

Orale ing del software

Leonardo Ascenzi

January 13, 2026

Contents

1	Significato include/extend negli use case	3
1.1	Include:	3
1.2	Extend:	3
2	Requisiti Software	4
2.1	Requisiti utente:	4
2.2	Requisiti di sistema:	4
2.2.1	Funzionali:	4
2.2.2	Non funzionali:	4
2.2.3	Dominio:	4
2.3	Differenze:	4
3	Modello ciclo di vita	4
3.1	Principali modelli	5
3.1.1	Build & Fix	5
3.1.2	Waterfall	5
3.1.3	Spirale	5
4	Risk Analysis	5
4.1	Fasi principali:	5
5	Architettura software - Architettura oggetti distribuiti	6
6	ORB - Object Request Broker	6
6.1	Funzionamento	6
6.2	Vantaggi	6

7	Relazione tra prodotto e costo di produzione	6
7.1	Fattori principali	7
7.2	Equilibrio costi	7
8	Service Oriented Architecture	7
9	Secondo quali attività va declinata la fase di testing?	8
10	Come si chiama il testing di validazione per i software a contratto?	9
11	Ciclo di vita del Software	9
11.1	Definizione Formale	9
11.2	Stadi	9
11.2.1	Stadio 1: Sviluppo	9
11.2.2	Stadio 2: Manutenzione	10
11.2.3	Stadio 3: Dismissione	10
11.3	Fasi dello Stadio di Sviluppo	10
11.3.1	1. Requisiti	10
11.3.2	2. Specifiche	10
11.3.3	3. Pianificazione	11
11.3.4	4. Progetto	11
11.3.5	5. Codifica	11
11.3.6	6. Integrazione	11
12	Qual é la manutenzione più frequentemente utilizzata?	12
12.1	Altri tipi di manutenzioni	12
13	Regola 10-90	12
14	Stima durata progetto	12
15	Organizzazione modello di qualità del Software	13
15.1	Principi e struttura secondo IEEE 1061	13
15.1.1	1. Attività di Revisione	13
15.1.2	2. Attività di Transizione	13
15.1.3	3. Attività di Operazioni	14
15.1.4	Attributi di qualità	14
15.1.5	Processo iterativo di valutazione	15
16	Diagramma UML per interazione tra oggetti	15

17 COCOMO	15
17.1 Formula di COCOMO	16
17.1.1 Esempio	16
18 Design Pattern	17
18.1 Cosa sono i Design Patterns? a cosa servono?	17
18.2 Come è fatta la struttura di un Design Patterns?	17
18.3 Lista dei Design Pattern	18
18.3.1 Abstract Factory	18
18.3.2 Factory Method	19
18.3.3 Adapter	19
18.3.4 Composite	19
18.3.5 Decorator	20
18.3.6 Observer	20
18.3.7 Template Method	21
18.3.8 Strategy	21

1 Significato include/extend negli use case

Il diagramma degli use case permettere di rappresentare graficamente le interazioni tra attori e sistema, specificando quali casi d'uso gli attori possono attivare.

1.1 Include:

Indica che un caso d'uso include **obbligatoriamente** un altro caso d'uso per completare la sua funzionalità. Utilizzato quando il comportamento di un caso d'uso é riutilizzabile per altri casi d'uso. Caso d'uso principale richiede l'esecuzione dei casi d'uso inclusi.

1.2 Extend:

Indica che un caso d'uso può estendere **opzionalmente** un altro caso d'uso, aggiungendo funzionalità alternative o opzionali. Utilizzato per rappresentare casi che si verificano solo in determinate condizioni. L'attivazione del caso d'uso esteso dipende da una determinata condizione.

2 Requisiti Software

I requisiti del software sono la **descrizione dei servizi** che un sistema software deve fornire, insieme ai vincoli da rispettare sia in fase di sviluppo che durante la fase di operatività.

2.1 Requisiti utente:

Descrizione in **linguaggio naturale** dei **servizi** che il sistema deve fornire e dei **vincoli operativi**.

