



Analisi e Progettazione del Software

Riassunto Appunti di Teoria

A.A. 2022/2023

➡ Parole Chiave

#00C89F

👉 Introduzione

- ➡ Processo Software
- ➡ UML

- ➡ Analisi

- ➡ Progettazione

👉 Processi Software

- ➡ Cascata
- ➡ Agile
- ➡ Sprint

- ➡ Iterazione
- ➡ Scrum
- ➡ Velocity

- ➡ Unified Process
- ➡ Backlog

👉 Ideazione

- ➡ Ideazione
- ➡ Obiettivi
- ➡ Responsabilità

- ➡ Requisiti
- ➡ Caso d'Uso

- ➡ Attori
- ➡ Scenario

👉 Elaborazione

- ➡ Elaborazione
- ➡ Classi Concettuali
- ➡ Contratti
- ➡ Logica Applicativa
- ➡ Diagramma di Sequenza
- ➡ Diagramma delle Attività

- ➡ Analisi OO
- ➡ Oggetti
- ➡ Architettura Logica
- ➡ Progettazione OO
- ➡ Di. Classi Software
- ➡ RDD

- ➡ Modello di Dominio
- ➡ SSD
- ➡ Package
- ➡ Messaggi
- ➡ Macchine a Stati
- ➡ Visibilità

👉 Pattern GRASP

- ➡ Pattern GRASP
- ➡ Low Coupling
- ➡ Pure Fabrication
- ➡ Protected Variations

- ➡ Creator
- ➡ Controller
- ➡ Polymorphism

- ➡ Information Expert
- ➡ High Cohesion
- ➡ Indirection

👉 Design Pattern GoF

- Design Pattern GoF
- Singleton
- Composite
- Adapter
- Strategy
- Facade
- Factory
- Observer

👉 Testing

- TDD
- Refactoring
- OO Abusers
- Couplers
- Test di Unità
- Code Smell
- Change Preventers
- Fixture
- Bloaters
- Dispensables

➡ Documenti ed Elaborati

👉 Ideazione

- **Visione e studio economico:** sommario del progetto, obiettivi e vincoli ad alto livello
- **Modello dei Casi d'Uso:** requisiti funzionali del progetto
- **Specifiche supplementari:** requisiti non funzionali rilevanti
- **Glossario:** dizionario dei dati e terminologia chiave del dominio
- **Piano di gestione dei rischi:** descrizione dei rischi e idee per gestirli
- **Prototipi di proof e concept:** documenti per chiarire la visione e validare le soluzioni
- **Piano dell'iterazione:** piano della prima iterazione di elaborazione
- **Piano delle fasi e sviluppo software:** piano approssimativo delle fasi successive
- **Scenario di sviluppo:** descrizione della personalizzazione dell'ambiente per il progetto

👉 Elaborazione

- **Modello di Dominio:** diagramma per visualizzare i concetti di dominio
- **Modello di Progetto:** progettazione logica, Classi Software, Packaged, Interazioni...
- **Documento dell'Architettura:** riassunto delle viste dell'architettura del progetto
- **Modello dei dati:** informazioni sulla base di dati e il salvataggio/recupero dei dati
- **Storyboard dei Casi d'Uso e UI:** descrizione dell'interfaccia utente, dell'accessibilità...
- **Documenti raffinati:** Modello dei Casi d'Uso, SSD, Contratti, Specifiche e Glossario

➡ Introduzione

➡ Software

Un **Software** è un programma per computer + la sua documentazione. Le sue caratteristiche principali sono:

[Manutenibilità Fidatezza Efficienza Accettabilità]

L'**ingegneria del software** è una disciplina, insieme di **teorie e metodi**, per creare del software di buona qualità e per **gestire la sua progettazione**.

Un **Processo Software** è un approccio disciplinato per gestire tutto il **ciclo di vita** del software. Definiscono:

[Ruoli Attività Organizzazione Temporale Metodologie]

Esistono attività **fondamentali** presenti in ogni processo software:

[Requisiti Analisi Progettazione Implementazione Validazione]
[Rilascio Manutenzione Gestione del Progetto]

➡ Analisi e Progettazione

Analisi: comprendere i problemi e i requisiti

[⇒ mod. di dominio]

Progettazione: ideare la soluzione ai problemi e rientrare nei requisiti

[⇒ classi software]

➡ UML

UML è il principale linguaggio per la modellazione visuale. Si possono rappresentare sia le componenti **statiche** [⇒ classi/oggetti e relazioni] sia **dinamiche** [⇒ SD, AD, SM]

Esistono diversi livelli di dettaglio per UML (*dalla meno dettagliata alla più dettagliata*):

[⇒ Abbozzo Progetto Linguaggio di Programmazione]

➡ Processi Software

👉 Processo Software a Cascata

Il processo software a **cascata** (o **sequenziale**) definisce delle attività da eseguire in sequenza una dopo l'altra:

[Definizione Requisiti Design Implementazione Integrazione Uso e Manutenzione]

È un metodo **poco flessibile** a cambiamenti e poco efficace, richiede **requisiti stabili e fissi**

👉 Processi Software Iterativi - Incrementali - Evolutivi

L'approccio iterativo a un processo software prevede di avere più **iterazioni** di durata fissa, in ognuna delle quali si svolgono **tutte le attività** e si produce un manufatto, che sarà la base per l'interazione successiva. **I requisiti si definiscono e stabilizzano nel tempo**

Ogni **iterazione** (2-6 settimane) ha un **sottoinsieme** di requisiti **fissi**

👉 Unified Process

Unified Process [UP] è un processo software i.i.e. per software **a oggetti**. Si basa su:

[Casi d'Uso + Rischi → Architettura]

