzione del software e si dividono in base alla loro origine:

- **Natura tecnica**:

  Complessità codice, cambiamenti linguaggi programmazione, cambiamenti infrastruttura, qualità formattazione.

- **Costo umano**:

  Stabilità team, responsabilità contrattuale, competenze personale.

Terminologia:

- **Manutenibilità**: facilità con cui un sistema può essere modificato.
- **Tracciabilità**: grado con cui si può stabilire una relazione tra due o più prodotti del processo di sviluppo.
- **Ripple Effect (catena)**: cambiamenti in una posizione del software possono provocare cambiamenti in altre posizioni.
- **Sistemi legacy**: sistema obsoleto usato in una azienta, la quale vuole disfarsene.
- **Analisi impatto**: processo con cui si identificano le conseguenze di un cambiamento sul resto del sistema.

La qualità del software dipende anche da quella dei suoi componenti. Distinguiamo 3 tipi di qualità che fanno riferimento:

- **Interna**: alle proprietà rilevanti durante sviluppo e manutenzione.
- **Esterna**: alle proprietà che lo caratterizzano durante l'esecuzione.
- **In uso**: alle proprietà in uno specifico ambiente di esecuzione.

Riuso software

È riconosciuto che un software migliore ha bisogno di un processo di progettazione basato sul riutilizzo del software.
Questo viene fatto a diversi livelli:

- Riuso sistemi
- Riuso app
- Riuso componenti
- Riuso oggetti e funzioni

Framework

Sono entità moderatamente grandi che possono essere riutilizzate a metà strada tra il riutilizzo di sistemi e componenti.

Il framework è il progetto di sottosistema composto da una collezione di classi astratte e concrete e le interfacce tra loro.
Il sottosistema è implementato aggiungendo componenti per riempire parti del progetto ed istanziare le classi astratte nel framework.

I **Web Application Framework** supportano la costruzione di siti web dinamici per tutti i tipi di linguaggi. Il modello di interazione è sul modello di progettazione composito **Model View Controller**.

Linee prodotto software

Sono insiemi di applicazioni con architettura comune e componenti condivisi, ogni app è specializzata per soddisfare diversi requisiti.
Include:

- **Componenti di base**: danno supporto all'infrastruttura e non vengono modificati con una nuova istanza.
- **Componenti configurabili**: possono essere modificati e configurati per specializzarsi in una nuova app.
- **Componenti specializzati**: specifici nel dominio, possono essere sostituiti quando si ha un'istanza totalmente o parzialmente nuova.

Differenze tra Framework e Linee di Prodotto Software

Framework

- Si basano su caratteristiche OOP come il polimorfismo.
- Si concentrano sulla fornitura di supporto tecnico.

Linee di Prodotto Software

- Non hanno bisogno di questo tipo di caratteristiche.
- Incorporano informazioni sul dominio della piattaforma.
- Controllano app per apparecchiature.
- Costituite da famiglie di app gestite dalla stessa organizzazione.

Sistemi Applicativi Integrati

Sono applicazioni che includono due o più sistemi applicativi, eventualmente legacy.
Per svilupparli dobbiamo fare delle scelte di progetazione.
L'integrazione si semplifica in presenza di un approccio orientato ai servizi che permette l'accesso alle funzionalità attraverso un'interfaccia standard.

Alcune app possono offrire un'interfaccia di servizio che, però, a volte deve essere implementata dall'integratore di sistema.
Si deve programmare un wrapper che nasconde l'app e fornisce servizi visibili dall'esterno.

Capitolo 10

10.1 Component based development paradigma

Detto:

- Component based software engineering - CBSE
- Component based development - CBD

È una branca dell'ingegneria del software che enfatizza la tecnica di suddivisione delle attività rispetto alle funzionalità disponibili.
Si basa sul riutilizzo per definire, implementare e comporre componenti indipendenti liberamente accoppiate.

Componenti

Sono piattaforme di partenza per l'orientamento ai servizi nel web service. Package, Moduli o Risorse Web che incapsulano funzioni, parliamo di Incapsulamento nei Componenti.

- **Modulari e coesi**: processi collocati in componenti separati in modo che tutti i dati e le funzioni siano semanticamente correlati.
- **Sostituibili**: con una versione aggiornata o alternativa, senza arrecare danni al sistema dove operano.
- **Riusabili**: per essere riusabili, il software deve essere:
  - Completamente documentato.
  - Accuratamente testato.
  - Robusto.

I componenti incapsulano sia le strutture dati sia gli algoritmi applicati ad essi.
L'ingegneria basata sui componenti si basa anche sulla teoria precedente: architettura software, framework, design pattern, ecc...

Un computer che esegue diversi componenti è detto **application server** e la loro combinazione è il calcolo distribuito.
Un **component model** è una definizione di proprietà che i componenti devono soddisfare, nonché di metodi e meccanismi per la composizione dei componenti.

Distributed computing

È una branca dell'informatica che studia i sistemi distribuiti ed i calcoli distribuiti, ovvero sistemi i cui componenti si trovano su diversi computer in rete, che comunicano tra loro per raggiungere un obiettivo comune.

Si usa per risolvere problemi computazionali. Un problema viene diviso in molte task, ognuno risolto da uno o più computer comunicanti.

Questi sistemi possono essere caratterizzati da 2 tipi di calcolo:

Parallelo

Tutti i processori possono avere accesso ad una memoria condivisa per scambiarsi informazioni e può essere visto come una particolare forma accoppiata di calcolo.

Distribuito

Ogni processore ha memoria distribuita e le informazioni sono date dallo scambio di messaggi.

Architetture

Varie architetture hardware e software sono usate per il calcolo distribuito, con vari livelli:

- **Basso livello**: è necessario interconnettere più cpu con qualche tipo di rete.
- **Alto livello**: è necessario interconnettere i processori in esecuzione nelle cpu con qualche tipo di sistema di comunicazione.

