

Ingegneria Software

1	UML	1
2	Use Classe	1
3	UML Class Diagram	2
4	Domain Model	4
5	UML Activity Diagram	4
6	Sequence Diagram	6
7	UML State Machine Diagram	7
8	SOLID	8
9	General Responsibility Assignment Software Patterns (GRASP)	8
10	AGILE Software Development	9
11	Testing	11
12	Design Pattern	11
13	Scrum	16
14	Modello di Analisi	18

1 UML

Elementi:

- Classe: gruppo di oggetti con attributi e metodi comuni.
- Caso d'uso: descrive un'interazione tra un attore e il sistema, rappresenta una funzionalità del sistema.

Relazioni:

- Associazione: dato un oggetto A, si può risalire ad un oggetto B legato concettualmente ad A.

A ————— B

- Generalizzazione: un elemento B è la specializzazione di un elemento A, ovvero B è un tipo di A.

B —————> A

- Dipendenza: un elemento A utilizza le funzionalità di un'altra classe B (relazione "utilizzatore-fornitore").

A - - - - -> B

- Realizzazione: un elemento A realizza, o implementa un'interfaccia B.

B - - - - -> A

2 Use Classe

I casi d'uso sono storie scritte usate per registrare e identificare i requisiti. Con i diagrammi dei casi d'uso si specificano i confini del sistema, il suo comportamento e le interazioni tra gli attori e il sistema stesso.

Elementi:

- Attore: soggetto dotato di comportamento, esterno al Sistema in Discussione.
- Sistema in Discussione: oggetto a cui si applicano i casi d'uso. Gli attori rimangono disegnati esternamente nel caso d'uso di sistema.
- Caso d'uso: collezione di scenari correlati, di successo e fallimento, che descrivono un attore che usa un sistema per raggiungere un obiettivo. I casi d'uso sono requisiti, soprattutto funzionali e

comportamentali, che indicano che cosa farà il sistema. Ogni caso d'uso descrive un'unità di funzioni complete che l'oggetto fornisce ai propri utenti.

- Scenario: sequenza specifica di azioni e interazioni tra il sistema e alcuni attori, descrive una particolare storia nell'uso del sistema o un percorso attraverso il caso d'uso.

Relazioni:

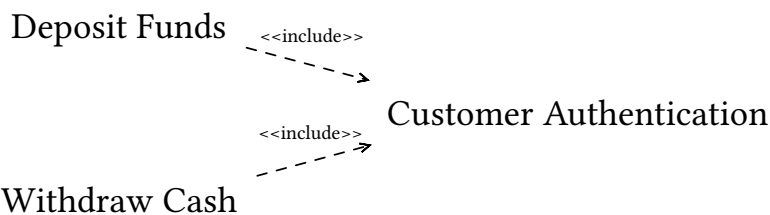
- Associazione: un attore è associato ad un caso d'uso quando interagisce con esso. Gli attori possono essere associati ai casi d'uso solo tramite associazione binaria¹.
- Generalizzazione: un elemento è generalizzazione di uno più generale.



- Estensione: si vuole sottolineare che c'è un comportamento addizionale e opzionale che non sempre ha luogo.



- Inclusione: si tratta di un refactoring e di un collegamento del testo per evitare la duplicazione. La relazione di inclusione serve a specificare dei passi ulteriori che caratterizzano un certo caso d'uso e che vengono eseguiti sempre.



Non esiste una notazione standard per quanto riguarda i dettagli dei casi d'uso, ma ci sono diversi template che possono essere usati. Quello più semplice è:

Use case Name
ID
Actors
Pre-conditions
Main Sequence
Alternative Sequences
Post-conditions

3 UML Class Diagram

Una classe è rappresentata da un rettangolo, in cui in alto è presente il nome della classe. Il rettangolo può essere diviso in compartimenti (righe orizzontali) che possono contenere informazioni addizionali, tra cui metodi e attributi.

Il diagramma delle classi UML nasce per rappresentare le classi come elementi di un programma. Serve a progettare soluzioni e sistemi SW, quindi viene utilizzato nel dominio della soluzione.

Modificatori di visibilità

¹non usato nella modellazione concettuale.

- +: public;
- : private;
- #: protected;
- ~: package, visibile a tutti gli elementi presenti nel package.

Molteplicità

Le molteplicità servono a specificare quante volte operazioni, proprietà sono presenti:

<multiplicity-range> ::= [<lower>..] <upper>. Molteplicità utilizzate frequentemente:

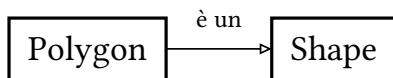
- 1 → proprietà obbligatoria
- 0..1: proprietà che può esserci 0 o 1 una volta, quindi facoltativa
- 1... *: proprietà che deve essere presente almeno 1 volta
- *: proprietà presente 0 o più volte (*).