2.2 Requisiti di sistema:

Specificati mediante la **stesura di un documento strutturato** che descrive in modo dettagliato i **servizi che il sistema software deve fornire**. Si dividono in **3** categorie:

2.2.1 Funzionali:

Definiscono le funzionalità del sistema, ovvero i servizi offerti, le reazioni a specifici input e il comportamento in diverse situazioni operative.

2.2.2 Non funzionali:

Descrivono le qualità del sistema e del processo di sviluppo, come efficienza, affidabilità, sicurezza ed altre caratteristiche esterne.

2.2.3 Dominio:

Derivano dal contesto applicativo del sistema.

2.3 Differenze:

I requisiti funzionali in breve **definiscono** tutte le **operazioni** che il **sistema deve eseguire**. Gli altri **definiscono** tutte le **funzionalità** che il **sistema deve rispettare**.

3 Modello ciclo di vita

Definisce le fasi attraverso cui il prodotto software evolve, dalla raccolta dei requisiti alla dismissione e l'ordine in cui tali fasi vengono eseguite. La scelta

del modello dipende da fattori come la natura e la maturità dell'organizzazione, le tecnologie adottate e i vincoli del cliente.

3.1 Principali modelli

3.1.1 Build & Fix

Modello privo di struttura formale; il prodotto viene sviluppato e poi modificato fino a soddisfare le esigenze del cliente.

3.1.2 Waterfall

Modello sequenziale a fasi successive con verifica alla fine di ogni fase; adatto a progetti con requisiti ben definiti.

3.1.3 Spirale

Modello iterativo che combina l'approccio del Waterfall con una forte enfasi sull'analisi dei rischi; ideale per progetti complessi e soggetti a cambiamenti nei requisiti.

4 Risk Analysis

È il processo di identificazione, valutazione e gestione dei potenziali problemi che possono compromettere il successo di un progetto software.

4.1 Fasi principali:

1. Identificazione dei rischi: Individuare tutte le minacce che possono influire negativamente sul progetto.
2. Valutazione dei rischi: Stimare l'impatto e la probabilità di ogni rischio.
3. Prioritizzazione: Stabilire quali rischi gestire per primi.
4. Pianificazione delle mitigazioni: Definire le azioni per ridurre la probabilità o l'impatto dei rischi.
5. Monitoraggio continuo: Assicurarsi che i rischi siano monitorati durante tutto il ciclo di vita del software.

5 Architettura software - Architettura oggetti distribuiti

L'architettura di sistema definisce la struttura dei componenti del sistema software, insieme alle relazioni tra questi ultimi. L'architettura ad oggetti distribuiti é un'architettura di sistema che non fa distinzioni tra client/server, infatti ogni oggetto distribuito può fungere sia da client che da server. La comunicazione remota fra gli oggetti é resa trasparente usando middleware basati sul concetto di software bus. Le applicazioni basate su questa architettura consistono in un insieme di oggetti che sono eseguiti su piattaforme distribuite ed eterogenee, e comunicano tramite invocazioni remote dei metodi.

6 ORB - Object Request Broker

É un componente middleware che abilita la comunicazione trasparente tra oggetti distribuiti su rete, indipendentemente da posizione fisica, linguaggio di programmazione o sistema operativo É il cuore delle architetture ad oggetti distribuiti, come CORBA.

6.1 Funzionamento

Intermediazione Richieste: il client invoca un metodo su un oggetto remoto; ORB localizza il server, inoltra la richiesta e restituisce il risultato. **Gestione Dati:** si occupa di serializzazione e deserializzazione dei parametri, oltre ai protocolli di rete **Trasparenze Fornite:** location transparency e implementation transparency.

6.2 Vantaggi

Permette agli sviluppatori di concentrarsi sulla logica applicativa, astruendo la complessità della distribuzione.

7 Relazione tra prodotto e costo di produzione

La relazione tra prodotto software e costo di produzione é prevalentemente positiva: maggiore é la dimensione, la complessità o la qualità del prodotto, maggiore é il costo iniziale di sviluppo, ma con impatti variabili sui costi a lungo termine.

