1 SOLID

- *Single Responsibility Principle* (SRP): una classe deve avere una sola responsabilità e una sola ragione per cambiare.
- *Open-closed principle* (OCP): le entità software devono essere aperte all'estensione, ma chiuse alla modifica. OCP può essere violato solo per fare refactoring.
- *Liskov Substitution Principle* (LSP): si deve poter usare un tipo specifico (sottoclasse) al posto di quello generico (superclasse) senza alterare il comportamento atteso del sistema.
- *Interface Segregation Principle* (ISP): le interfacce devono essere specifiche e non generiche, evitando di forzare le classi a implementare metodi che non usano.
- *Dependency Inversion Principle* (DIP): le classi di alto livello non devono dipendere da quelle di basso livello, ma entrambe devono dipendere da astrazioni. Le astrazioni non devono dipendere dai dettagli, ma i dettagli dalle astrazioni.

2 General Responsibility Assignment Software Patterns (GRASP)

Si attribuiscono responsabilità agli elementi e gli elementi dialogano in base alle responsabilità. Le responsabilità sono assegnate durante la fase di design degli oggetti. GRASP può essere usato per fare RDD (Responsibility-Driven Design) garantendo i principi SOLID o per imparare OO design con le responsabilità.

I pattern GRASP sono problemi ricorrenti ai quali si è trovata una soluzione riutilizzabile, il cui funzionamento è stato dimostrato. I pattern sono:

2.1 Creator

Problema: Chi è *responsabile* dell'istanziazione di un oggetto A?
Soluzione: Si assegna a B la responsabilità di creare A se una delle seguenti condizioni è vera:

- B aggrega A;
- B contiene A;
- B usa spesso metodi di A;
- B registra A;
- B ha tutti i dati per creare A

2.2 Information Expert

Problema: Come si assegnano le *responsabilità* a oggetti affinché i sistemi siano più facili da capire, mantenere ed estendere?
Soluzione: Si assegna la responsabilità all'Information Expert, ovvero la classe che ha l'informazione necessaria per soddisfare la responsabilità

2.3 Controller

Problema: Quale oggetto è *responsabile* di gestire una system operation[1]?
Soluzione: Si assegna la responsabilità a una classe che rappresenta:

- tutto il sistema;
- un oggetto radice;
- un dispositivo su cui sta venendo eseguito il software;
- un sottosistema grande

2.4 Low Coupling

Problema: Come si progetta un sistema con:

- poche dipendenze;

[1] *System Operation*: invocazione di un metodo che corrisponde all'attivazione di qualcosa nella UI.

- facile da modificare
- favorisce il riuso del codice

Soluzione: Si assegna la responsabilità affinché l'accoppiamento tra le parti rimanga basso: non si creano dipendenze tra classi che non sono necessarie per soddisfare le responsabilità.

2.5 High Cohesion

Problema: Come si mantengono gli oggetti concentrati, comprensibili, gestibili e supportati per il Low Coupling?
Soluzione: Si assegna la *responsabilità* affinché la coesione rimanga alta

2.6 Pure Fabrication

Problema: Cosa si fa quando nessuna classe ha le informazioni necessarie per soddisfare una *responsabilità*[2]?
Soluzione: Si crea una classe artificiale (non ispirata al dominio) per mantenere la coesione, ridurre l'accoppiamento e supportare altri principi come Single Responsibility.

2.7 Indirection

Problema: Dove si assegna la *responsabilità* per evitare Direct Coupling tra due o più oggetti?
Soluzione: Si assegna la responsabilità a un oggetto intermediario che fa da mediatore tra gli altri due, riducendo il Direct Coupling e migliorando la manutenibilità.

2.8 Polimorfismo

Problema: La variazione condizionale causata da statement del *control flow* produce codice difficile da leggere[3].
Soluzione: Si usano alternative basate sul tipo. Il polimorfismo permette di attivare comportamenti diversi in base all'oggetto usato senza andare ad utilizzare controlli di flusso

2.9 Protected Variations (PV)

Problema: Come si limitano le portate dei cambiamenti? Come si evitano cambiamenti a catena?
Soluzione: Si assegnano le responsabilità per creare interfacce e classi astratte stabili attorno a punti dove si prevedono variazioni.