Istanza

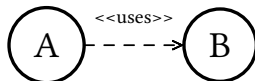
Il diagramma degli oggetti definisce le istanze. Un'istanza è un oggetto che ha un'identità, uno stato e un comportamento. Il diagramma degli oggetti è una rappresentazione statica di un sistema in un momento specifico, utile per visualizzare le relazioni tra gli oggetti e le loro istanze. La notazione grafica dell'istanza si distingue da quella della classe perché il nome è sottolineato ed è composto da una concatenazione `instanceName : Class`.

Relazioni

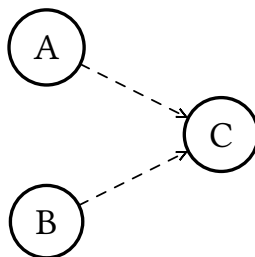
- Generalizzazione/Specializzazione: corrisponde alla frase "è un..." e dal verso opposto si può leggere come specializzazione.



- Dipendenza: relazione di tipo "utilizzatore/fornitore" fra gli elementi.

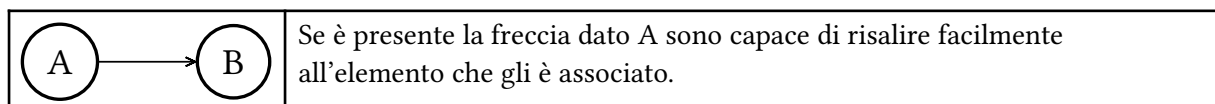


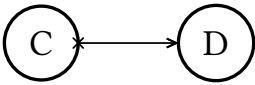
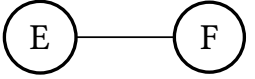
- Realizzazione: dichiara che un elemento ne realizza un altro. Il caso più comune è quando le classi realizzano/implementano un'interfaccia.



A e B implementano C.

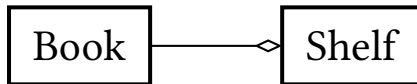
- Associazione: dichiara che ci possono essere collegamenti tra le istanze dei tipi associati: dato un elemento A si è in grado di risalire ad un elemento B di un altro tipo, legato ad A in maniera concettuale.



	Se è presente una croce, allora dato C non è immediato ottenere l'elemento a cui è associato (D).
	Se non è presente né freccia né croce è indeterminata la facilità con cui si può risalire all'elemento associato.

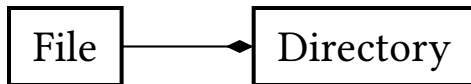
Aggregazione e composizione sono due tipi particolari di associazione.

- Aggregazione: è un rafforzamento del legame dell'associazione, ma l'elemento aggregato esiste anche senza l'altro elemento. L'elemento che viene aggregato generalmente ha una cardinalità aperta (*).



Book esiste anche senza shelf.

- Composizione: è ancora più forte dell'aggregazione, perché l'elemento che è associato dipende/è parte dell'elemento aggregante.



Il file non può esistere senza la directory che lo contiene..

Classe Astratta: il nome di una classe astratta è compreso fra graffe o viene scritto in corsivo: {abstract}

Interfaccia: il nome delle interfacce viene compreso fra ghirne << >>.

4 Domain Model

Illustra i concetti significativi di un dominio, le caratteristiche e come si relazionano tra loro. Un modello di dominio è una rappresentazione visuale di classi concettuali o di oggetti del mondo reale di un dominio. Il modello di dominio è una rappresentazione di classi concettuali del mondo reale, non di oggetti software, è un dizionario visuale delle astrazioni significative della terminologia del dominio e del contenuto informativo del dominio.

Come creare un modello di dominio

1. Trovare le classi concettuali: ci sono 3 strategie:
 1. usare o modificare modelli esterni;
 2. utilizzare un elenco di categorie;
 3. identificare nomi e locuzioni nominali.
2. Disegnarle come class in un diagramma delle classi UML;
3. Aggiungere associazioni e attributi.

5 UML Activity Diagram

diagramma di attività è un diagramma UML del comportamento che mostra le attività sequenziali e parallele in un processo. Un'attività non può cominciare finché un'altra non è terminata. I diagrammi di attività possono essere usati per modellare il comportamento di casi d'uso, interfacce, componenti, operazioni delle classi, algoritmi. Ci sono dei token che scorrono attraverso il grafo dell'attività.

Elementi:

- L'elemento principale è l'attività che si sta descrivendo. Viene rappresentata come un flusso di azioni coordinato. L'attività è strutturata in nodi².
- Activity edges permettono di collegare fra loro i vari nodi e definiscono come i flussi si muovono all'interno dell'attività.
 - Il control flow indica la posizione del token
 - Object flow indica il passaggio di dati tra le azioni.

È possibile specificare delle guardie che definiscono dei predicati che devono essere veri perché il token possa attraversare l'edge.