La programmazione distribuita rientra in una delle seguenti **architetture**

1. **Client-Server**: i client contattano il server per i dati.
2. **Three-Tier**: spostano l'intelligenza del client su un livello intermedio in modo da utilizzarli senza stato.
3. **N-Tier**: si riferiscono ad applicazioni web che inoltrano le loro richieste ad altri servizi aziendali.
4. **Peer-To-Peer**: tutte le responsabilità sono equamente divise tra le macchine anche dette **peer**, esse fungono sia da client che da server.

Nel calcolo distribuito processi concorrenti comunicano tra loro attraverso protocolli, tipicamente in relazione **master-slave**.

10.2 Service Oriented Architecture

È uno stile che supporta all'orientamento ai servizi ed è applicata in quei casi in cui dei componenti forniscono servizi ad altri.
È pensata per essere indipendente dai fornitori, prodotti e tecnologie.

Servizio

È un'unità discreta di funzionalità a cui si può accedere da remoto e su cui possiamo agire ed aggiornare in modo **indipendente**.
Ogni servizio ha 4 proprietà:

1. Attività business ripetibile
2. Autocontenuti
3. È una scatola nera
4. Può essere composto da altri servizi

Diversi servizi possono essere usati insieme come una rete per fornire le funzionalità di una grande app.

La SOA è supportata da terminologie e standard che facilitano la comunicazione e la cooperazione. Essa è legata al concetto di API (Application Programming Interface).

Valori fondamentali delle SOA

La SOA separa le funzioni in unità distinte, o servizi, che gli sviluppatori rendono accessibili su rete per permettere la combinazione e la riutilizzazione.
Il manifesto delle SOA del 2009 prevede 6 valori fondamentali:

1. Valore business
2. Obiettivi strategici
3. Servizi condivisi
4. Interoperabilità intrinseca
5. Flessibilità
6. Perfezionamento evolutivo

Principi

Non ci sono standard relativi alla composizione di una SOA, ma dalle fonti pubblicate, in linea di massima, abbiamo:

1. Contratto servizio standardizzato
2. Autonomia riferimento del servizio
3. Trasparenza posizione del servizio
4. Longevità del servizio
5. Astrazione del servizio
6. Autonomia del servizio
7. Apolidia del servizio
8. Granularità del servizio
9. normalizzazione
10. Possibilità di comporli
11. Scoperta del servizio
12. Riutilizzabilità
13. Incapsulamento

Funzionamento

Ogni building block può essere:

- **Fornitore di servizi**: crea un sito web e fornisce le sue info al registro dei servizi.
- **Broker, registro o deposito servizi**: deve rendere le info riguardanti il servizio web disponibili a qualsiasi potenziale richiedente.
- **Richiedente/consumatore**: tramite i broker si legano ai fornitori per richiedere uno o più servizi.

Real Time Computing

È il termine per indicare i sistemi hardware e software soggetti ad un vincolo di tempo indicati come deadline, dell'ordine dei millisecondi.
L'elaborazione in tempo reale fallisce se non completata entro la scadenza specifica. Questo tipo di sistema controlla un ambiente ricevendo dati, elaborandoti e restituendo i risultati in modo sufficientemente rapido da influenzare l'ambiente in quel momento.

Criteri

Un sistema è in tempo reale se la correttezza totale di un'operazione dipende anche dal tempo in cui viene eseguita.
Si classificano i sistemi in base alle conseguenze di una mancata deadline:

- **Hard**: fallimento totale
- **Firm**: tollerabili, ma possono degradare la qualità del servizio, l'utilità diventa zero.
- **Soft**: tollerabili, ma possono degradare la qualità del servizio, l'utilità diventa zero dopo la deadline.

Capitolo 11

Pattern

11.1 Pattern Architetturali

Includono informazioni su quando il suo utilizzo è appropriato e i dettagli su vantaggi e svantaggi.

1. Model View Controller

È la base per la gestione delle interazioni in molti sistemi web, supportato da molti linguaggi.
Si basa sul principio di separazione delle responsabilità, il sistema è strutturato in 3 componenti logiche che interagiscono.

- **Model**: si occupa dei dati del sistema e delle operazioni associate, quindi dello stato e della logica dell'applicazione.
- **View**: definisce e gestisce il modo in cui i dati sono presentati all'utente, quindi dell'interfaccia in risposta ad aggiornamenti del model.
- **Controller**: gestisce l'interazione degli utenti e traduce l'input in aggiornamenti passai al Model e View.

Quando si usa?

Quando ci sono più modi per visualizzare ed interagire con i dati e quando non sono noti i requisiti futuri di interazione e presentazione dati.

Vantaggi

- Rappresentazioni multiple della stessa informazione.
- Consente di modificare facilmente le interfacce utente.
- Consente di cambiare facilmente le risposte agli input dell'utente.
- Promuove il riuso.
- Consente agli sviluppatori di aggiornare simultaneamente interfaccia logica o input di un'app.
- Consente agli sviluppatori di focalizzarsi su un aspetto per volta.

Svantaggi

- Potrebbe richiedere altro codice o codice complesso per interazioni semplici.

Come funziona il ciclo?

Sebbene le classi MVC siano separate, devono comunque comunicare regolarmente, pertanto ogni oggetto in MVC deve memorizzare un riferimento agli altri oggetti con cui interagisce.
La comunicazione prevede una direzione attraverso i riferimenti:

1. La view riceve l'input dall'utente e lo passa al controller.
2. Il controller riceve l'input dell'utente dalla view.
3. Il controller modifica il model in risposta all'input dell'utente.
4. Il model cambia in base agli aggiornamenti del controller.
5. Il model notifica i cambiamenti alla view.
6. La view aggiorna l'interfaccia utente,.

Responsabilità

Model

- Memorizza i dati e fornisce dei metodi specifici per definire i valori di questi dati e per recuperarli successivamente.
- I metodi di gestione dei dati non sono generici. Sono personalizzati per l'app e devono essere conosciuti al controller e alla view.
- I controller sono personalizzati per manipolare un model.
- I dati di un model possono essere cambiati da classi esterne o dalla propria logica interna.
- Il model deve poter far registrare le view.
- Se il model subisce un cambio significativo deve comunicarlo alle view.
- Il model implementa la logica del pattern MUC.
- Il model può fornire servizi di validazione dati.