7.1 Fattori principali

Dimensione del prodotto: La complessità e le funzionalità aumentano direttamente lo sforzo e i costi, spesso in modo non lineare. **Qualità:** Investire in qualità eleva i costi iniziali, ma riduce quelli futuri di manutenzione e correzioni. **Processi di sviluppo:** Metodologie efficienti e automazione abbassano i costi mantenendo la qualità

7.2 Equilibrio costi

Scalabilità e modularità: Favoriscono costi minori in fase di aggiornamenti e manutenzioni, bilanciando l'investimento iniziale. **Sintesi:** Esiste un trade-off tra costi di produzione upfront e spese post-rilascio, ottimizzabile con stime accurate basate su dimensioni funzionali.

8 Service Oriented Architecture

La SOA è un'architettura software distribuita che consiste in molteplici servizi. I Servizi sono distribuiti in maniera tale da poter essere eseguiti su nodi differenti con differenti service provider. L'obiettivo è quello di sviluppare applicazioni software composte da servizi distribuiti, in modo tale che i singoli servizi possano essere eseguiti su più piattaforme differenti e implementati con differenti linguaggi di programmazione. All'interno delle SOA troviamo ORB che permette la comunicazione tra i client e i servizi distribuiti.

Il fulcro dell'implementazione SOA sono i web services, e sfruttano vari protocolli per far comunicare i servizi fra di loro.

Tra questi troviamo: **SOAP:** protocollo basato su XML e HTML che permette lo scambio di informazioni in un sistema distribuito. Utilizza protocolli di trasporto come HTTP, SMTP. **REST:** architettura che utilizza i metodi HTTP per operazioni CRUD.

Oltre che ai web services, le SOA utilizzano altri servizi, tra cui: **UDDI:** aiuta a localizzare e identificare i servizi presenti in una rete. **WSDL:** Linguaggio basato su XML utilizzato per descrivere le interfacce dei servizi in un'architettura di

9 Secondo quali attività va declinata la fase di testing?

La fase di testing é articolata in una serie di attività che assicurano la soddisfazione dei requisiti definiti. Queste attività sono:

1. Pianificazione del testing:

- Definizione degli obiettivi del testing.
- Identificazione delle risorse necessarie.
- Pianificazione temporale e stima dei costi.

2. Progettazione dei Test Case:

- Creazione di scenari di test basati sui requisiti funzionali e non funzionali.
- Identificazione delle versioni software da testare.

3. Preparazione dell'ambiente di test:

- Configurazione degli ambienti (Hardware, Software, Reti).
- Installazione delle versioni del software da testare.

4. Esecuzione dei test:

- Esecuzione dei casi di test progettati.
- Registrazione dei risultati (Successo o Fallimento) e documentazione delle anomalie.

5. Analisi dei risultati:

- Confronto tra risultati ottenuti e quelli attesi.
- Identificazione e classificazione dei difetti.

6. Risoluzione dei problemi:

- Collaborazione con il team di sviluppo per risolvere i difetti identificati.
- Verifica delle correzioni.

7. Test di regressione:

- Assicurarsi che le modifiche al codice non abbiano introdotto nuovi errori.

8. Valutazione finale e report:

- Verifica se il prodotto soddisfa i criteri di accettazione.
- Redazione di un report con le conclusioni del testing.

10 Come si chiama il testing di validazione per i software a contratto?

Il testing di validazione per i software a contratto é chiamato **Acceptance Testing**.

- **Definizione:** É l'ultima fase del ciclo di testing, in cui il cliente o l'utente finale verifica che il software soddisfi i requisiti contrattuali e sia pronto per il rilascio.
- **Scopo:** Validare che il software sia conforme alle specifiche contrattuali e garantire la piena utilizzabilità nel contesto operativo del cliente.