L'idea dei diagrammi di attività è quella di definire un insieme di azioni e specificare come si evolvono nel corso del tempo. Questa struttura di controllo permette di esprimere le alternative che il flusso di controllo può realizzare all'interno del modello.

- La barra verticale abilita due flussi: cioè entra un flusso e ne escono due paralleli.
- Object node: nodo che definisce un punto del flusso in cui diverse azioni si passano dei dati sotto forma di oggetto. L'object node specifica dei dati che vengono trasportati tra un'azione e l'altra.
- Azione: elemento a cui è associato un nome, che rappresenta un passo atomico all'interno dell'attività.
- Event action: azione che corrisponde a eventi/segnali/messaggi generati.
- Una call behavior action rappresenta la chiamata di un'attività, cioè l'attività si espande in un diagramma di sotto-attività.
- Gli activity parameter nodes sono object nodes situati o all'inizio o alla fine del flusso, in quanto rappresentano che la prima azione per avviarsi necessita di un parametro.
- Le Input pin sono object node che ricevono valori da altre azioni attraverso object flows. Le Output pin sono object node che ritornano valori a altre azioni attraverso object flows.
- I connectors sono una notazione che permette di evitare di disegnare degli edge lunghissimi e semplificare la grafica dell'AD. Sono elementi puramente notazionali, quindi non hanno nessun effetto sul diagramma.
- Un control node è un activity node usato per coordinare i flussi fra altri nodi.
- Structured activity nodes sono nodi che contengono altri nodi, ovvero presentano una struttura particolare.
 - Il conditional node è un structured activity node che rappresenta una scelta esclusiva fra un certo numero di alternative (è una specie di if).
 - Un loop node è un structured activity node che rappresenta un loop con setup, testo e body (è una specie di for).
 - Una expansion region è un structured node che prende in input una collezione e su ogni elemento di questa agisce individualmente producendo come output una collezione. Tale procedimento può avvenire sequenzialmente ("iterative"), concorrentemente ("parallel").
- L'attività può essere partizionata per sottolineare che un determinate azioni hanno alcune caratteristiche in comune. Alle partizioni vengono applicati dei vincoli.
- Interruptable region (regione interrompibile) è un tipo di activity region che definisce un meccanismo per il quale se avviene un certo evento, mentre gli elementi all'interno di tale regione sono attivi, allora queste attività devono essere interrotte.

²Activity Node: un elemento che rappresenta un'azione o un'attività in un diagramma di attività.

- L'Interrupting edge serve a specificare l'evento che fa interrompere la regione ed eventualmente l'azione da intraprendere dopo l'interruzione.

6 Sequence Diagram

Questo diagramma fa parte della più comune categoria dei diagrammi di interazione, esso si concentra sullo scambio del messaggio tra un numero di lifeline. Il diagramma di sequenza descrive un'interazione concentrandosi sulla sequenza dei messaggi che sono scambiati e le loro occorrenze specifiche sulle lifeline (usato specialmente per la programmazione ad oggetti). **Elementi**

- La lifeline rappresenta un oggetto o un attore che partecipa all'interazione. È rappresentato da una linea verticale tratteggiata che si estende verso il basso dal nome dell'oggetto o dell'attore.
- Un message è un elemento che definisce uno specifico tipo di comunicazione tra lifeline di un'interazione ed è rappresentato da una freccia. Il messaggio può essere:
 - Una chiamata sincrona: tipicamente rappresenta operazioni di chiamata/invio;
 - Una chiamata asincrona: si manda il messaggio e si continua immediatamente senza aspettare la risposta;
 - Un segnale asincrono: diverso da messaggio asincrono, il messaggio è una richiesta, non un'invocazione di una specifica richiesta
 - Un reply message: il messaggio di risposta ad una operazione call è mostrato come una linea tratteggiata e freccia con la "testa" aperta
 - Un create message: il messaggio di creazione è mandato alla lifeline in modo che lo crei lei stessa.
 - Un delete message: il messaggio di eliminazione è mandato per terminare un'altra lifeline.
- Lost and founds: si spedisce il messaggio a qualcuno che non si vuole modellare oppure si ricevono messaggi da qualcuno di non modellato
- Un interaction fragment è un elemento che rappresenta la più generale unità di interazione, esso descrive cosa avviene in un sistema³. Sono interaction fragment:
 - Occurrence: rappresenta un momento nel tempo (evento) all'inizio o alla fine di un messaggio o all'inizio/fine di una execution.
 - Execution: rappresenta un periodo nella lifetime del partecipante in cui:
 1. Si esegue una unità di comportamento o azione nei limiti della lifetime;
 2. Si manda un signal ad un altro partecipante;
 3. Si aspetta una risposta al message da un altro partecipante.
 - Combined: definisce una combinazione di interaction fragment. Serve per definire una regione del diagramma che può o meno avere luogo.
 - Una interaction use è un interaction fragment che consente di usare (o chiamare) un'altra interazione.