I suoi punti fondamentali sono:

- Affrontare i **rischi maggiori** nelle **prime iterazioni**
- Creare un'**architettura coesa**
- **Coinvolgere** di continuo l'**utente**

Vengono individuate 4 fasi, ognuna delle quali ha più iterazioni:

[Ideazione Elaborazione Costruzione Transizione]

In ogni iterazione si completa un **mini-progetto**, la cui natura è determinata dal numero di iterazioni precedenti ↳ all'inizio ci si interesserà di più a livello astratto (progettazione), nelle iterazioni più avanzata invece di implementazione e testing

👉 Sviluppo Agile

Uno sviluppo iterativo **agile** si adatta **velocemente** e **facilmente** ai **cambiamenti**. Si valorizza la **semplicità**, la **chiarezza di comunicazione** e l'**incrementalità** delle consegne

Unified Process agile **diminuisce** il numero di **attività ed elaborati** - non **definisce** tutti i **requisiti** nelle prime iterazioni, ma in modo **incrementale** - utilizza **UML** in parte minore, **solo dove necessario**

👉 Scrum

Scrum è un metodo agile per **massimizzare la qualità** del software e **minimizzare i tempi** di rilascio. Si occupa di organizzazione e **gestione del tempo**. Ogni giorno avviene uno **scrum** [⇨ *meeting*] dove si discutono gli **obiettivi giornalieri** basandosi sul **backlog** [⇨ *lista di obiettivi da raggiungere*]. Si parla di **sprint** [⇨ *iterazione da 2-4 settimane*] e **velocity** [⇨ *quantità di lavoro fattibile in ogni iterazione*]. Esistono vari ruoli:

[**Scrum Master** **Team di Sviluppo (max 7)** **Product Owner**]

➡ Ideazione

☞ La fase di Ideazione

La fase di **ideazione** è la prima che viene svolta in UP, è composta **una sola iterazione**

Lo scopo è formare una **visione comune** del progetto, che comprende tempi e costi

Si individuano pochi **requisiti funzionali** [\Rightarrow *casi d'uso*], i più importanti, ed i **requisiti non funzionali** più critici, e si prepara l'**ambiente di sviluppo** pianificando la prima iterazione della fase di elaborazione

Non vengono definiti precisamente tutti i **requisiti**, ma è importante individuare gli **attori**

☞ Requisiti

Un **requisito** è una **capacità o condizione** a cui il progetto deve essere conforme

Esistono due tipi di requisiti:

[**Funzionali** [\Rightarrow *casi d'uso*] **Non Funzionali** [\Rightarrow *specifiche suppl.*]]

I requisiti non funzionali, le **proprietà** del sistema, spesso sono più **vincolanti** di quelli funzionali, che definiscono **funzionalità e servizi**

I requisiti non funzionali si dividono a loro volta in due categorie:

[**Obiettivi** (*non quantificabili*) **Requisiti n.f. Verificabili** (*quantificabili*)]

Gli **obiettivi** indicano le **intenzioni** degli utenti, e da essi possono nascere requisiti non funzionali più precisi

☞ Casi d'Uso

I **casi d'uso** sono storie scritte che descrivono interazioni tra uno o più **attori** e il sistema

Servono per individuare e registrare i **requisiti funzionali** del progetto, definiscono dei **contratti** relativi al comportamento del sistema

Unified Process incoraggia alla modellazione dei casi d'uso, la pianificazione iterativa è guidata dai requisiti, si elabora il **modello dei casi d'uso**, che comprende il **testo** dei casi d'uso, un eventuale diagramma UML, i **diagrammi di Sequenza di Sistema** [➡ SSD] e i **contratti**

I casi d'uso sono di **facile comprensione** per i clienti e mettono in risalto gli **obiettivi utente**

Un **attore** è un'entità dotata di un comportamento, uno **scenario** è una sequenza di azioni e interazioni (può essere di successo o di *fallimento*), e un **caso d'uso** è un insieme di scenari
Esistono diversi tipi di attori:

[**Primario** **Finale** **Di Supporto** **Fuori Scena**]

Spesso **attore primario** e **finale** coincidono

Esistono tre livelli di dettaglio per i casi d'uso:

[**Formato Breve** **Formato Informale** (più scenari) **Formato Dettagliato** (3-10 pag.)]

I casi d'uso hanno anche diverse scale di grandezza:

[**Obiettivo Utente** **Sotto-Funzione** **Sommario**]

L'approccio corretto è a **scatola nera**, si descrivono le **responsabilità** del sistema ma non il metodo con cui le svolge e le **intenzioni** dell'utente, si scrive in modo **essenziale** ma completo e **non fainfendibile**

Esistono tre test di **utilità** di un caso d'uso:

[**Test del Capo** **Test Elementary Business Process** **Test della Dimensione**]

➡ **Diagramma dei Casi d'Uso**

Il **diagramma dei casi d'uso** è un diagramma grafico di contesto che rappresenta il quadro generale delle funzionalità del sistema, spesso include solo i casi d'uso a livello **obiettivo utente**

Le **relazioni** nel diagramma dei casi d'uso sono generalmente orientate partendo da **chi ha l'intenzione** verso l'**oggetto** a cui l'intenzione è rivolta, possono essere di tre tipi:

[**Include** **Extend** **Generalizzazione**]

➡ Elaborazione

👉 La fase di Elaborazione

La fase di **elaborazione** è la seconda individuata da **UP**, è composta da 2 o più iterazioni

Si produce un nucleo dell'**architettura** e viene testato realisticamente

La **prima iterazione** è incentrata sull'architettura ed affronta gli aspetti più difficili e **rischiosi** del progetto, si realizzano e implementano i casi d'uso individuati nella fase di **ideazione**, e si pianificano le iterazioni successive in base al livello di **rischio**, alla **copertura** e alla **criticità** dei requisiti da realizzare