11 Ciclo di vita del Software

11.1 Definizione Formale

Il ciclo di vita di un software é il quadro strutturale che organizza in fasi, attività e compiti tutti i processi di sviluppo, esercizio e manutenzione di un prodotto software, dalla definizione dei requisiti fino alla dismissione del sistema. É suddiviso in 3 stadi, il primo stadio a sua volta é composto da 6 fasi.

11.2 Stadi

11.2.1 Stadio 1: Sviluppo

- Requisiti
- Specifiche
- Pianificazione
- Progetto

- Codifica
- Integrazione

11.2.2 Stadio 2: Manutenzione

Copre circa il 60% dei costi del ciclo di vita

11.2.3 Stadio 3: Dismissione

11.3 Fasi dello Stadio di Sviluppo

11.3.1 1. Requisiti

Obiettivo: Identificare e raccogliere le esigenze degli stakeholder. **Descrizione:** In questa fase si effettua la raccolta, l'analisi e la documentazione dettagliata delle esigenze degli utenti e degli stakeholder, per definire cosa il sistema deve fare e i suoi vincoli operativi. **Attività Principali:**

- Interviste con utenti e clienti.
- Analisi delle esigenze funzionali.
- Identificazione dei requisiti non funzionali.
- Creazione di una lista iniziale di requisiti.

11.3.2 2. Specifiche

Obiettivo: Documentare e dettagliare i requisiti raccolti nella fase precedente. **Descrizione:** Approfondisce i requisiti in specifiche tecniche dettagliate, producendo documenti come **SRS (Software Require Specification)** o **SFS (Software Functional Specification)** che serviranno nelle fasi successive. **Attività Principali:**

- Analisi di fattibilità tecnica ed economica.
- Stesura dei documenti di specifiche.
- Validazione dei requisiti con il cliente per verificare che siano corretti.

11.3.3 3. Pianificazione

Obiettivo: Organizzare il lavoro per il completamento del progetto. **Descrizione:** Si definisce il piano del progetto: stime di costi/risorse/tempi, allocazione team, scelta strumenti/metodologie, identificazione rischi e definizione milestone. Produce il **Project Plan** con **Gant**, budget e criteri di monitoraggio. È cruciale per allineare stakeholder e gestire scope.

11.3.4 4. Progetto

Obiettivo: Progettare l'architettura e i dettagli tecnici del software. **Descrizione:** Si definisce la struttura del sistema e le componenti, fornendo un modello chiaro da seguire nella fase di sviluppo. **Attività Principali:**

- Progettazione dell'architettura.
- Creazione dei diagrammi UML, diagrammi di flusso e design delle interfacce.
- Progettazione di database e definizione delle entità.
- Redazione del piano di test iniziale per assicurarsi che il progetto sia verificabile.

11.3.5 5. Codifica

Obiettivo: Tradurre il design in un software funzionante. **Descrizione:** In questa fase gli sviluppatori scrivono il codice sorgente seguendo le specifiche di progetto, standard di coding e best practice. Include unit testing durante lo sviluppo e commit in repository git. L'importanza di questa fase è per l'appunto la trasformazione da design a eseguibile, con enfasi su qualità e leggibilità.

11.3.6 6. Integrazione

Obiettivo: Collegare i diversi moduli e testare il sistema completo. **Descrizione:** Si assemblano i moduli/test unitari in un sistema coerente, verificando integrazioni tramite test di integrazione. Identifica e risolve difetti di interfaccia. Culmina in un prototipo parziale o completo pronto per essere testato.

12 Qual é la manutenzione piú frequentemente utilizzata?

La manutenzione piú frequentemente utilizzata é quella **correttiva**. La manutenzione correttiva é l'insieme delle attività volte a correggere difetti, errori o malfunzionamenti riscontrati nel software dopo il suo rilascio in produzione, ripristinando la funzionalità conforme alle specifiche originali.

12.1 Altri tipi di manutenzioni

1. Adattiva Modifiche al software per adattarlo ai cambiamenti dell'ambiente operativo.
2. Perfettiva Migliora le prestazioni o aggiunge nuove funzionalità per soddisfare le richieste degli utenti.
3. Preventiva Effettuare modifiche per ridurre rischi di guasto o per prevenire dei problemi futuri.