View

- Creare interfaccia utente e aggiornarla.
- Attendere cambiamenti dello stato del model.
- Passare gli eventi dell'input al controller.
- Non puà cambiare il model ma può chiedergli delle informazioni.
- Il cambiamento di una view si riflette sulle altre.
- La logica è centralizzata nel model, evitando duplicazioni di codice e modificando la view a runtime.

Controller

- Si pone in attesa di notifiche dalla view e traduce gli input in cambiamenti del model.
- Interpreta opportunamente gli input prima di cambiare il model.
- Quando gli input non hanno effetti sul model esso istruisce la view per cambiare l'interfaccia.

2. Layered Architecture

È un altro modo di ottenere separazione ed indipendenza in quanto le funzionalità sono organizzate in strati separati che si basano sulle funzionalità ed i servizi dati dallo strato sottostante.

Descrizione

Uno strato fornisce i servizi altro strato superiore, quindi quelli di livello più basso rappresentano i servizi base.
Quest'architettura è modificabile e portabile; inoltre, se l'interfaccia non è modificata, un nuovo strato con funzionalità estese può sostituire uno esistente senza modificare altre parti del sistema.
Se si modificano le interfacce di uno strato o si aggiungono funzionalità, ne risente solo quello adiacente.

Quando si usa

Si usa:

- Per costruire nuove funzioni per un sistema esistente.
- Quando lo sviluppo degli strati (e relative funzionalità) è distribuita su più team.
- Quando c'è una richiesta di protezione su più livelli.

Vantaggi

- Consente la sostituzione di interi strati se l'interfaccia non viene modificata.
- Le funzioni ridondanti possono essere fornite in ogni strato per aumentare la fidatezza.

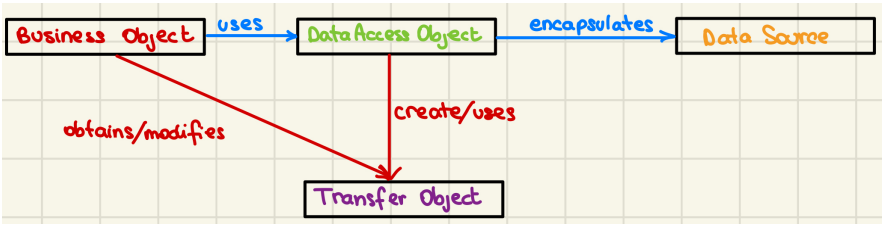
Svantaggi

- Difficile ottenere separazione netta degli strati.
- Le performance possono essere un problema a causa delle varie interpretazioni in quanto le richieste sono elaborate da ogni strato.

3. Data Access Object

Ha lo scopo di disaccoppiare l'accesso ai dati rispetto alla memorizzazione sottostante, consente di scegliere il tipo di DBMS da usare durante l'implementazione.
Gli oggetti DAO consentono la portabilità dell'app da una sorgente dati all'altra, senza cambiare la logica business.

Portiamo un esempio di diagramma delle classi per il pattern DAO:



- **Business Object**: classe con la logica business che ha la responsabilità di specificare cosa e come modificare, non come memorizzare.
- **DataAccess Object**: nasconde la **Data Sources** effettiva.

  Una classe DataAccess Object può essere sostituita con un'altra se cambia la sorgente informativa.
- **Transfer Object**: viene usato per trasferire i dati da DataAccess Object a Business Object e viceversa. Rappresenta i dati nel Database e non è direttamente connesso a Data Source. Qualunque cambiamento a Transfer Object deve passare per DataAccess Object prima di diventare permanente.

4. Repository

Spiega come una serie di componenti interattivi possono condividere dati.

Descrizione

Tutti i dati di un sistema vengono gestiti in un database centrare (repository) accessibile da tutti i componenti del sistema.
I componenti interagiscono soltanto attraverso la repo.

Quando si usa?

Si usa:

- Quando nel sistema sono generati grandi volumi di informazioni che devono essere memorizzate per lungo tempo.
- Nelle applicazioni in cui i dati sono generati da un componente ed usati da un altro.
- Nei sistemi guidati dai dati, dove la loro inclusione nella repositori innesca l'azione/utilizzo di uno strumento

Vantaggi

- I componenti possono essere indipendenti.
- Le modifiche fatte da un componente possono essere rese note agli altri.
- Tutti i dati sono gestiti in modo coerente poichè si trovano nello stesso posto.

Svantaggi

- La repository è un punto comune di malfunzionamento.
- Possono esserci inefficienze di organizzazione nella comunicazione con la repository.
- Può essere difficile distribuire le repository su più computer.

5. Client-Server

Illustra una tipica organizzazione a runtime di sistemi distribuiti.

Descrizione

In questo tipo di architettura il sistema è presentato come un insieme di servizi, ciascuno dei quali è fornito da un server separato.
I client sono utenti e accedono ai server per usufruire di tali servizi.

Quando si usa?

Quando occorre accedere ai dati di un database da più postazioni. Poichè i server possano essere replicati, lo schema può essere utilizzato anche quando il carico su un sistema è variabile.

Vantaggi

- I server possono essere distribuiti in una rete.
- Le funzionalità comuni sono a disposizione di tutti i client.
- Le funzionalità comuni non devono essere implementate da tutti i servizi.

Svantaggi

- Ogni servizio è un punto comune di malfunzionamento, soggetto a attacchi DOS e a guasti del server.
- Prestazioni imprevedibili, dipendenti da rete e da sistema.
- Problemi di gestione se i servizi sono di proprietà di più organizzazioni.

Come funziona il ciclo?

I principali componenti di questo modello sono:

- **Insieme di server** che offrono servizi ad altri sottoinsiemi.
- **Insieme di client** che richiedono i servizi offerti dai server.
- **Rete** che permette ai clienti di accedere ai servizi.

Le architetture sono di solito come architetture di sistemi distribuiti, ma il modello logico di servizi indipendenti eseguiti su server separati può essere implementato su un singolo dispositivo.