👉 Analisi a Oggetti [OO]

L'**analisi a oggetti** riguarda la comprensione del problema e l'identificazione dei **concetti** nel suo **dominio**, vengono modellati tre aspetti:

[**Informazioni** [➡ *Modello di Dominio*] **Funzioni** [➡ *SSD*] **Comportamento** [➡ *Contratti*]]

👉 Modello di Dominio

Il **modello di dominio** è l'elaborato principale dell'**analisi**, è una rappresentazione visuale delle **classi concettuali**

La realizzazione ed espansione del modello di dominio è limitata ai **requisiti** dell'**iterazione** corrente

Gli elementi che costituiscono questo modello sono:

[**Classi Concettuali** **Attributi** **Associazioni**]

Il **modello di dominio** è utile in analisi per comprendere il dominio del problema e definire un **linguaggio comune** sul sistema, ed è la fonte d'ispirazione per la **progettazione**

Una **classe concettuale** è un concetto preso dal **mondo reale**, si compone di:

[**Simbolo** (*nome*) **Intensione** (*attributi*) **Estensione** (*istanze o oggetti*)]

Una classe **descrive** un insieme di **oggetti** che sono sue **istanze**

Utilizzare gli stessi **nomi** delle classi concettuali per le **classi software** facilità la comprensibilità

Un **attributo** è una **proprietà** o caratteristica della classe, a cui ogni oggetto assegna un **valore**

Esiste il concetto di **classe descrizione** [\Leftrightarrow Datatype], ovvero una classe che contiene **informazioni** su qualcos'altro, è utile per ridurre la ridondanza di informazioni

Le classi possono essere in **relazione semantica** tra di loro (\Leftrightarrow associazioni), le istanze delle associazioni sono dette **collegamenti**, non corrispondono alle relazioni tra classi software

Le **associazioni** sono rappresentate graficamente con una linea che collega le classi, hanno un'**etichetta** e vi sono indicate le **cardinalità**

L'**aggregazione** è una particolare associazione formata da un insieme di parti, esiste **indipendentemente** da esse, e le sue parti possono essere **condivise** tra più aggregazioni
Si dice **incompleto** se mancano alcune sue parti

La **composizione** è un concetto più rigido rispetto all'**aggregazione**, ogni parte appartiene **solamente** a un composto, il quale ha delle **responsabilità** nei confronti delle sue parti, e può anche rilasciarle

Il **modello di dominio** viene corretto e modificato col passare delle iterazioni, e deve risultare **utile** piuttosto che **corretto**

Requisiti

Nella fase di **elaborazione** vengono rifiniti i casi d'uso, e si inizia a modellare gli **SSD** e i **contratti**

👉 Diagramma di Sequenza di Sistema [SSD]

Un **Diagramma di Sequenza di Sistema** illustra eventi di input e output relativi al **sistema**, e in che modo gli **attori** interagiscono con esso

Il **testo dei casi d'uso** è il documento di input per gli **SSD**

Un **sistema** deve reagire a tre tipi di eventi:

[Eventi Esterni	Eventi Temporali	Guasti o Eccezioni]
---	----------------	------------------	--------------------	---

Gli eventi esterni, o **eventi di sistema**, sono generati dagli **utenti**, e di conseguenza il sistema esegue un'**operazione di sistema**, ovvero un concetto astratto che un soggetto può essere chiamato ad eseguire, indicano **funzionalità pubbliche** e insieme formano

l'interfaccia pubblica del sistema

L'**implementazione** delle operazioni di sistema sono i **metodi**

In UML si usano i **Diagrammi di Sequenza** per rappresentare gli **SSD**, di norma si modellano gli SSD degli **scenari principali di successo** di ciascun caso d'uso

Unified Process non richiede l'utilizzo degli **SSD** ma ne riconosce l'utilità

👉 Contratti

I **contratti** sono elaborati che indicano nel dettaglio i **cambiamenti** agli oggetti di **dominio** in relazione alle **operazioni di sistema**

La maggior parte dei **contratti** viene iniziata e conclusa in fase di **elaborazione**

Non sono obbligatori in **UP**, bensì **complementari** ai casi d'uso

I **contratti** sono composti da:

[Nome	Caso d'Uso	Pre-Condizioni	Post-Condizioni]
---	------	------------	----------------	-----------------	---

Le **post-condizioni** sono osservazioni sui **cambiamenti di stato** degli oggetti **dopo** l'operazione di sistema in considerazione, sono classificate in base al concetto su cui agiscono:

Istanze Attributi Relazioni

Sono espresse in **linguaggio naturale**

 **Dai Requisiti alla Progettazione**

La transizione da **analisi** a **progettazione** avviene gradualmente ogni iterazione: a seguito delle prime concentrate sull'analisi, si susseguiranno iterazioni sempre più interessate all'**implementazione** della **soluzione** dei problemi individuati in analisi

 Architettura Software

L'**architettura software** è un insieme di decisioni significative sull'organizzazione del sistema, raccoglie tutte le decisioni più importanti per sviluppare il progetto, su più **punti di vista**

Spesso si usa una struttura a **più viste** per redigere il documento dell'**architettura software**. Tra queste viste, troviamo l'**architettura dei casi d'uso** e l'**architettura logica**.

👉 Architettura Logica

L'**architettura logica** si occupa di dividere le classi software in **strati**, **package** o **sottoinsiemi** in base alla loro **semantica** a livello software.