Il diagramma di sequenza serve per modellare il dominio della soluzione, il livello di dettaglio è molto ricco nel descrivere ciò che avviene. Con esso si vuole specificare una possibile traccia di esecuzione, non tutte perché altrimenti si complicherebbe.

6.1 Communication Diagram

I communication diagram quindi servono per fare la stessa cosa dei sequence diagram ma sono meno potenti. In questo caso si usa una numerazione per rappresentare l'orizzonte temporale. La notazione dei messaggi nei communication diagram segue le stesse regole dei sequence diagram.

³Ogni interaction fragment è concettualmente uguale ad un'interazione.

L'evoluzione dei messaggi di sistema ha una sequence expression: essa serve per capire l'ordine temporale dei messaggi.

```
Sequence-expression ::=  
Sequence-term '.' ... ':' message-name  
Sequence-term ::= [integer[name]][recurrence]
```

7 UML State Machine Diagram

I diagrammi a stati di UML mostrano una vista dinamica, fanno parte dei diagrammi comportamentali e sono usati per la progettazione. Illustra gli eventi e gli stati interessanti per un oggetto, e il suo comportamento in reazione a un evento, mostra cioè il ciclo di vita di un oggetto.

Le state machine sono applicate in due modi:

1. Per modellare il comportamento di un oggetto reattivo complesso, in risposta agli eventi: behavioral state machine;
2. Per modellare le sequenze valide delle operazioni, ovvero specifiche di protocollo o di linguaggio: protocol state machine.

Stato: Condizione di un oggetto in un certo intervallo di tempo, caratterizzato da un insieme di reazioni e risposte. Dire che un sistema è in un certo stato vuol dire che offre le stesse risposte di altri nello stesso stato, altrimenti sono in stati differenti. Oggetti dipendenti dallo stato reagiscono in modo diverso agli eventi a seconda del loro stato. Gli stati sono rappresentati con rettangoli arrotondati, ma possono essere internamente suddivisi in più scomparti.

Gli stati possono essere:

- Semplici: senza vertici interni o transazioni
- Compositi: contengono una o più regioni, gli stati in queste regioni sono chiamati sottostati
- Descritti da macchine

Gli stati possono essere associati ai seguenti comportamenti:

- Enter: si attiva quando il sistema entra in uno stato;
- Exit: si attiva quando il sistema esce da uno stato;
- doActivity: si attiva quando il sistema è nello stato.

Transizione: In un diagramma gli stati sono collegati da transizioni. La transizione è una relazione tra due stati che descrive in modo atomico che quando si verifica un evento, l'oggetto passa da uno stato all'altro. Una transizione può essere caratterizzata anche da una guardia condizionale, ovvero un test booleano.

Le transizioni sono atomiche, cioè il token o si trova allo stato sorgente o allo stato destinazione, ma una transizione può essere raggiunta, attraversata o completata. Le guardie sono valutate prima che una transizione abbia luogo, a meno che non partano da uno pseudostato. Le guardie tra stati possono bloccare il token, mentre quelle tra pseudostati no.

Evento: avvenimento significativo o degno di nota, cioè un'occorrenza osservabile che avviene in un momento specifico.

Stato Finale: definisce il termine del comportamento modellato attraverso il diagramma.

Pseudostato: è simile al nodo di controllo del flusso, ma ha una semantica diversa: non è in grado di trattenere il token. Sono pseudostati:

- Initial: Quando parte il sistema viene messo un token nello pseudostato iniziale, la transizione all'uscita da questo pseudostato non può essere associata né un evento né una guardia

- Join: È il duale del fork, serve per rimettere insieme due comportamenti che provengono da regioni ortogonali differenti;
- Fork: Divide una transizione in entrata in una o più transizioni⁴.
- Choice: il token entra ed esce in una delle due parti, alternativamente.
- History: usato per tenere traccia dello stato di una regione. Permettono di uscire da una regione, per poi rientrarvi con lo stesso stato che si aveva al momento dell'uscita.

8 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.

9 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:

9.1 Creator

Problema: Chi è *responsabile* dell'istanziamento 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*

9.2 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

9.3 Controller

Problema: Quale oggetto è *responsabile* di gestire una system operation⁵?

Soluzione: Si assegna la responsabilità a una classe che rappresenta:

⁴In genere è meglio non usare join e fork.

⁵*System Operation*: invocazione di un metodo che corrisponde all'attivazione di qualcosa nella UI.

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

9.4 Low Coupling

Problema: Come si progetta un sistema con:

- poche dipendenze;
- 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.

9.5 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

9.6 Pure Fabrication

Problema: Cosa si fa quando nessuna classe ha le informazioni necessarie per soddisfare una *responsabilità*⁶?

Soluzione: Si crea una classe artificiale (non ispirata al dominio) per mantenere la coesione, ridurre l'accoppiamento e supportare altri principi come Single Responsibility.