13 Regola 10-90

La regola 10-90 é una variante della regola 90-90, un aforismo umoristico dell'ingegneria del software. Questa regola afferma che: "Solo il 10% del codice critico consuma il 90% delle risorse di esecuzione o sviluppo". Il 10% del codice in questione si definisce **core** del programma.

14 Stima durata progetto

Per stimare la durata totale di un progetto software partendo dal documento di specifica, é necessario utilizzare tecniche di pianificazione e stima che si basano su diversi fattori come la dimensione stimata del prodotto, le risorse disponibili e i processi di sviluppo. La prima fase della stima é quella stimare le dimensioni totali del software. usando tecniche note che vedremo avanti. Successivamente si stima la durata:

1. Modello algoritmico **COCOMO**:
 - Stimando effort e poi tempo
2. Usando metodi empirici:

- Suddividere il progetto in attività specifiche e stimare la durata di ogni attività e successivamente sommare tutte queste stime per averne una complessiva.

15 Organizzazione modello di qualità del Software

Il modello di qualità del software si struttura principalmente attorno allo standard **IEEE 1061**, che fornisce un framework sistematico per definire, misurare e valutare la qualità attraverso metriche quantitative legate a obiettivi specifici del progetto.

15.1 Principi e struttura secondo IEEE 1061

Lo standard organizza la qualità del software identificando prima gli obiettivi di qualità rilevanti per il contesto. Successivamente, associa a questi obiettivi attributi di qualità misurabili e metriche per valutarli. Un elemento centrale sono gli indici di qualità, che aggregano più metriche in indicatori sintetici per monitorare le prestazioni complessive. Questi indici si dividono in tre gruppi principali, allineati al modello McCall.

15.1.1 1. Attività di Revisione

- **Manutenibilità:** effort richiesto per individuare e correggere un errore in un programma operativo.
- **Flessibilità:** effort per modificare un prodotto software già in uso.
- **Testabilità:** effort per testare il software e verificarne la correttezza funzionale.

15.1.2 2. Attività di Transizione

- **Portabilità:** effort per trasferire il software ad un nuovo ambiente hardware/software.
- **Riusabilità:** grado in cui parti del software possono essere riutilizzate in altre applicazioni.
- **Interoperabilità:** effort per integrare il software con altri sistemi.
- **Evolutività:** effort per aggiornarlo con nuovi requisiti.

15.1.3 3. Attività di Operazioni

- **Correttezza:** misura in cui il software soddisfa specifiche e obiettivi utente.
- **Affidabilità:** probabilità che svolga le funzioni previste con la precisione richiesta.
- **Efficienza:** risorse di calcolo e codice necessarie per eseguire una funzione.
- **Integrità:** protezione da accessi non autorizzati a software o dati.
- **Usabilità:** effort per imparare, usare, preparare input e interpretare output.

15.1.4 Attributi di qualità

Oltre agli indici, **IEEE 1061** elenca attributi fondamentali, come:

- **Complessità:** comprensibilità e verificabilità degli elementi e delle loro interazioni.
- **Precisione:** accuratezza dei calcoli e dei risultati.
- **Completezza:** implementazione esaustiva delle funzionalità richieste.
- **Coerenza:** uniformità di design, implementazione e notazioni.
- **Tolleranza:** capacità di operare in condizioni avverse.
- **Tracciabilità:** legami chiari tra elementi del processo di sviluppo.
- **Espandibilità:** possibilità di ampliare memorie o funzioni.
- **Generalità:** ampiezza delle applicazioni potenziali.
- **Modularità:** indipendenza dei moduli.
- **Autodocumentazione:** presenza di documentazione inline.

Questi attributi sono mappati su metriche concrete

15.1.5 Processo iterativo di valutazione

IEEE 1061 definisce un ciclo chiaro:

1. **Definizione degli obiettivi:** contestualizzati al progetto.
2. **Identificazione degli attributi:** selezione di quelli prioritari.
3. **Scelta metriche:** quantitative e rilevanti per ciascun attributo.
4. **Misurazione:** calcolo e confronto con benchmark o target.
5. **Analisi e Feedback:** miglioramento continuo basato sui risultati.