Si può dividere secondo più criteri:

Livello (tier)	Strato (layer)	Partizione (partition)
-----------------------	-----------------------	-------------------------------

In UML si può rappresentare come parte del **modello di dominio**

Ogni **package** rappresenta un *namespace*

L'architettura logica a strati prevede più strati o gruppi di classi o package che hanno delle responsabilità coese

Gli strati **inferiori** offrono **servizi generali**, mentre quelli **superiori** servizi **specifici per l'applicazione**

Gli strati superiori possono ricorrere a servizi degli strati inferiori

L'architettura a strati può essere **rilassata** se ogni strato può usufruire dei servizi di qualsiasi strato al di sotto di esso, o **stretta** se solo dello strato subito sotto di esso

Si individuano generalmente tre strati:

[**Interfaccia Utente** **Logica Applicativa** [\rightarrow dominio] **Servizi Tecnici**]

Gli strati permettono di **separare** i vari problemi e **incapsularli**, e inoltre di **riutilizzare** servizi e funzioni diminuendo la ridondanza. Strati diversi possono essere sviluppati da **team** diversi

La **logica applicativa** è lo strato che riguarda gli oggetti di **dominio**, e si divide a sua volta in **strato di dominio** e **strato application** per gestire i flussi da e per il dominio (\rightarrow controller)

☞ Progettazione a Oggetti [OO]

Ci sono diversi approcci alla **progettazione**, progettare mentre si scrive il codice, modellare in UML e solo dopo scrivere il codice, disegnare unicamente il modello UML e generare il codice da esso

In un ottica agile, si **modella** solo per **comprendere** e **comunicare**, e non per documentare, si dedica al disegno UML solo poche ore, massimo un giorno, ad ogni **iterazione**

I modelli per **oggetti** possono essere **statici** [\rightarrow diagramma delle classi, package] o **dinamici** [\rightarrow diagrammi di sequenza, macchine a stati, diagrammi attività]

👉 Diagrammi di Interazione

I **diagrammi di interazione** illustrano come gli **oggetti interagiscono** scambiandosi **messaggi**

Un'**interazione** è una specifica sullo scambio di **messaggi**, allo scopo di eseguire un compito [⇨ indicato nel *messaggio trovato*]

Esistono quattro tipi di diagrammi di interazione:

[**di Sequenza** **di Comunicazione** **di Interazione Generale** **di Temporizzazione**]

👉 Diagrammi di Sequenza [SD]

I **diagrammi di sequenza** [⇨ *SD*] mostrano interazioni tra **linee vitali verticali** [⇨ *lifeline*]

Vengono modellati a livello **software**

Ogni linea di vita rappresenta un **oggetto** o un'**interfaccia**, è possibile integrarli con **vincoli** e frammenti di **codice**

👉 Diagrammi di Comunicazione [CD]

I **diagrammi di comunicazione** [⇨ *CD*] svolgono lo stesso ruolo degli **SD**, ma

rappresentano gli scambi di messaggi con un migliore uso dello **spazio**, sotto forma di **grafo**

Le **lifeline** sono formate nel momento in cui c'è un messaggio diretto a loro

👉 Diagrammi delle Classi Software

Il **diagramma delle classi software** illustra **classi**, **interfacce** e **relazioni** a livello software, è usato per la **modellazione statica del modello di dominio**

Un **oggetto**, istanza di una **classe**, è composto da:

[**Identificativo Univoco** **Valore degli Attributi** **Comportamento**]

Un **classificatore** è un elemento che descrive le caratteristiche **strutturali** di un elemento, nel diagramma delle classi software sono dunque le **classi** e le **interfacce**

In UML esistono concetti come i **vincoli**, regole che estendono la **semantica** di un elemento, **stereotipi**, che definisco un nuovo **elemento di modellazione** e i **tag** che aggiungono **informazioni** specifiche agli elementi. Un **profilo** è un insieme di questi tre concetti

↳ Macchine a Stati [SM]

Le **macchine a stati** [$\Rightarrow SM$] sono dei diagrammi per la modellazione del **comportamento dinamico** dei classificatori, mostrano il **ciclo di vita** di un oggetto tramite una successione di **stati, transizioni ed eventi**

Dato un **evento**, li **oggetti** possono reagire sempre allo stesso modo (**indipendenti dallo stato**) oppure rispondere in modo diverso in base alla loro condizione (**dipendenti dallo stato**)

Ogni **stato** ha un nome e delle azioni di **ingresso** e di **uscita**, può avere delle **transizioni interne** ($\Rightarrow cappi$) e **attività interne**, ovvero attività che vengono eseguite mentre l'oggetto si trova in quello stato

Esistono **stati composti**, che hanno al loro interno una ($\Rightarrow stati composti semplici$) o più ($\Rightarrow macchine ortogonali$) **sotto-macchine**

Si possono avere diversi **pseudostati**, come *giunzione, selezione, di ingresso/uscita...*

Si possono inoltre avere linee di **fork** o **join** per suddividere o ricongiungere diverse linee di flusso

Per **Unified Process** le SM vanno realizzate solo se strettamente necessario

👉 Diagramma delle Attività [AD]

Un **diagramma delle attività** (⇒ AD) si usa per descrivere i **flussi** (⇒ *flowchart*) di dati o oggetti, in **Unified Process** non sono obbligatori e vengono utilizzati esclusivamente per visualizzare il **workflow** di un processo

Un **token** può essere un **oggetto** o un **insieme di dati** che viene passato da **nodo a nodo** attraverso le **transizioni** per rappresentare il flusso

I **nodi azione** rappresentano **unità di lavoro** discrete e atomiche, che possono invocare un'**attività**, un **comportamento** o un'**operazione**

Un **nodo di controllo** controlla il flusso delle **attività** e la gestione dei **token**