6. Pipe-and-Filter

Rappresenta l'organizzazione a runtime di un sistema dove le trasformazioni funzionali elaborano i loro input e generano output.

Descrizione

L'elaborazione dei dati del sistema è organizzata in modo che ciascun componente di elaborazione è discreto e svolge un particolare tipo di trasformazione dei dati.
I dati fluiscono in un tubo (pipe) da un componente all'altro per essere elaborati.

Quando si usa?

Nelle applicazioni per l'elaborazione dei dati (barch o basata su transazioni), dove gli input vengono elaborati in fasi separate per generare i relativi output.

Vantaggi

- Facile da capire e supporta il riutilizzo delle trasformazioni.
- Il flusso delle operazioni rispecchia la struttura di molti processi aziendali.
- Semplice evoluzione che permette di aggiungere trasformazioni.
- Può essere implementato come sistema sequenziale o parallelo.

Svantaggi

- Il formato di trasferimento dei dati deve essere concordato.
- Ogni trasformazione deve leggere l'input e fornire l'output nel formato concordato.
- Aumentando così gli overhead del sistema e protrebbe rendere impossibile il riutilizzo dei componenti che usano strutture dati non compatibili.

Come funziona il ciclo?

I dati passano da una trasformazionne all'altra attraverso una sequenza ed ogni passo di elaborazione è implementato come trasformazione.
I dati possono essere elaborati da ogni trasformazione elementare per elementi o in blocco.

11.2 Design Pattern

Gli ingegneri del software hanno, nel corso del tempo, individuato delle best practice per la progettazione di software anche a livello di codice.
I **Design Pattern** sono quindi dei pattern che specificano le linee guida sul progetto, ma a basso livello.
In generale il pattern non rappresenta un algoritmo, ma una metodologia.
Descrivendo un design pattern si forniscono delle voci che sono quasi sempre le stesse, ovvero:

1. L'**intento** del pattern: descrizione del problema relativo al apttern e soluzione.
2. La **motivazione**: spiegazione dettagliata del problema e della soluzione fornita.
3. La **struttura delle classi**: mostra ogni parte del pattern e come queste sono legate.
4. L'**esempio di codice**.

Perché usarli?

Rappresentano un toolkit di soluzioni provate e collaudate a problemi comuni e dunque insegnano a risolvere ogni tipo di problema utilizzando i principi della OOP.
Vengono utilizzati per 3 motivi principali:

- Risparmio di tempo.
- Permettono di scrivere codice standard.
- Supportano l'informatica del riutilizzo.

I design pattern sono 23 in totale divisi in 3 categorie:

- **Pattern Creazionali**: hanno a che fare con la creazione di **istanze**.
- **Pattern Strutturali**: Hanno a che fare con la composizione di **classi** ed **istanze**.
- **Pattern Comportamentali**: Hanno a che fare con le inetrazioni di classi ed oggetti tra loro.

11.3 Pattern Creazionali

1. Factory Method

Fornisce un'interfaccia per la creazione di oggetti in una superclasse, ma permette alle sottoclassi di modificare il tipo di oggetti che verranno creati.

Esempio pratico

Abbiamo una app per la logistica. La prima versione si occupa solo dei camion, vogliamo incorporarci la logistica marittima; come facciamo per il codice? (Supponendo che la maggior parte del codice sia accoppiato alla classe "Truck").

Il pattern **factory** suggerisce di sostituire le chiamate dirette alla costruzione di oggetti (usando new) con chiamate ad uno speciale metodo factory.
In questo modo possiamo **sovrascrivere** factory in una sottoclasse e cambiare la classe dei prodotti creati dal metodo, ciò è possibile solo se tali sottoclassi hanno in comune una superclasse o interfaccia.

2. Abstract Factory

Consente di produrre linee di oggetti correlati senza specificarne le classi concrete.

Esempio pratico

Simulatore di un negozio di mobili. Il nostro codice è costituito da classi che rappresentano: **famiglia** di prodotti e varianti di questa famiglia.

Vogliamo un modo per creare singoli oggetti d'arredo che si abbinino ad altri oggetti della stessa famiglia senza cambiare il codice esistente quando si aggiungono nuovi prodotti/famiglie.

Il patter abstract factory suggerisce di dichiarare esplicitamente le interfacce (chiamiamole interfacce 1) per ogni prodotto distinto della famiglia e fare in modo che tutte le varianti dei prodotti seguano quelle interfacce 1.
Dichiariamo, a questo punto, l'Abstract Factory, un'interfaccia con un elenco di metodi di creazione per tutti i prodotti che fanno parte della famiglia. Tali metodi devono restituire i tipi di prodotti astratti rappresentati dalle interfacce 1.
Per ogni variante di una famiglia creiamo una classe factory separata basata sull'Abstract Factory.

Una factory è una classe che restituisce prodotti di un particolare tipo, il **codice del client** deve operare sia con le fabbriche che con i prodotti in modo da permetterci di cambiare fabbrica e variante che gli si passa senza rovinare il codice.
Gli oggetti concreti sono creati dall'app in fase di inizializzazione, poco prima dell'inizializzazione in se, l'app deve selezionare il tipo di factory a seconda della configurazione o delle impostazioni dell'ambiente.

3. Builder

Costruire oggetti complessi passo dopo passo e produrre diversi tipi e rappresenzazioni di un oggetto usando lo stesso codice per la loro costruzione. Esso è utile quando si parla di customer massification, cioè massificare la personalizzazione dei prodotti rendendo ogni oggetto un assemblaggio completo di più oggetti.

Esempio pratico

Pensiamo a come costruire un oggetto "House" che può essere una casa semplice o con più dettagli.
Si potrebbero avere 2 approcci:

1. Creare una sottoclasse per ciascuna variante ma avremmo un numero eccessivo di queste.
2. Potremmo porre tutti i parametri delle varianti come attributi di "House" ma avremmo molti parametri "null" e chiamate dei costruttori brutte.