Una corretta implementazione di PV aiuta a rispettare LSP.

3 AGILE software development

AGILE è una collezione di principi e pratiche per lo sviluppo software che enfatizza la collaborazione, la flessibilità e la consegna continua di valore.

3.1 Pratiche AGILE

- **Code Review**: prima di essere rilasciato sul mercato, il codice deve passare dei test. La revisione viene fatta da tutti i membri del team;
- **Pair Programming**: il codice viene scritto a coppie: ci si alterna tra pilota (scrive) e navigatore (fa code review);
- **Test Driven Design**: Si scrive il codice in base a i test che deve passare[4];

[2] Ovvero quando non c'è un information expert.
[3] Il comportamento del codice varia in base al tipo di un oggetto e si usa un `if` o uno `switch` per controllare il tipo. Ad esempio `if(myDog instanceof Dog){...}`
[4] Al posto di scrivere il codice e poi eseguire test su di esso.

3.2 User Stories

Le user stories sono descrizioni funzionali dal punto di vista dell'utente finale. Sono uno strumento AGILE per comunicare i requisiti in modo semplice. Sono implementate con dei template:

- `as a <role>, I want <goal> so that <benefit>`
- *given-when-then*

Le storie non devono sopravvivere al loro processing, ma i loro acceptance test si.

- **Storia**: descrive le feature di alto livello, non è molto specifica e viene raffinata nel corso del progetto;
- **Epica**: storia grande sviluppata in più di un'interazione;
- **Tema**: collezione di storie correlate;

3.3 INVEST

Criteri per valutare la qualità di una storia

- **Independent**: le storie non devono dipendere l'una dall'altra;
- **Negotiable**: le storie sono il risultato di una negoziazione e possono essere ri-negoziate;
- **Valuable**: le storie devono fornire valore;
- **Estimable**: il team deve essere in grado di stimare il livello di complessità e la quantità di lavoro richiesta per l'analisi della storia;
- **Small**: una storia deve essere realizzata in un'iterazione;
- **Testable**: una storia è finita solo quando le feature corrispondenti passano i test di accettazione;

3.4 Extreme Programming

Metodo di sviluppo software basato su caratteristiche AGILE. Si basa su 4 attività: coding, testing, listening e designing. E 5 valori: comunicazione, semplicità, feedback, coraggio e rispetto.

Pratiche dell'extreme programming sono:

- **Test Driven Development**: si inizia a scrivere il codice scrivendo i test;
- **Whole Team**: tutte le figure necessarie per lo sviluppo lavorano insieme in modo collaborativo e continuo;
- **Continuous Process**:
  - interazione continua;
  - miglioramento del design;
  - aggiornamenti piccoli.

  L'indicatore dello stato del progetto è la funzionalità del software;
- **Planning Game**: processo di pianficazione basato sulle storie. Si fa prima di ogni iterazione ed è composto da
  - pianificazione delle release con i clienti;
  - pianificazione dell'iterazione solo tra sviluppatori

  I clienti ordinano le storie in base alla loro importanza, gli sviluppatori in base al rischio. Si scelgono le storie da completare entro la prossima release;

4 Testing

Il testing è l'attività principale tra quelle di *validazione e verifica*, usate per controllare che il software testato sia conforme alle specifiche. Con il testing si rileva la presenza di un qualche tipo di errore logico.

Livelli di testing:

- Unit → classe/metodo: poco costosi sia da scrivere che da eseguire
- Integrazione → gruppo di moduli
- End to end → intero sistema: si manda in esecuzione l'intera applicazione. Non sempre sono automatizzabili

Analisi statica: si controlla il codice per trovare bug, senza eseguire il codice. Si basa su metodi formali come *model checking*, *data-flow analysis*, *abstract interpolation* e *symbolic execution*. Si usano pattern di bug per valutare la qualità del codice.

Analisi dinamica: il codice viene eseguito: il test viene progettato con un approccio
- *whitebox*: si usa la struttura del codice per definire i test
- *blackbox*: ci interessa solo il risultato senza guardare il codice che c'è dietro[5].

	Contro	**Pro**
white	Complesso	Copertura maggiore Si acquisisce conoscenza sul codice creando i test
black	Copertura sconosciuta	I tester non devono essere sviluppatori, si avvicina di più ai requisiti

I test vengono validati creando mutazioni del codice che poi viene testato. Se questo passa, vuol dire che c'è un problema.