I **nodi oggetto** rappresentano **pile** o **code di token** usati nelle attività

Un **pin** è un particolare **nodo oggetto** che rappresenta un **input** o un **output**

Una **partizione** è un raggruppamento logico di di **azioni correlate**

👉 Progettazione Guidata dalla Responsabilità

La **Progettazione Guidata dalla Responsabilità** (⇒ *RDD*) considera un progetto object-oriented come un insieme di oggetti con **responsabilità** che **collabora** per fornire **funzionalità**

Una **responsabilità** è un'**astrazione** di cosa un oggetto **fa** o **rappresenta**, può essere:

[**Responsabilità di Fare** **Responsabilità di Conoscere**]

I **metodi** sono implementati per adempiere alle **responsabilità**

La **RDD** viene eseguita iterativamente, si individuano le singole **responsabilità** una per volta, ci si chiede a quale **oggetto** assegnarla e come tale oggetto la **soddisfa**

☞ Progettare la Visibilità

La **visibilità** è la capacità di un oggetto di avere un **riferimento** ad un altro oggetto per poter scambiare **messaggi** con esso

Esistono quattro modi in cui un oggetto può avere **visibilità** di un altro:

[Attributo Parametro Locale Globale]

La visibilità per **attributo** e **globale** si dicono **permanenti**, in quanto l'istanza dell'oggetto a cui si fa riferimento rimane nel tempo, mentre per **parametro** o **locale** si dicono **temporanee**

☞ Dalla Progettazione all'Implementazione

Gli elaborati creati durante la fase di **progettazione** sono i principali input per l'**implementazione** del **codice** in un linguaggio **object oriented**

Si inizia ad implementare il codice dalla classe **meno accoppiata**

➡ Pattern GRASP

👉 Pattern GRASP

I **pattern GRASP** (➡ General Responsibility Assignment Software Patterns) descrivono dei **principi** di base per la progettazione di **oggetti** e l'assegnazione di **responsabilità**

Un **pattern** è una coppia **problema/soluzione** conosciuta, contiene su come risolvere problemi noti, si basa sull'esperienza

👉 Creator

Creator è un pattern che si occupa del problema di **creare** un oggetto A

La **responsabilità** di **creazione** di A viene affidata a una classe B che, **contiene o aggrega**, **registra**, **utilizza strettamente**, possiede **dati** per **inizializzare** A

👉 Information Expert

Information Expert stabilisce un **principio base** per **assegnare responsabilità** agli oggetti

Una **responsabilità** è assegnata alla classe che possiede **informazioni** necessarie per soddisfarla

Spesso è necessario far **collaborare** più **esperti parziali**, creando però problemi di accoppiamento e coesione

👉 Low Coupling

Low Coupling si occupa di ridurre l'**impatto** dei **cambiamenti**, assegnando le responsabilità facendo sì che l'**accoppiamento** (quanto un elemento è collegato ad un altro) rimanga **basso**

Ha come pattern correlato **Protected Variations**

👉 Controller

Controller stabilisce quali oggetti **ricevono e controllano le operazioni di sistema**

Tale responsabilità è assegnata ad un oggetto che **rappresenta il sistema** (\hookrightarrow punto di accesso al software) o uno **scenario** di un caso d'uso

L'oggetto selezionato deve semplicemente **delegare** il messaggio di operazione dallo strato di UI a quello di **dominio**

👉 High Cohesion

High Cohesion descrive come mantenere gli oggetti **focalizzati, comprensibili e gestibili**

Assegna le responsabilità in modo che la **coesione funzionale** (quanto siano **correlate** le responsabilità) rimanga **alta**

È **molto bassa** se una classe è responsabile in **aree funzionali molto diverse**

È **bassa** se una classe è responsabile di un compito *complesso* in *una sola* area funzionale

È **moderata** se una classe ha responsabilità *leggere* in *poche* aree diverse

È **alta** se una classe ha responsabilità *moderate* in *un'unica* area, e **collabora** con altre classi

👉 Pure Fabrication

Pure Fabrication si occupa di assegnare responsabilità a un oggetto in modo tale da **non violare** altri pattern come High Cohesion e Low Coupling

Viene creata una **classe artificiale** di convenienza alla quale viene assegnato un **insieme di responsabilità altamente coeso**

👉 Polymorphism

Polymorphism è un pattern che si occupa di gestire alternative basate sul **tipo**, sfruttando il **polimorfismo** per gestire variazioni di comportamento in base al tipo dell'oggetto

☞ Indirection

Indirection indica dove assegnare una responsabilità per evitare **accoppiamento** diretto, o in altre parole come **disaccoppiare** oggetti e tenere alto il **riuso**

Si assegnano le responsabilità a oggetti **intermediari** (\Leftrightarrow *Adapter*) tra dominio e servizi esterni

☞ Protected Variations

Protected Variations descrive come **progettare** oggetti in modo tale che **variazioni** di essi non si **riversino** su altri elementi

Si identifica dove sono previste variazioni, e si utilizzano **interfacce stabili** per quegli oggetti in modo da **ridurre l'impatto** e il **costo** dei cambiamenti

➡ Design Pattern GoF

👉 Design Pattern GoF

I **Design Pattern GoF** (↔ *Gang of Four*) sono una **soluzione** progettuale a problemi ricorrenti e a differenza dei **pattern GRASP**, mostrano soluzioni **concrete** con i relativi **diagrammi** delle classi

Sono in totale **23**, e sono classificati in base al loro **scopo**:

[**Creazionale** **Strutturale** **Comportamentale**]