Il Pattern Builder suggerisce di estrarre il codice di costruzione dell'oggetto della propria classe e spostarlo in oggetti separati chiamati builder. Il pattern organizza la costruzione dell'oggetto in una serie di passi corrispondenti a metodi diversi.
Per creare un oggetto si esegue una serie specifica di passi che cambia in base ai parametri desiderati.

I vari metodi possono essere dichiarati in un interfaccia e questa può essere implementata da diversi builder che costruiscono lo stesso oggetto in modo differente → il client deve conoscere i metodi a disposizione per ottenere quello che desidera.

4. Prototype

Permette di copiare oggetti esistenti senza rendere il codice dipendente dalle loro classi, e quindi senza conoscerle in dettaglio.
Supponiamo di avere un oggetto e volerne creare la copia esatta, dobbiamo considerare tutti i campi dell'oggetto originale e copiare i loro valori sul nuovo ma ciò a volte non possiamo farlo a causa della visibilità di alcuni campi.
Un ulteriore problema è dato dal fatto che così facendo il codice diventa dipendente dalla classe dell'oggetto duplicato, inoltre capita di conoscere solo l'interfaccia che l'oggetto segue ma non la sua classe concreta.

Se il pattern Prototype suggerisce di delegare il processo di clonazione agli oggetti reali che vengono clonati, quindi dichiarare un'interfaccia comune per tutti gli oggetti che supportano la clonazione.
L'implementazione di tale metodo semplicemente crea un oggetto della classe corrente e porta tutti i valori dei campi dal vecchio oggetto in quello nuovo, così facendo è l'oggetto originale ad auto-clonarsi.

5. Singleton

Consente di garantire che una classe abbia una sola istanza, fornendo un punto di accesso globale a tale istanza.

Il pattern Singleton risolve due problemi violando il principio della responsabilità unica.

- **Assicurarsi che una classe abbia una sola istanza**

  Per controllare l'accesso a qualche risorsa condivisa. Quello che il pattern fa è che quando si vuole creare un nuovo oggetto, ma ne è già stato creato uno, ci restituisce quello già creato.

- **Fornire un punto di accesso globale all'unica istanza**

  Anzichè creare variabili globali per memorizzare oggetti essenziali, dove un qualsiasi codice può sovrascrivere il contenuto, possiamo accedere ad alcuni oggetti da qualsiasi punto del programma proteggendo l'istanza dalla sovrascrizione.

Tutte le implementazioni del pattern Singleton hanno due passi in comune:

1. **Rendere il costruttore privato** per evitare che altri oggetti lo chiamino.
2. **Creare un metodo statico di creazione che funga da costruttore**: chiama il costruttore privato per creare un oggetto e salvarlo in un campo statico; così facendo tutte le successive chiamate a questo metodo restituiscono l'oggetto salvato nel campo statico che è sempre lo stesso.

11.4 Pattern Strutturali

Riguardano le modalità con la quale classi ed oggetti vengono aggregati per formare entità più complesse.
Ne esistono 2 categorie:

- **Basati sugli oggetti**

  Descrivono le modalità di composizione di oggetti al fine di estendere in fase di esecuzione le funzionalità di una classe.

- **Basati sulle classi**

  Sfruttano l'ereditarietà multipla.

1. Adapter

Consente la collaborazione di oggetti con interfacce compatibili.

Esempio pratico

App per il monitoraggio del mercato aziendale che scarica vari dati in XML e li visualizza. Ad un certo punto decidiamo di migliorare l'app con librerie di analisi intelligente di terze parti, ma questa funziona solo con dati in formato JSON.

Il patter Adapter suggerisce di creare un adattatore, speciale oggetto che converte l'interfaccia di un oggetto in modo che un altro possa capirlo mascherando gli oggetti, e quindi la complessità del tutto. La conversione usa 3 passaggi:

1. L'adattatore definisce un'interfaccia compatibile con uno degli oggetti esistenti.
2. Tramite l'interfaccia l'oggetto esistente può chiamare i metodi dell'adattatore.
3. Alla ricezione di una chiamata, l'adattatore passa la richiesta al secondo oggetto, ma in un formato e ordine che esso si aspetta.

2. Bridge

È un pattern strutturale che consente di suddividere una classe in due gerarchie che possono essere sviluppate indipendentemente l'una dall'altra.

Esempio pratico

Supponiamo di avere una classe "Shape" avente due sottoclassi "Sphere" e "Cube". Immaginiamo di voler estendere questa gerarchia con colori per ogni forma.
Avendo due sottoclassi si andranno a generare quattro combinazioni di sottoclassi. Volendo estendere ulteriormente andrebbe ad aumentare il numero di sottoclassi.
Il problema nasce poichè vogliamo estendere le classi delle forme lungo due dimensioni indipendenti.

Il pattern Bridge suggerisce di passare l'ereditarietà alla composizione degli oggetti: significa che si estrae una delle dimensioni in una gerarchia di classe separata in modo che le classi originarie facciano riferimento ad un oggetto detta nuova gerarchia.

Tornando all'esempio, la classe Shape ottiene un attributo che si riferisce ad uno degli oggetti colore e questo riferimmento fungerà da ponte tra Shape e Color, quindi con questo pattern l'aggiunta di colori non richiederà di cambiare la gerarchia delle forme e viceversa.

Astrazione e Implementazione

- **Astrazione** (anche detta interfaccia)

  è uno strato di controllo ad alto livello per alcune entità che non deve svolgere alcun lavoro concreto da solo ma delegare il lavoro al livello di implementazione.

  Si considera GUI.

- **Implementazione** (anche detta piattaforma)

  Codice che lavora. Si considera API.

3. Composite

Consente di comporre gli oggetti in strutture ad albero e poi lavorare con queste strutture come fossero singoli oggetti.

Esempio pratico

Immaginiamo due tipi di oggetti "Product" e "Box", correlati tra loro dal fatto che una Box può contenere diversi Product e Box più piccole.
Supponiamo di voler creare un sistema di ordini che utilizzi queste classi dove gli ordini possono riguardare prodotti semplici o box (per queste il prezzo totale si può trovare solo dopo averle scartate (approccio eccessivamente complesso)).