Questa struttura garantisce che la qualità sia oggettiva, misurabile e allineata agli stakeholder, riducendo soggettività e rischi.

16 Diagramma UML per interazione tra oggetti

Per specificare l'interazione tra oggetti, usando la struttura dei diagrammi UML, possiamo usare il **Sequence Diagram**. Un altro diagramma UML che permette di fare questa cosa si chiama **Collaboration Diagram**. Questo diagramma è una rappresentazione equivalente del sequence diagram, infatti a partire da uno possiamo generare l'altro e viceversa. Il funzionamento del Collaboration Diagram è identico a quello del sequence diagram, cambia solo:

- Il Sequence Diagram descrive lo scambio di messaggi tra oggetti in ordine temporale, mentre il Collaboration Diagram descrive lo scambio di messaggi tra oggetti mediante relazioni.
- I Sequence vengono usati in fase di OOA, mentre i Collaboration in fase di OOD

17 COCOMO

Modello algoritmico introdotto da Boehm per determinare il valore dell'effort in base alla dimensione del prodotto. Il valore dell'effort viene successivamente utilizzato per determinare durata e costi di sviluppo. COCOMO comprende 3 modelli:

- **Basic:** Per stime iniziali.

- **Intermediate:** Usato dopo aver suddiviso il sistema in sottosistemi.
 - **Advanced:** Usato dopo aver suddiviso in moduli ciascun sottosistema
- La stima dell'effort viene effettuata a partire da:
- Stima delle dimensioni del progetto in **KLOC**
 - Stima del modo di sviluppo del prodotto, che misura il livello intrinseco di difficoltà nello sviluppo, tra:
 - **Organic:** Per prodotti di piccole dimensioni.
 - **Semi-Detached:** Per prodotti di dimensioni intermedie.
 - **Embedded:** Per prodotti complessi.

17.1 Formula di COCOMO

Stima dell'effort nominale:

$$EffortNom = a \cdot (KLOC)^b \quad (1)$$

a, b sono coefficienti specifici per ciascuna categoria di progetto.

Stima effort:

$$Effort = EffortNom \cdot C \quad (2)$$

Dove C è il costo driver multiplier, vedi sotto.

Stima del tempo:

$$Time = c \cdot (Effort)^d \quad (3)$$

c, d come a, b

17.1.1 Esempio

MM = uomo/mesi, quante persone per mese devono lavorare al progetto
Modello intermediate, modo organic:

- **Passo 1:** Determinare l'effort nominale con la formula (1)

$$3.2 \cdot (33)^{1.05}$$

- **Passo 2:** Ottenere la stima dell'effort applicando un fattore moltiplicativo C (Cost Driver Multiplier) basato su x cost drivers. Usare formula (2)

$$126 \cdot 1.15 = 145MM$$

- **Passo3:** Stima del tempo della consegna. USare formula (3)

$$T = 2.5 \cdot 145^{0.38} \sim 16.56$$

18 Design Pattern

18.1 Cosa sono i Design Patterns? a cosa servono?

Uno dei compiti più difficili per un progettista di software è individuare un insieme di oggetti ben definiti, riusabili il più possibile e con relazioni e gerarchie chiare ed efficaci. I Design Patterns sono proprio la soluzione a questo problema. Questi trovano il fulcro della soluzione ad un problema ricorrente e fanno in modo che possa essere riutilizzata infinite volte anche non applicandola necessariamente nello stesso modo. In particolare essi creano un linguaggio condiviso tra gli sviluppatori in modo tale da offrire soluzioni a problemi che ritornano spesso nello sviluppo software e creano un codice strutturato in modo corretto. Inoltre capitalizzano l'esperienza della progettazione OO e ne favoriscono il riuso. Le soluzioni che propongono non sono mai già note ma allo stesso tempo non sono neanche soluzioni universali e quindi non sono sempre applicabili a qualsiasi problema.