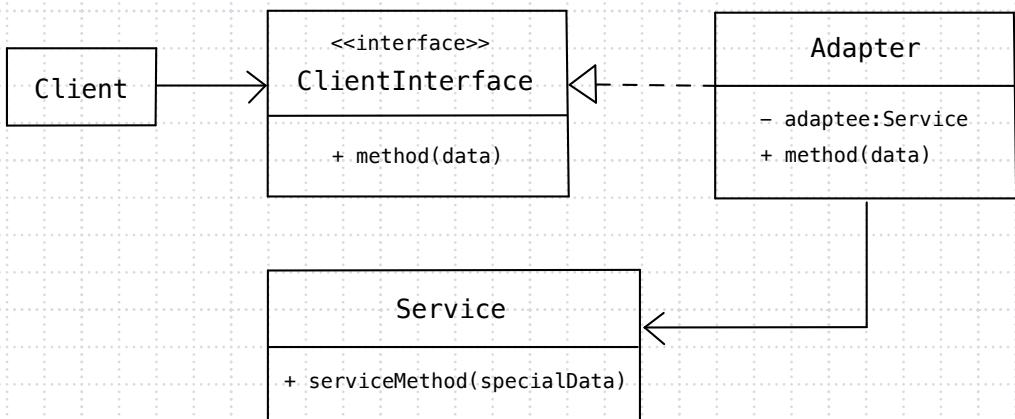
Possono essere anche classificati in base al **raggio d'azione**:

[**Classi** **Oggetti**]

👉 Adapter

Adapter è un pattern **strutturale** di **classe** (e **oggetto**) che si occupa di gestire la **comunicazione** tra interfacce **incompatibili** tra di loro

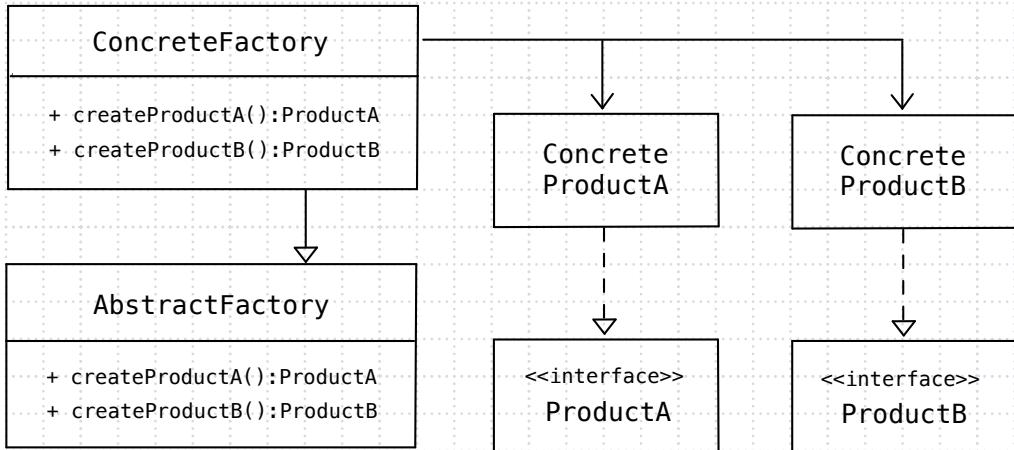
Si crea una classe (o oggetto) **adattatore intermedio** che *implementa* una delle due interfacce effettuando le operazioni di **conversione** necessarie per farla comunicare con la seconda **interfaccia**



☞ Factory

Factory o **Abstract Factory** è un pattern **creazionale** di **oggetti** che si occupa di definire una classe responsabile della **creazione** di oggetti in situazioni in cui si vuole **separare** tale responsabilità dalle classi che **utilizzano** gli oggetti, per una coesione migliore

Si crea una classe **Pure Fabrication** chiamata **factory** che gestisce interamente la creazione degli oggetti

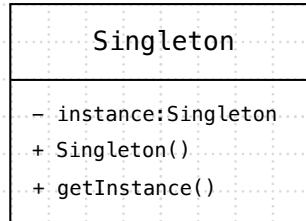


☞ Singleton

Singleton è un pattern **creazionale** di **oggetti** che definisce una classe di cui è **consentita** una sola **istanza** alla quale gli altri oggetti fanno riferimento **globalmente**

Dentro la classe **singleton** si definisce un **metodo statico** che restituisce l'istanza stessa della classe, questo metodo (\mapsto `getInstance()`) è un metodo **synchronized** per evitare di creare due istanze diverse in caso sia presenti diversi **thread**

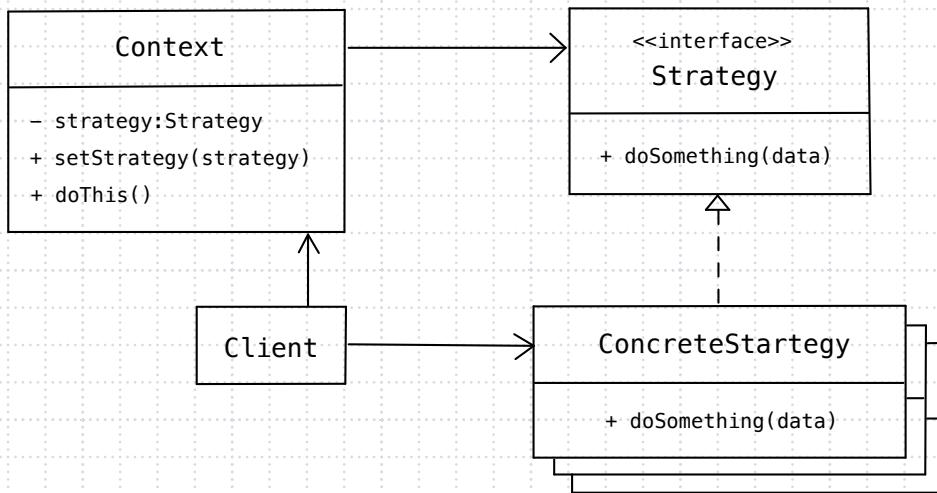
```
public static class Singleton {
    private static Singleton instance = null;
    public static synchronized Singleton getInstance() {
        if(instance == null)
            instance = new Singleton();
        return instance;
    }
    /* ... */
}
```