Il pattern Composite suggerisce di lavorare con Product e Box attraverso un'interfaccia comune che dichiara un metodo per calcolare il prezzo totale. Il più grande vantaggio è che non è necessario preoccuparsi delle classi di oggetti concreti che compongono l'albero, nel senso che non serve sapere se un oggetto è un prodotto o una box, è possibile trattarli egualmente attraverso l'interfaccia comune.

4. Decorator

Permette di aggiungere nuovi comportamenti agli oggetti posizionandoli all'interno di speciali oggetti wrapper che contengono i comportamenti.

Esempio pratico

Immaginiamo di lavorare ad una libreria di notifiche che permette ad altri programmi di notificare ai propri utenti eventi importanti.
Inizialmente sviluppiamo una classe Notifier con compi campi, un costruttore ed un metodo *send*().

Se gli utenti desiderano ricevere notifica su Facebook e Slack potremmo estendere la classe Notifier con sottoclassi che abbiano metodi di notifica aggiuntivi. Se gli utenti volessero avere diversi tipi di notifica contemporaneamente dovremmo creare altre sottoclassi date dalle varie combinazioni, e il codice aumenterebbe eccessivamente.

Il pattern Decorator suggerisce di inserire un wrapper, ovvero un oggetto che può essere collegato a qualche oggetto target.
Esso conterrà lo stesso insieme di metodi del target delegando ad esso le richieste che riceve, ma può alterare il risultato. Agli occhi del client i due oggetti sono identici.

Nell'esempio delle notifiche trasformiamo i metodi in decoratori, in questo modo il client dovrebbe avere un notificatore di base in un insieme di decoratori che corrispondono alle preferenze del cliente.

5. Facade

Fornisce un'interfaccia semplificata per una libreria, un framework o qualsiasi altro insieme complesso di classi. Tale pattern è fondamentale con sistemi legacy.

Esempio pratico

Immaginiamo di lavorare con un insieme di classi legato ad una sofisticata libreria o ad un framework. Normalmente si dovrebbero inizializzare tutti gli oggetti ed eseguire i metodi nell'ordine corretto. Così facendo la logica di business delle classi diventerebbe legata ai dettagli di implementazione delle classi di terze parti, rendendola difficile da comprendere e mantenere.

Il pattern Facade suggerisce di creare una classe facade che fornisce una semplice interfaccia ad un sottosistema complesso. Essa può fornire funzionalità limiitate rispetto a quelle del sottosistema integrale, ma ciò non risulta un problema dal momento che essa include le caratteristiche che interessano al client.

6. Flyweight

Consente di inserire più oggetti nella quantità di RAM disponibile condividendo parti comuni dello stato tra più oggetti invece di mantenere tutti i dati in un oggetto.

Esempio pratico

Supponiamo di aver creato un videogioco, uno sparatutto con una serie di particelle realistiche. Provandolo abbiamo notato che continua a bloccarsi dopo alcuni minuti a causa di RAM insufficiente. Il problema è legato alle particelle in quanto ognuna di queste è rappresentata da un oggetto contenente molti dati. Man mano che si va avanti con la partita, le nuove particelle non entrano più nella RAM causando il crash.

Controllando la classe Particle si nota che i campi Color e Spirit consumano molta più memoria rispetto ad altri, eppure sono dati quasi identici su tutte le particelle. Questi valori costanti prenderanno il nome di stato instrinseco, mentre il resto dello stato dell'oggetto, spesso alterato dall'esterno, è detto stato estrinsico.
Il pattern Flyweight suggerisce di non memorizzare più lo stato estrinseco all'interno dell'oggetto, ma passare questo stato a metodi specifici che si basano su di esso.
Un'oggetto che memorizza solo lo stato instrinseco si chiama Flyweight.

7. Proxy

Consente di fornire un sostituto o un segnaposto per un altro oggetto. Un proxy controlla l'accesso all'oggetto originale, consentendo di eseguire qualcosa prima o dopo che la richiesta arrivi all'oggetto originale.

Esempio pratico

Supponiamo di avere un oggetto massimo che consuma una grande quantità di risorse di sistema di cui abbiamo bisogno di tanto in tanto, ma non sempre.
Una soluzione pigra potrebbe essere creare l'oggetto solo quando è necessario: tutti i client dovrebbero eseguire codice di inizializzazione differito. Questo causerebbe tantissime duplicazioni.

Il pattern Proxy suggerisce di creare una nuova classe proxy con la stessa interfaccia di un oggetto di servizio originale. In questo modo si può aggiornare l'app in modo che passi il proxy a tutti i client dell'oggetto originale. Al ricevimento di una richiesta da parte di un client, il proxy crea un oggetto di servizio reale e delega ad esso tutto il lavoro.
Il vantaggio sta nel fatto che il proxy può essere passato a qualsiasi client che si aspetta un oggetto di servizio reale.

11.5 Pattern Comportamentali

Si riferiscono alla distribuzione delle responsabilità di oggetti correlati tra loro.
Si focalizzano sulle modalità di comunicazione e collaborazione tra oggetti e classi fornendo una soluzione per incapsulare le diverse funzionalità in un oggetto specifico con l'intento di delegare ad esso l'esecuzione del codice vero e proprio.

Esistono due categorie di pattern comportamentali:

- **Behavorial class pattern** (usano l'ereditarietà)
- **Behavorial object pattern** (usano la composizione)

1. Chain of Responsability

Consente di passare le richieste lungo una catena di gestori. Al ricevimento di una richiesta ogni gestore decide di elaborare la richiesta oppure passarla al successivo gestore della catena.

Esempio pratico

Immaginiamo di lavorare ad un sistema di ordini online. Vogliamo limitare l'acceso al sistema così che solo gli utenti autenticati possono creare ordini. Questi controlli devono essere eseguiti in sequenza.
Una volta aggiunti i controlli sequenziali il codice di questi si gonfia sempre di più, inoltre la modifica di un controllo può influire sugli altri.