I Pattern si distinguono in tre categorie principali:

- **Creazionali:** Riguardano la creazione degli oggetti.
- **Strutturali:** Si concentrano sull'organizzazione di classi e oggetti.
- **Comportamentali:** Descrivono come gli oggetti interagiscono e come si distribuiscono le responsabilità.

18.2 Come è fatta la struttura di un Design Patterns?

Un Design Patterns ha la seguente struttura: Per prima cosa troviamo il nome e la classificazione che illustrano l'essenza, lo scopo e il raggio d'azione del Pattern. In seguito troviamo la motivazione ovvero una descrizione del problema e dello scenario al quale bisogna applicare il relativo Pattern. In seguito troviamo l'applicazione del Pattern e subito dopo la struttura cioè una descrizione grafica della configurazione degli elementi che risolvono il problema. Troviamo anche le classi, gli oggetti che ne fanno parte e i risultati dopo l'applicazione di tale Pattern (conseguenze). Sono presenti poi alcune tecniche e consigli per applicare il Pattern (implementazione). Infine troviamo dei codici d'esempio che illustrano come implementare il Pattern in un certo linguaggio, degli esempi di applicazioni reali che lo utilizzano e altri Pattern che sono collegati a questo.

- **Nome e Classificazione:** Il nome illustra l'essenza di un Pattern; la classificazione lo identifica in termini di scopo e raggio d'azione.

- **Motivazione:** Scenario che descrive in modo astratto il problema al quale applicare il Pattern.
- **Applicabilità:** Applicazioni del Pattern.
- **Struttura:** Descrive graficamente la configurazione di elementi che risolvono il problema.
- **Partecipanti:** Classi ed oggetti che fanno parte del Pattern.
- **Conseguenze:** Risultati dopo l'applicazione del Pattern.
- **Implementazione:** Tecniche e suggerimenti per applicare il Pattern.
- **Codice di esempio:** Frammenti di codice che illustrano come implementare in un certo linguaggio di programmazione il Pattern.
- **Usi noti:** Esempi di applicazioni in sistemi reali.
- **Pattern collegati:** Altri Pattern.

18.3 Lista dei Design Pattern

18.3.1 Abstract Factory

L'Abstract Factory è un design pattern creazionale che fornisce un'interfaccia per creare famiglie di oggetti correlati senza specificare le loro classi concrete. Ogni factory concreta produce un'intera suite di prodotti coerenti tra loro.

1. Struttura

- **AbstractFactory:** Dichiarare i metodi per creare ogni tipo di prodotto della famiglia.
- **Factory concrete:** Implementano tutti i metodi della Abstract Factory ritornando prodotti specifici della famiglia.
- **Interfacce Astratte:** Per ogni tipo di prodotto.
- **Client:** Usa la factory per ottenere intere famiglie di prodotti coerenti.

18.3.2 Factory Method

Il Factory Method è un design pattern creazionale che definisce un'interfaccia per creare oggetti, ma delega alle sottoclassi la responsabilità di decidere quale classe concreta istanziare.

1. Struttura

- **Creator:** Classe che definisce il metodo `factoryMethod()`.
- **ConcreteCreator:** Sottoclasse che implementa il metodo `factoryMethod()`.
- **ConcreteProduct:** Oggetto concreto creato dal metodo `factoryMethod()`.
- **Product:** Rispettiva classi di `ConcreteProduct`.

18.3.3 Adapter

L'Adapter è un design pattern strutturale che consente a due interfacce incompatibili di lavorare insieme, permettendo a oggetti con interfacce diverse di collaborare senza modificare il loro codice sorgente.

1. Struttura

- **Target Interface:** L'interfaccia che il client si aspetta.
- **Adaptee:** La classe esistente con interfaccia incompatibile che deve essere utilizzata.
- **Adapter:** Implementa il Target e "wrappa" l'adaptee, traducendo le chiamate da target ad adaptee.
- **Client:** Usa l'adapter senza sapere che dietro c'è l'adaptee.