9.7 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.

9.8 Polimorfismo

Problema: La variazione condizionale causata da statement del *control flow* produce codice difficile da leggere⁷.

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

9.9 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.

10 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.

⁶Ovvero quando non c'è un information expert.

⁷Il 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){...}`

10.1 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 passare⁸;

10.2 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;

10.3 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;

10.4 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 pianificazione 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

⁸Al posto di scrivere il codice e poi eseguire test su di esso.

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;

11 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 dietro⁹.

	Contro	Pro
<i>white</i>	Complesso	Copertura maggiore Si acquisisce conoscenza sul codice creando i test
<i>black</i>	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 SUT¹⁰ 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 indispensabile per poter far funzionare i test. Una classe testabile deve essere associata ad un'interfaccia.

12 Design Pattern

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder	Adapter Bridge	Chain of Responsibility Command

⁹Si testano punti di discontinuità, dei valori casuali attorno ad essi e tutte le combinazioni possibili dei parametri.

¹⁰SUT: System Under Testing

		Prototype Singleton	Composite Decorator Facade Flyweight Proxy	Iterator Mediator Memento Observer State Strategy Visitor
--	--	------------------------	--	---

12.1 Pattern Strutturali

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'è 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.

1. **Facade (GoF):** Si usa quando è necessaria un'interfaccia comune e unificata verso un insieme disparato di implementazioni o interfacce all'interno di un sottosistema. Il problema sorge quando c'è un accoppiamento indesiderato verso molti elementi interni del sottosistema, o quando l'implementazione del sottosistema è suscettibile al cambiamento. Con Facade si ha un unico punto di contatto verso il sottosistema. Fornisce protected variations rispetto ai cambiamenti nell'implementazione di un sottosistema e supporta low coupling tramite un oggetto Indirection¹¹. Le facade sono accessibili tramite Singleton. Un Adapter che nasconde un sistema esterno può anche essere considerato una forma di Facade.
2. **Proxy:** Permette di intercettare e controllare l'accesso ad un oggetto per indirizzare problemi ortogonali¹². Il proxy fornisce il sostituto o un segnaposto per un altro oggetto per controllare l'accesso ad esso, aggiungendo un livello di indirection. Fornisce protected variations e low coupling.
3. **Decorator (GoF):** Si usa per aggiungere responsabilità ad un oggetto interno tramite incapsulamento. I decorator forniscono una flessibilità maggiore rispetto all'ereditarietà perché

¹¹Indirection: pattern GRASP che introduce di un livello intermedio tra due entità, con lo scopo di disaccoppiarle, aumentare la flessibilità, o ritardare una decisione di binding.

¹²I problemi ortogonali sono problemi significativi che non fanno parte del dominio del problema.

possono essere combinati. Un Decorator interpone un servizio sull'oggetto incapsulato. Applica il principio di indirection.

4. **Adapter (GoF):** 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. Adapter coinvolge quattro componenti principali:

- Target: l'interfaccia attesa dal client.
- Client: utilizza l'interfaccia Target.
- Adaptee: la classe esistente con un'interfaccia incompatibile.
- Adapter: adatta l'interfaccia di Adaptee a quella di Target.

Fornisce un livello di indirection tra il client e il componente adattato. Supporta protected variations dato che protegge i client o le parti interne del sistema dalle variazioni o instabilità nelle interfacce esterne. Riduce l'accoppiamento mantenendo il client accoppiato solo all'interfaccia stabile dell'Adapter. Può essere considerato una forma di Pure Fabrication.

L'Adapter si concentra sulla conversione delle interfacce tra componenti esistenti con interfacce non corrispondenti, mentre Facade si concentra sulla semplificazione dell'accesso a un sottosistema complesso, fornendo un'interfaccia unificata e di alto livello.

5. **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.
6. **Composite:** Si usa per gestire un gruppo o una struttura di composizione di oggetti allo stesso modo (polimorficamente). Si definiscono classi per oggetti compositi e atomici in modo che implementino la stessa interfaccia. Sia gli oggetti individuali (chiamati anche "oggetti atomici" o "foglia") che gli oggetti che rappresentano un gruppo di questi oggetti implementano questa stessa interfaccia. Un oggetto composito contiene una collezione di altri oggetti che implementano la stessa interfaccia. Questa collezione può includere sia oggetti atomici che altri oggetti compositi, creando così una struttura ad albero. Gli elementi dell'albero hanno due ruoli distinti: ruolo intermedio che rimanda ad altri elementi e ruolo terminale. È basato sul principio del Polymorphism, poiché oggetti di tipi diversi (atomico e composito) rispondono allo stesso messaggio tramite un'interfaccia comune. Fornisce protected variations, proteggendo il cliente dalla variabilità nella struttura (singolo oggetto vs. gruppo).
7. **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. Il flyweight ha una singola (single responsibility) responsabilità: rappresentare lo stato intrinseco. I client sono disaccoppiati dalla logica di condivisione: non gestiscono come i flyweight sono riutilizzati, favorendo low coupling.