Il patter Chain of Responsability si basa sulla trasformazione di particolari comportamenti in oggetti standalone chiamati handler, nel nostro esempio ogni controllo dovrebbe essere associato ad una propria classe.
Il pattern suggerisce di collegare questi gestori in una catena dove ogni gestore ha un campo per memorizzare un riferimento al gestore successivo nella catena. Un gestore può decidere di non passare la richiesta ulteriormente nella catena e fermare qualsiasi ulteriore elaborazione.

È fondamentale che tutte le classi degli handler implementino la stessa interfaccia. Inoltre ogni handler concreto dovrebbe preoccuparsi solo di quello seguente che ha un metodo *execute()*, in questo modo si possono comporre catene a runtime senza accoppiare il codice alle loro classi complete.

2. Command

Trasforma una richiesta in un oggetto a se stante che contiene tutte le informazioni sulla richiesta. Tale trasformazione permette di parametrizzare i metodi con diverse richieste, ritardare o mettere in coda l'esecuzione di una richiesta e supportare le operaizoni annullabili.

Esempio concreto

Lavoriamo ad un nuovo text editor. Dobbiamo creare una barra degli strumenti con una serie di pulsanti. Creiamo una classe Button che può essere utilizzata sia per la barra degli strumenti che per altri pulsanti generici. Si suppone che tutti questi pulsanti facciamo cose diverse e quindi dobbiamo capire dove posizionare il codice per i vari gestori.

Una buona gestione si basa sul principio di separazione dei problemi, cioè struttura di una app in strati.
Il pattern Command suggerisce che gli oggetti GUI non dovrebbero inviare queste richieste direttamente, ma dovrebbero estrarre tutti i dettagli della richiesta in una classe command separata, con un metodo che inneschi tale richiesta. I command servono come collegamenti tra le varie GUI e gli oggetti della logica di business. Così la GUI si occupa di attivare il comando.
Dobbiamo poi fare in modo che i comandi implementino la stessa interfaccia che solitamente ha un singolo metodi di esecuzione che non accetta parametri e permette di utilizzare vari comandi con lo stesso mittende della richiesta, senza accoppiarlo a classi concrete di comandi.

Tornando all'editor di testo, dopo aver applicato il pattern Command è sufficiente mettere un singolo campo nella classe base Button che memorizza un riferimento ad un oggetto di comando.

3. Iterator

Permette di attraversare gli elementi di una collezione senza esporre la rappresentazione sottostante.

Esempio pratico

Le collezioni sono uno dei tipi di dati più utilizzati nella programmazione. La maggior parte delle collezioni memorizza i propri elementi in semplici liste. Ma per accedere ad essi serve un metodo veloce che si complica quando abbiamo a che fare con una struttura dati complessa (come quella ad albero) anche perche queste possono essere attraversate in vari modi.
Quindi per soddisfare criterio di attraversamento bisognerebbe implementare algoritmi diversi.

L'idea del patter Iterator è quella di estrarre il comportamento di attraversamento di una collezione in un oggetto chiamato Iteratore.
L'oggetto Iteratore si occupa di implementare l'algoritmo di attraversamento a tutti i dettagli dello stesso, come la posizione corrente, il numero di elementi rimanenti, ecc...
La presenza di tali dati consente di far attraversare la stessa collezione da più iteratori nello stesso momento, quindi quando un iteratore ha concluso il suo attraversamento non restituisce più nulla.
Tutti gli iteratori devono implementare la stessa interfaccia in modo da rendere il codice del client compatibile con qualunque tipo di collezione o con qualsiasi algoritmo di attraversamento.

4. Mediator

Consente di ridurre le interdipendenze tra gli oggetti. Il pattern inserisce un mediatore che si occupa di far comunicare tra loro i vari oggetti solo passando attraverso di lui.

Esempio pratico

Supponiamo di avere una finestra GUI per gestire i profili dei clienti. Questa finestra è composta da vari elementi che possono interagire tra loro, questo gran numero si concretizza in altrettante classi che devono avere riferimenti agli altri elementi della finestra.
Così facendo oltre ad avere un codice caotico non si possono riutilizzare quelle classi in altre parti del codice visto che possiedono riferimenti che potrebbero non servire.

Il pattern Mediator suggerisce di interrompere ogni comunicazione diretta tra i componenti che si desidera rendere indipendenti l'uno dall'altro. I componenti dovranno invece collaborare indirettamente, chiamando uno speciale oggetto mediatore che reindirizza le chiamate ai componenti appropriati, quindi in questo modo i componenti dipenderanno da una singola classe mediatore, evitando i multipli riferimento alle altre classi in ognuno di essi.
Per semplificare questo processo è buona pratica inserire un interfaccia comune a tutti gli elementi e faccia da tramite per comunicare con il mediatore.

5. Memento

Permette di salvare e ripristinare lo stato precedente di un oggetto senza rivelare i dettagli della sua implementazione.

Esempio pratico

Immaginiamo di creare un text editor. Vogliamo consentire che gli utenti annullino le operazioni effettuate sul testo. Decidiamo di usare l'approccio diretto: prima di ogni operazione l'app registra lo stato di tutti gli oggetti e lo salva da qualche parte nella memoria.
Quando un utente vuole tornare indietro di una azione l'app recupera questa istantanea e la usa per ripristinare lo stato di tutti gli oggetti.
Analizzando questo approccio, è immediato rendersi conto del fatto che causerebbero delle problematiche:

1. Servirebbero tantissime chiamate a metodi per ottenere i valori degli attributi quando salviamo l'istantanea.
2. Aggiungendo altre classi bisognerebbe modificare anche le classi responsabili della copia dello stato.
3. Si metterebbero a rischio la sicureza dell'app rendendo accessibili a tutti i dati copiati.

Il pattern Memento delega la creazione degli Snapshot dello stato all'oggetto stesso che deve essere fotografato.
È l'oggetto stesso che si occupa di copiarsi. Il pattern suggerisce di memorizzare, poi, la copia dello stato dell'oggetto in un oggetto speciale chiamato Memento. Il suo contenuto non è accessibile a nessun altro oggetto se non a quello che lo ha prodotto.
Infatti, altri oggetti devono comunicare con i memeto utilizzando un'interfaccia limitata che può permettere di recuperare i metadati dello snapshot, ma non lo stato dell'oggetto originale.