2. Approcci Implementativi

- **Class Adapter:** Usa ereditarietà.
- **Object Adapter:** Usa composizione.

18.3.4 Composite

Il Composite è un design pattern strutturale che consente di comporre oggetti in strutture ad albero per rappresentare gerarchie parte-tutto. Permette ai client di trattare oggetti individuali e composizioni di oggetti allo stesso modo, offrendo un'interfaccia uniforme.

1. Struttura

- **Component:** Definisce l'interfaccia comune per foglie e composti.
- **Leaf:** Rappresenta gli elementi finali della gerarchia.
- **Composite:** Rappresenta i nodi interni che contengono altri componenti.
- **Client:** Lavora solo con interfacce Component, senza sapere se sta lavorando con foglia o composto

18.3.5 Decorator

Il Decorator è un design pattern strutturale che consente di aggiungere responsabilità a un oggetto dinamicamente, offrendo un'alternativa flessibile all'ereditarietà per estendere la funzionalità. Mantiene la stessa interfaccia dell'oggetto originale mentre aggiunge nuovi comportamenti.

1. Struttura

- **Component:** Definisce l'interfaccia per gli oggetti a cui aggiungere responsabilità.
- **ConcreteComponent:** L'oggetto base a cui aggiungiamo funzionalità.
- **Decorator:** Implementa Component e ottiene un'istanza di component.
- **ConcreteDecorator:** Estende Decorator e aggiunge specifiche responsabilità.
- **Client:** Lavora con l'interfaccia Component senza sapere se sta usando un componente base o un decorator.

18.3.6 Observer

L'Observer è un design pattern comportamentale che definisce una dipendenza uno-a-molti tra oggetti, in modo che quando un oggetto cambia stato, tutti gli oggetti dipendenti vengono notificati automaticamente e aggiornati. Implementa un meccanismo di sottoscrizione/notifica.

18.3.7 Template Method

Obiettivo: Ha lo scopo di definire la struttura di un algoritmo, delegando alle sottoclassi l'implementazione di passaggi specifici. La classe base decide come è organizzato l'algoritmo, le sottoclassi decidono come implementare i passaggi variabili

Si usa:

- Quando implementiamo algoritmi che condividono la stessa struttura ma differiscono in alcuni dettagli
- Quando si vuole evitare duplicazioni di passaggi perché ci sono comportamenti comuni (vengono inseriti nel template)

La classe base (AbstractClass) contiene:

- `templateMethod`: definisce la sequenza di passi dell'algoritmo
- Metodi astratti che le sottoclassi devono implementare
- Metodi hook che hanno un comportamento standard, ma le sottoclassi possono anche ridefinire. Il template Method crea una struttura di controllo invertito dove è la classe padre che richiama le operazioni ridefinite nelle sottoclassi.

Simile al factory method perché invoca metodi astratti tramite interfaccia e l'implementazione dei metodi rimandata a classi concrete. Si utilizzano però in problemi diversi:

- Il template method è il metodo che invoca i metodi astratti per generalizzare un algoritmo
- Il factory method è un metodo astratto che deve creare e restituire l'istanza di classe concreta per sganciare il cliente dalla scelta del tipo specifico

18.3.8 Strategy

Obiettivo: Ha lo scopo di definire ed incapsulare una famiglia di algoritmi in modo da renderli intercambiabili indipendentemente dal client che li usa. Pensiamo alla classe degli algoritmi di ordinamento, in cui ne esistono diversi (BucketSort, QuickSort ecc). Costruiamo un'applicazione che li supporti tutti e che permette una scelta rapida dell'algoritmo. Molte classi differiscono solo per il comportamento. Il Pattern **strategy** offre un modo per avere un

interfaccia comune. Questo Pattern è applicabile quando sono necessarie più varianti di uno stesso algoritmo, in base al tipo di dato in ingresso o a delle condizioni operative. La strategy elimina i blocchi condizionali necessari se tutti i comportamenti fossero in un'unica classe, ma i client devono conoscere le diverse strategie.