👉 Strategy

Strategy è un pattern **comportamentale** di **oggetto** che si occupa di trovare una soluzione all'implementazione di un **insieme di algoritmi** o politiche **correlate** tra di loro ma che possono **cambiare nel tempo**

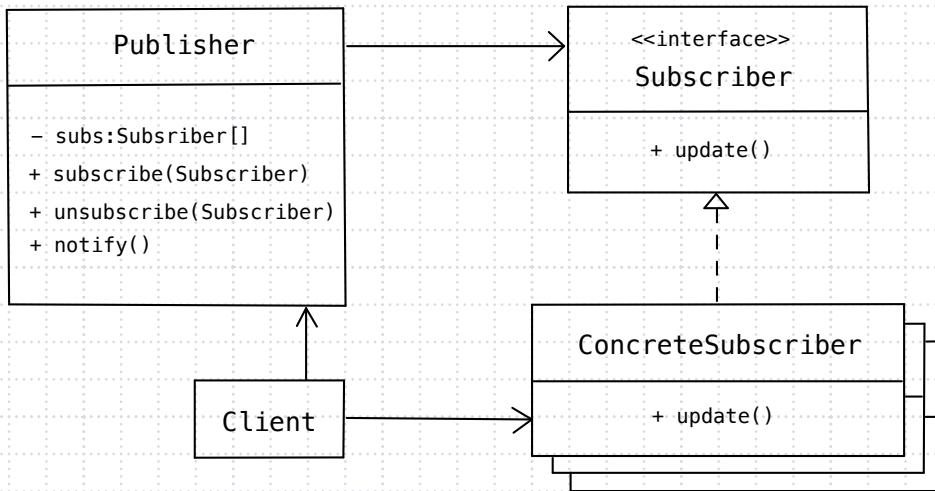
Ciascun algoritmo o politica è definito in una **classe separata** che implementa un'**interfaccia comune**, la classe di **contesto** che ha necessità di *eseguire* uno degli algoritmi *parla* direttamente con l'*interfaccia comune* senza sapere di preciso **quale** algoritmo sta utilizzando, è una classe **client** esterna a istanziare il tipo concreto di algoritmo e passarlo al **contesto**



👉 Observer

Observer è un pattern **comportamentale** di **oggetto** che indica come gestire diversi tipi di oggetti detti **subscriber** che sono interessati a **eventi** o **cambiamenti di stato** di un altro oggetto detto **publisher**, ma vogliono ognuno reagire in modo differente

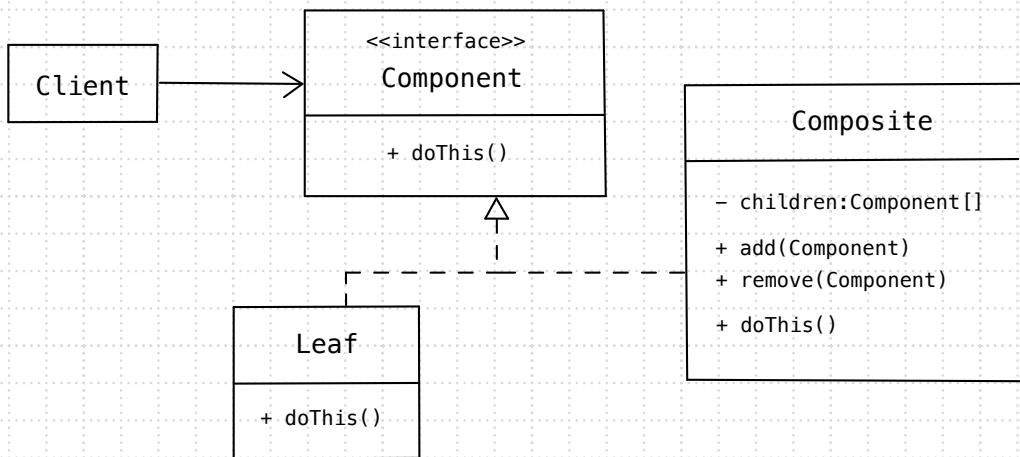
Si crea un **interfaccia comune** detta **listener** o **subscriber** dalla quale si implementano tutti gli oggetti **subscriber**, che vengono registrati **dinamicamente** nel **publisher** dal **client**, che è l'unico in relazione diretta con i **subscriber concreti**