Unit Testing: si testa una singola funzione, il *subject* è molto piccolo e non può essere ulteriormente scomposto. I SUT[6] devono essere isolati. Il test set di ogni unit deve avere casi indipendenti.

Isolation: si creano degli oggetti finiti finti che sostituiscono le dipendenze reali. Si caricano gli oggetti finiti con il minimo dispensabile per poter far funzionare i test. Una classe testabile deve essere associata ad un'interfaccia.

5 Design Pattern

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

[5] Si testano punti di discontinuità, dei valori casuali attorno ad essi e tutte le combinazioni possibili dei parametri.
[6] SUT: System Under Testing

5.1 Pattern strutturali

5.1.1 Privilegiare la composizione rispetto all'ereditarietà

Quando due o più classi condividono del comportamento comune, si hanno due opzioni:
- Ereditarietà: creare una superclasse con il metodo comune da ereditare.
- Composizione: creare una classe separata che contiene il comportamento comune e farla usare (delegare) alle classi interessate.

Si privilegia la composizione:
- **Problemi dell'ereditarietà**:
  - È troppo generosa: oltre alla sostituibilità (Liskov), porta con sé anche il codice (i metodi) della superclasse.
  - Se si eredita, non c'e scelta: si prende tutto, anche parti che non servono.
  - Il linguaggio (come Java) verifica automaticamente la compatibilità dei tipi, ma presume che se A estende B, allora A può essere usata ovunque sia previsto B (principio di sostituibilità). Questo non è sempre vero nel comportamento pratico.
  - Non rispetta OCP e PV.
- **Vantaggi della composizione**:
  - Le classi non sono legate da una gerarchia rigida.
  - Il comportamento comune viene delegato a una classe esterna.
  - Le classi contengono un riferimento a un oggetto (aggregation), e delegano ad esso.
  - Si ha più controllo e minore accoppiamento.

Facade: Consente di isolare un client dalla complessità interna di un sottosistema. Si definisce un'interfaccia di alto livello che rende un sottosistema più facile da usare.

Proxy: Permette di intercettare e controllare l'accesso ad un oggetto per indirizzare problemi ortogonali[7]. Il proxy fornisce il sostituto o un segnaposto per un altro oggetto per controllare l'accesso ad esso.

Decorator: Si usa per aggiungere responsabilità ad un oggetto in modo dinamico. I decorator forniscono una flessibilità maggiore rispetto all'ereditarietà perché possono essere combinati.

Adapter: Permette a classi con interfacce incompatibili (parametri o tipi diversi) di collaborare tra loro, a differenza di Proxy dove l'intermediario ha la stessa interfaccia.

Bridge: Si usa per separare un'astrazione dalla sua implementazione affinché le due possano variare indipendentemente. Ha la struttura uguale all'Adapter, ma quello che cambia è l'intento. Con questo pattern si rompe la gerarchia nel quale è il client a decidere. Adapter serve a far funzionare le cose dopo che sono state disegnate, mentre Bridge viene pensato prima ancora della creazione del modulo di basso livello.

Composite: Si compongono gerarchie parte-tutto in strutture ad albero. Composite permette di trattare oggetti singoli e composizioni di oggetti in modo uniforme. Gli elementi dell'albero hanno due ruoli distinti: ruolo intermedio che rimanda ad altri elementi e ruolo terminale.

Flyweight: Si riduce l'uso di memoria condividendo quanti più dati possibili tra oggetti simili. Ha metodi per accedere allo stato condiviso. Il client non sa come è fatto l'oggetto, sa solo che può essere condiviso.

[7] I problemi ortogonali sono problemi significativi che non fanno parte del dominio del problema.

5.2 Pattern creazionali

5.2.1 `new` è pericoloso

Ogni volta che si crea una classe concreta con new `new` si crea una dipendenza di cattiva qualità che viola DIP: se cambia il costruttore della classe concreta bisogna cambiare ogni occorrenza di `new` di quella classe.