12.2 Pattern Creazionali

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.

1. **Abstract Factory (GoF):** Al posto di usare new, si delega la creazione dell'oggetto a una classe a parte, la Factory. Le factory separano il client¹³ dal processo di istanziazione e delegano la

¹³Con client si intende qualsiasi componente (classe, modulo, funzione, sistema...) che usa un altro componente per ottenere un servizio o una funzionalità.

creazione dell'oggetto a un'interfaccia comune. Questa è una dipendenza di buona qualità perché vi si possono solo aggiungere metodi. La abstract factory si usa per creare famiglie di oggetti correlati che implementano un'interfaccia comune, favorendo la variabilità (protected variations).

2. **Factory:** Consente a una classe di delegare l'istanziamento di oggetti a sottoclassi, permettendo così di decidere quale classe concreta istanziare al momento dell'esecuzione. Si basa sull'ereditarietà: le sottoclassi sovrascrivono il metodo factory per creare oggetti specifici. Permette alle sottoclassi di modificare il tipo di oggetto creato. Viene utilizzato per separare la responsabilità della creazione complessa in oggetti ausiliari coesi. È un oggetto *pure fabrication* la cui responsabilità è gestire la creazione di altri oggetti¹⁴. Garantisce Low Coupling e aumenta il potenziale di riuso. Dato che in genere serve una sola istanza delle factory, sono spesso accessibili tramite il pattern Singleton. Factory è una semplificazione delle Abstract Factory: al posto di coordinare la creazione di famiglie di oggetti correlati, si concentra sulla creazione di un unico oggetto complesso.
3. **Singleton (GoF):** 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. Singleton è spesso utilizzato per gli oggetti Factory e Facade, ed è una Pure Fabrication.
4. **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.
5. **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.

12.3 Pattern Comportamentali

1. **Template Method (GoF):** Si definisce lo scheletro di un algoritmo in un metodo¹⁵, che contiene a sua volta chiamate a metodi astratti (hook methods) che verranno implementati nelle sottoclassi. È comune che il Template Method sia *public*, mentre gli "hook methods" siano *protected*. 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.
2. **Strategy (GoF):** Consente di separare un oggetto da parte del suo comportamento e cambiarlo a runtime. Si definisce una serie di algoritmi incapsulati tra loro intercambiabili: ogni algoritmo è definito in una classe separata (pure fabrication), ma hanno un'interfaccia comune. Strategy favorisce l'implementazione di OCP e obbedisce PV. Inoltre favorisce la composizione rispetto all'ereditarietà. Usa il polimorfismo e favorisce il low coupling.
3. **State (GoF):** State è come Strategy solo l'oggetto cambia il suo comportamento in base al suo stato (interno). Si definiscono classi separate per ciascuno stato. L'oggetto il cui comportamento è stato-dipendente (*context*) mantiene un riferimento all'oggetto che rappresenta il suo stato corrente. Quando il *context* riceve una richiesta la cui gestione dipende dallo stato, delega l'esecuzione specifica a questo oggetto stato. Le classi di stato concrete sono un esempio di Pure Fabrication. La loro creazione supporta High Cohesion e Low Coupling.
4. **Observer (GoF):** Si usa quando diversi tipi di oggetti "sottoscrittori" (subscriber) sono interessati ai cambiamenti di stato o agli eventi di un oggetto "editore" (publisher), e desiderano

¹⁴"Pure Fabrication" è una classe inventata per la convenienza del designer, non ispirata direttamente da un concetto del dominio

¹⁵Questo metodo è il *template method*, in genere si trova in una classe astratta.

reagire in modo unico quando l'editore genera un evento: si propagano le modifiche di una classe su una serie di oggetti. Gli oggetti interessati ricevono la notifica del cambiamento, non viceversa. Si crea una dipendenza uno a molti, così quando uno cambia stato, tutti i dipendenti sono notificati. Favorisce il disaccoppiamento. Observer è basato sul Polimorfismo: L'editore interagisce con i suoi sottoscrittori tramite un'interfaccia comune, e il comportamento effettivo (come reagire all'evento) è polimorfico, definito nelle classi concrete dei sottoscrittori.

L'Observer fornisce Protected Variations e Low Coupling. Se Facade è usato per la collaborazione da un layer inferiore ad uno superiore, l'Observer è usato per la comunicazione da un layer superiore ad uno inferiore.