☞ Composite

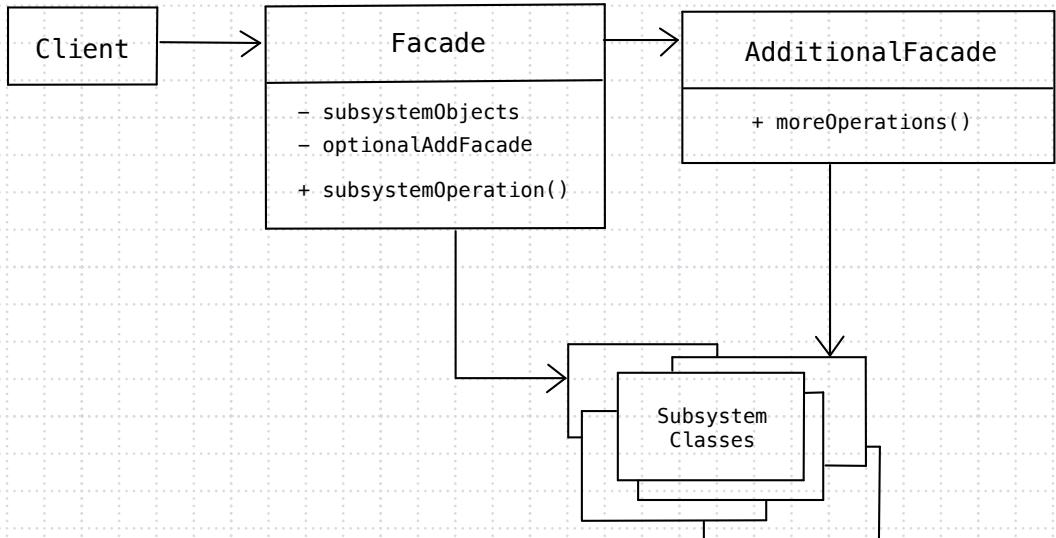
Composite è un pattern **strutturale** di **oggetti** che si occupa di trattare insiemi di oggetti **polimorfi** di cui alcuni **semplici** (atomici) ed altri **composti** da più oggetti dello stesso tipo. Si definisce un'**interfaccia comune** agli oggetti atomici e composti, che verranno definiti in **classi separate**.

L'oggetto composto (⇨ *container* o *composite*) contiene un **riferimento** alla stessa interfaccia comune, che può essere **istanziato** a sua volta in un oggetto atomico (⇨ *leaf*) o composto, creando così una struttura ad **albero**.



☞ Facade

Facade è un pattern **strutturale** di **oggetti** che si occupa di gestire un **insieme disparato** di implementazioni o interfacce creando una singola classe di **facciata** a fare da **punto di contatto** tra il **client** e il **sottosistema**, responsabile della **collaborazione** dell'insieme



➡ Testing

☞ Sviluppo Guidato dai Test

Lo **sviluppo guidato dai test** (⇒ TDD) è una *best practice* applicabile a **Unified Process**,

consiste nello **scrivere i test prima del codice**

In questo modo il programmatore ha un'**idea più chiara** di quello che il programma deve fare ed è più **soddisfacente** affrontare la programmazione come una **sfida**

Esistono vari tipi di test, dalla verifica di singole parti (**unità**), al funzionamento complessivo del sistema dal punto di vista dell'utente:

[**di Unità** **di Integrazione** **di Sistema** **di Accettazione**]

☞ Test di Unità

Un **test di unità** consiste nei seguenti passi:

1. **Preparare** l'oggetto da verificare (⇒ **fixture**)
2. Far **eseguire** delle operazioni alla fixture
3. **Verificare** che i risultati corrispondano a quelli previsti
4. Opzionalmente **rilasciare** e ripulire oggetti e risorse

Un framework diffuso per i test unitari è xUnit

☞ Test Unitari Ciclici

Il **TDD** utilizza cicli molto brevi in cui si scrive un test unitario che **fallisce**, si scrive il **codice più semplice** per far **passare** il test e infine si **ristruttura** (⇒ *refactor*) e pulisce il codice per migliorarlo (o si passa direttamente al test successivo, ristrutturando dopo più test)

Questo tipo di ciclo viene detto **ciclo base**, esiste un secondo tipo detto **ciclo doppio**

Un ciclo **doppio** è composto da un ciclo più esterno che effettua test di **accettazione**, con al suo interno un ciclo base *innestato*; per ogni test di **accettazione** si scrivono più test **unitari**

👉 Refactoring

Il **refactoring** è un metodo strutturato per **ristrutturare** del codice esistente senza modificarne il comportamento tramite **piccoli passi** di trasformazione e **test** dopo ogni passo
Oltre a **migliorare il codice**, consente di preparalo a cambiamenti

Alcuni degli obiettivi principali sono eliminare codice **duplicato** (⇨ **regola del tre**, al più due ripetizioni), migliorare la **chiarezza** e **abbreviare** i metodi lunghi

Si effettua **refactoring** ogni volta che si aggiunge una **funzionalità** o si corregge un **bug**

👉 Code Smell

Un **code smell** è una caratteristica del codice che indica la presenza di **cattive pratiche** di programmazione, sono divisi in **categorie**, di cui le seguenti 10 nell'ambito della programmazione OO

👉 Bloaters

La categoria dei **bloaters** è composta da

[Long Method Large Class Long Parameter List]

Long Method → **Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, Replace Method with Method Object, Decompose Conditional**

Large Class → **Extract Class, Extract Subclass, Move Method, Replace Parameter with Method Call, Preserve Whole Object, Introduce Parameter Object**

Long Parameter List → **Replace Parameter with Method Call, Preserve Whole Object, Introduce Parameter Object**

👉 OO Abusers

La categoria dei **OO Abusers** è composta da

[**Switch Statement** **Refused Bequest**]

Switch Statement → **Replace Conditional with Polymorphism, Extract Method, Move Method**

Refused Bequest → **Replace Inheritance with Delegation, Extract Superclass**

👉 Change Preventers

La categoria dei **Change Preventers** è composta da

[**Shotgun Surgery**]

Shotgun Surgery → **Move Method, Move Field, Inline Class**

👉 Dispensables

La categoria dei **Dispensables** è composta da

[**Duplicated Code** **Data Class** **Comments**]

Duplicated Code → **Extract Method, Move Method, Extract Class**

Data Class → **Extract Method, Extract Class**

Comments → **Extract Variable, Extract Method, Rename Method, Introduce Assertion**

☞ Couplers

La categoria dei **Couplers** è composta da

[**Feature Envy**]

Feature Envy → **Extract Method, Move Method**