6. Observer

Permette di definire un meccanismo di sottoscrizione per notificare a più oggetti gli eventi che accadono all'oggetto che stanno osservando.

Esempio pratico

Immaginiamo che un cliente sia interessato ad un nuovo prodotto e sia in attesa del suo rilascio. Il cliente potrebbe visitare il negozio ogni giorno per verificare la disponibilità. Ma la maggior parte di questi viaggi sarebbe inutile. In alternativa il negozio potrebbe inviare email a tutti i clienti, ma questo sconvolgerebbe altri clienti non interessati al prodotto.

Il pattern Observer offre una soluzione molto semplice a livello logico: chi è interessato (subscriber) al cambiamento dell'oggetto interessante (detto subject o publisher) si iscrive ad una lista in modo tale che il publisher, cambiando stato, mandi una notifica dell'avvenimento a tutti i subscriver iscritti a una lista. Al livello di implementazione il pattern suggerisce di aggiungere un meccanismo di sottoscrizione alla classe del publisher in modo che i singoli oggetti possano iscriversi o cancellarsi da un flusso di eventi provenienti da quel publisher.
Così, ogni volta che un evento importante accade al publisher individua i suoi subscriber e chiama il metodo di notifica specifico sui loro oggetti.

7. State

Permette ad un oggetto di modificare il suo comportamento quando il suo stato interno cambia.

Esempio pratico

Immaginiamo di avere una classe Document. Un documento può essere in uno di tre stati: Draft, Moderation e Published.
Vogliamo che il metodo *publish()* del documento funziona in modo diverso in base allo stato in cui si trova l'oggetto:

- In Draft sposta il documento alla moderazione.
- In Moderation rende il documento pubblico.
- In Published non fa nulla.

Per implementare questo comportamento dovremmo usare una serie di costrutti *if-elif* ma volendo aggiungere altri stati il codice crescerebbe a dismisura.

Il pattern Stage suggerisce di creare nuove classi per tutti i possibili stati di un oggetto e di estrarre tutti i comportamenti specifici dello stato in queste classi. In questo modo invece di implementare tutti i possibili comportamenti da solo, l'oggetto originale memorizza un riferimento a uno degli oggetti di stato che rappresenta il suo stato corrente e delega il lavoro relativo allo stato di quell'oggetto, il tutto tramite un'interfaccia comune.

8. Strategy

Permette di definire una famiglia di algoritmi, di mettere ciascuno di essi in una classe separata e di rendere i loro oggetti intercambiabili.

Esempio pratico

Supponiamo di sviluppare una app che utilizza moltissime collezioni per memorizzare i dati. Immaginiamo di aver bisogno di ordinare spesso queste collezioni.
Abbiamo a disposizione diversi algoritmi di ordinamento ma ognuno di questi si comporta in modo diverso in base alle situazioni e non necessariamente quello più efficiente nella media è ugualmente il più efficiente in una situazione specifica, quindi dobbiamo trvare l'algoritmo giusto.

Il pattern Strategy suggerisce di prendere una classe che fa qualcosa di specifico in molti modi diversi ed estrarre tutti questi algoritmi in classi separate chiamate Strategie. La classe originale deve avere un campo per memorizzare un riferimento ad una delle strategie in modo da delegare il lavoro ad un oggetto strategia invece di eseguirlo da solo.
Così facendo la classe originale non è responsabile della selezione di un algoritmo ma è il client a passare la strategia desiderata alla classe originale.
Anche qui l'interfaccia generica ha un ruolo centrale, quindi l'interfaccia esponde solo un singolo metodo per l'attivazione dell'algoritmo incapsulato nella strategie selezionata.

9. Template Method

Definisce lo scheletro di un algoritmo nella superclasse ma lascia che le sottoclassi sovrascrivano specifici passi dell'algoritmo senza modificarne la struttura.

Esempio pratico

Immaginiamo di creare un app di data mining che analizza dei documenti aziendali dai vari formati da cui estrarre i dati significativi in un formato uniforme. Si potrebbe creare una classe per ogni formato ma avrebbero un codice simile.
Vogliamo dunque eliminare la duplicazione lasciando intatta la struttura dell'algoritmo.

Il pattern Template Method suggerisce di suddividere un algoritmo in una serie di passi, trasformarli in metodi separati ed inserire una serie di chiamate a questi metodi all'interno di un singolo template method. I passi possono essere astratti o avere un'implementazione predefinita.
Per utilizzare l'algoritmo si presuppone che il client debba fornire la propria sottoclasse, implementare tutti i passi astratti e, se necessario, sovrascrivere gli opzionali.

Ci sono 2 tipi di passi:

- **Astratti**: devono essere implementati da ogni sottoclasse.
- **Opzionali**: hanno qualche implementazione di default ma possono essere sovrascritti.

Esiste un'ulteriore passo detto Hook, opzionale con un corpo vuoto, ovvero un metodo ne implementato ne astratto. Un Template Method funzionerebbe anche se un hook non viene sovrascritto.

10. Visitor

Consente di separare gli algoritmi dagli oggetti su cui operano.

Esempio pratico

Immaginiamo di sviluppare un'app che funzioni con informazioni geografiche strutturate come un grafo, Ogni nodo del grafo rappresenta un'entità della citta. Al livello di codice **tipo di nodo = classe** e **specifico nodo = oggetto**.
Se avessimo la necessità di esportare il tutto in formato XML potremmo pensare di aggiungere un metodo di esportazione ad ogni classe nodo, ma ciò potrebbe causare problemi nel caso volessimo un diverso formato di esportazione.

Il pattern Visitor suggerisce di collocare il nuovo comportamento in una classe separata chiamata Visitor, invece di integrarlo nelle esistenti.
L'oggetto originale viene passato ad uno dei metodi del visitor come argomento, fornendogli l'accesso ai dati contenuti nell'oggetto.
Anche qui si estrae un'interfaccia comune per tutti i visitor per permettere ai modi di lavorare con ogni nuovo visitor introdotto.