5. **Memento**: Si usa per catturare ed externalizzare 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
6. **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.
7. **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. Inserendo un intermediario si favorisce il principio di Indirection.
8. **Visitor**: 'l'Iterator, ma usa *Inversion of Control*¹⁶. 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.
9. **Command (GoF)**: Si incapsula una richiesta come un oggetto Command che contiene tutte le informazioni necessarie sull'azione da eseguire. Così si parametrizzano i client con code e operazioni. Assomiglia a Visitor. Command permette di ritardare, mettere in coda le richieste. Command viene spesso utilizzato insieme al pattern Composite.
10. **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.
11. **Interpreter**: Si definisce la grammatica di una lingua con una struttura ad albero. Favorisce modularità del codice ed estensione.

12.4 Pattern Moderni

1. **Hollywood Principle**: Si incoraggia a scrivere codice che non dipende da implementazioni specifiche, ma piuttosto da interfacce o astrazioni. Un componente di basso livello non controlla il flusso, ma è invitato a collaborare da un componente di livello superiore solo quando serve. Questo principio è spesso associato all'Inversione di Controllo¹⁷

¹⁶ToC: Si inverte il flusso di controllo rispetto alla programmazione tradizionale: non è il codice dell'applicazione a controllare il flusso, ma è il framework o l'ambiente a chiamare il codice.

¹⁷Sarà B ad attivare un comportamento di A, e non A ad attivarsi da sola.

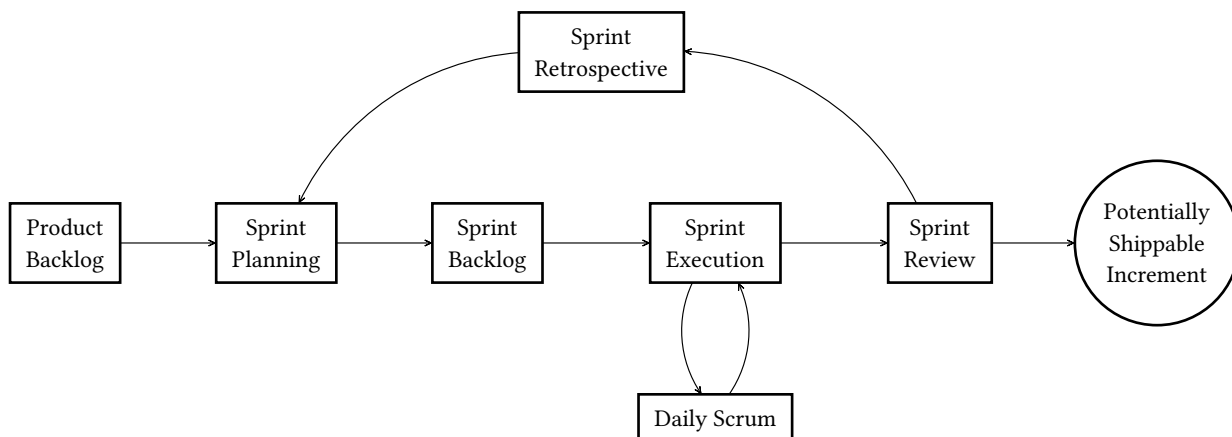
2. **Null Object:** Si usa per specificare che un parametro non è inizializzato. Si ritorna un oggetto convenzionale al posto di null per far capire che l'oggetto non è istanziato. Il vantaggio di usare questa istanza convenzionale è che si può specificare questo oggetto implementando delle funzionalità di default per vari metodi.
3. **Dependency Injection (DI):** Si forniscono le dipendenze ad un oggetto dall'esterno piuttosto che crearle internamente (basato su IoC). Si riduce il coupling, facilitando il testing¹⁸ e la modularità. Si fa injection, ovvero si passano gli oggetti come argomenti di costruttore, metodi setter, interfacce o un metodo specifico.
4. **Clean Dependency Injection:** Sono delle linee guida per implementare DI aderendo ai principi SOLID, in particolare SRP e DIP.
 - dipendenze che non cambiano nel ciclo di vita dell'istanza → *constructor injection*;
 - dipendenze necessarie durante l'invocazione del metodo → *dependency injection*;
 - quando il binding (scelta dell'implementazione concreta) deve essere fatto a runtime, si passa una factory invece dell'oggetto stesso.

Clean DI avviene solo nella *composition root*,¹⁹ tutto il resto del codice dipende da astrazioni (interfacce).

13 Scrum

Scrum è una metodologia per lo sviluppo software in aziende di piccole-medie dimensioni. Fa parte dei metodi AGILE, ma se non è una metodologia completa: si focalizza sugli aspetti di gestione del progetto ed è infatti comune usare Scrum con altre pratiche del mondo AGILE.

Scrum ha un ciclo di vita iterativo: la fase di sprint si ripete più volte fino alla realizzazione di un prodotto potenzialmente rilasciabile.



13.1 Ruoli

- **Core (committed):**
 - *product owner*: proiezione degli stakeholder all'interno del progetto. Rappresenta le richieste del cliente e ha il compito di decidere le priorità, deadline e specifiche del progetto. Ha potere di veto sul lavoro che è stato svolto.
 - *scrum master*: figura senior all'interno dell'organizzazione, attiva nello sviluppo, con buona conoscenza di scrum e al servizio dei colleghi. È responsabile della corretta applicazione delle pratiche scrum all'interno del progetto.