Abstract Factory: Al posto di usare `new`, si delega la creazione dell'oggetto a una classe a parte, la Factory. Le factory separano il client[8] dal processo di istanziazione e delegano la creazione dell'oggetto a un'interfaccia comune. Questa è una dipendenza di buona qualità perché vi si possono solo aggiungere metodi. Nelle *Abstract Factory*, il client non sa come è fatto l'oggetto, ma sa che può essere creato.

Singleton: Garantisce che una classe abbia una sola istanza, fornendo un punto di accesso globale ad essa. Sono un tipo di *code smell*. In genere sono Singleton Factory, Logger, Classi di configurazione e accesso alle risorse. Si usa il Singleton solo quando l'unicità è parte del dominio stesso, non solo dell'implementazione.

Builder: Si usa per costruire oggetti complessi passo per passo, separando la costruzione dalla rappresentazione finale. Un Director delega la costruzione di parti della struttura a diversi Builder (possono essere interfacce), e restituisce l'oggetto aggregato.

Prototype: Si usano istanze di oggetti esistenti come prototipi per creare nuovi tipi. Si copiano oggetti esistenti senza rendere il codice dipendente dalle loro classi.

5.3 Pattern comportamentali

Template Method: Si definisce lo scheletro di un algoritmo in un metodo[9], delegando alcuni passi alle sottoclassi. I comportamenti più comuni saranno in cima all'albero di ereditarietà. In questo modo le dipendenze sono dirette verso elementi più stabili e si favorisce l'aderenza a OCP.

Strategy: Consente di separare un oggetto da parte del suo comportamento e cambiarlo a runtime. Si definisce una serie di algoritmi incapsulati tra loro intercambiabili. Strategy favorisce l'implementazione di OCP e obbedisce PV. Inoltre favorisce la composizione rispetto all'ereditarietà.

State: State è come strategy solo l'oggetto cambia il suo comportamento in base al suo stato (interno).

Observer: Permette di propagare le modifiche di una classe su una serie di oggetti. Si fa si che gli oggetti interessati ricevono la notifica del cambiamento cambiato, non viceversa. Si crea una dipendenza uno a molti, cosi quando uno cambia stato, tutti i dipendenti sono notificati. Favorisce il disaccoppiamento.

Memento: Si usa per catturare ed esternalizzare lo stato interno di un oggetto affinché possa essere ripristinato in un momento successivo senza violare l'incapsulamento.

1. Un *caretaker* chiede ad un *originator* di salvare il suo stato.
2. Il *caretaker* ripristina lo stato quando necessario
3. Il *caretaker* non sa come è fatto lo stato ma sa che può essere ripristinato
4. Il *Memento* è l'oggetto che contiene lo stato dell'originator. Non può essere modificato dall'esterno

[8] Con client si intende qualsiasi componente (classe, modulo, funzione, sistema…) che usa un altro componente per ottenere un servizio o una funzionalità.

[9] Questo metodo è il *template method*, in genere si trova in una classe astratta.

Iterator: Fornisce un modo per accedere agli elementi di un oggetto aggregato senza esporre la sua rappresentazione interna. Il client non sa come è fatto l'oggetto aggregato, ma sa che può essere iterato.

Mediator: Si definisce un oggetto che incapsula come un insieme di oggetti interagiscono. Il mediatore promuove il *loose coupling* evitando che gli oggetti si riferiscano l'uno all'altro esplicitamente, e permette di variare le interazioni tra gli oggetti.

Visitor: Simile all'Iterator, ma usa IoC. Al posto di usare `Iterator.next()` si richiama un metodo per ogni elemento, si prende ognuno degli elementi che compongono il Visitor e si invoca un metodo su di esso. Il Visitor ha metodi per ogni tipo di elemento, e il client non sa come è fatto l'element, ma sa che può essere visitato.

Command: Si incapsula una richiesta come un oggetto, permettendo di parametrizzare i client con code e operazioni. Assomiglia a Visitor. Command permette di ritardare, mettere in coda le richieste.

Chain of Responsibility: Si passa una richiesta lungo una catena di *handler*. La catena di responsabilità permette di invocare più oggetti senza sapere chi gestirà la richiesta. Favorisce il decoupling e la flessibilità.

Interpreter: Si definisce la grammatica di una lingua con una struttura ad albero. Favorisce modularità del codice ed estensione.

5.4 Pattern Moderni