¹⁸Si elimina la dipendenza da new e Factory.

¹⁹La Composition Root è l'unico punto del sistema dove si conosce l'implementazione concreta di un'interfaccia.

- *development team*: team composto da 5-9 persone che si occupa di realizzare le task. Le task sono l'insieme di attività che servono per ogni interazione, non imposte dal product owner, ma autoassegnate per realizzare il progetto. In questo team devono esserci tutte le figure necessarie per completare il progetto, non si vogliono avere dipendenze esterne per evitare tempi di attesa.
- **Additional** (involved):
 - clienti
 - *executive manager*

13.2 Artefatti

Product backlog: Lista di tutte le cose che devono essere fatte nel progetto. Verrà completato in più iterazioni. In questa lista, curata dal *product owner*, ci sono:

- Requisiti funzionali (user stories)
- bugfix
- requisiti non funzionali
- chores: elementi che vengono inseriti dai membri del team e che servono a produrre valore al team di sviluppo e non ai clienti, come aggiornare l'ambiente di sviluppo.

Storie, epiche e temi

Sprint Backlog: tutto ciò che deve essere fatto all'interno dello sprint in un dato intervallo di tempo. Ad ogni task è associato un tempo per completarla (in genere inferiore ad una giornata).

Burn down chart: diagramma che mostra come nel tempo varia il numero di task e le ore disponibili.

13.3 Sprint Planning

Momento in cui si stabilisce come riempire lo sprint backlog. Solitamente si fa una riunione di 1-2 giorni e ci si organizza in due fasi:

1. Product owner definisce gli obiettivi e presenta gli elementi essenziali che vuole all'interno del prodotto e per ogni elemento viene indicato dettaglio e le richieste, stimando l'impegno e tempo necessario.
2. (Solo dal team) si selezionano gli elementi da dividere in task, popolando poi lo sprint backlog.

13.4 Scrum Estimation

Si stabilisce quante task ad altra priorità scegliere. Le task si stimano in ore, mentre gli elementi nel product si valutano con le user stories. Queste vengono valutate in base alla loro complessità e agli story points che misurano il valore prodotto. L'assegnazione delle user stories avviene con il planning poker.

Il numero di storie da scegliere per completare lo sprint backlog può essere in base a:

- **Capacity driven planning**: le storie vengono analizzate in base al tempo richiesto ed effort.
- **Velocity driven planning**: le storie vengono scelte dal backlog in base ai *story points* associati. Questa stima è accurata per team che conoscono la loro *velocity*²⁰.

13.5 Daily Scrum

Riunione giornaliera di 15 minuti dove ogni membro del team dice

- cosa ha fatto ieri per contribuire allo sprint;
- cosa farà oggi per contribuire allo sprint;
- se prevede ostacoli per il raggiungimento del goal dello sprint.

²⁰Velocity: metrica di avanzamento che indica quanti story point macina un team di sviluppo.

13.6 Sprint Review

Alla fine di uno sprint si fa la review con tutti i membri del team e gli stakeholder, dove viene presentato l'incremento con tutti i problemi e soluzioni. Si discute anche delle tempistiche di consegna.

13.7 Retrospecting

Dopo allo *sprint review*, *scrum master* e team di sviluppo discutono delle problematiche riscontrate nello sviluppo, cercando di identificare miglioramenti prima del prossimo sprint.

13.8 Scrum Considerato Dannoso/Pericoloso

In caso di assenza dello *scrum master*, il compito viene svolto da uno o più membri del team di sviluppo. A rotazione, questo però potrebbe anche essere un vantaggio se dimostrato praticamente.

13.9 Kanban

Kanban è un metodo di gestione del lavoro che nasce nella produzione industriale, adattato allo sviluppo software da David Anderson. Si basa su un approccio snello (lean) e just-in-time, cioè produce valore in modo continuo e sostenibile, limitando gli sprechi e migliorando il flusso di lavoro. A differenza di Scrum che lavora a sprint, Kanban non ha iterazioni fisse: il lavoro avanza in modo continuo lungo una pipeline, composta da più stadi come:

- To Do;
- In Progress;
- Testing;
- Done

Ogni elemento di lavoro (feature, bug, task) scorre attraverso questi stadi fino alla conclusione.

La Kanban board rappresenta visivamente la pipeline e consente al team di:

- vedere lo stato attuale del lavoro;
- identificare colli di bottiglia;
- limitare il lavoro simultaneo.

C'è un limite al lavoro in corso (WIP limit): Ogni stadio della pipeline ha un numero massimo di elementi che può contenere. Se uno stadio è pieno, non si possono aggiungere altri elementi finché non se ne libera uno. Questo previene colli di bottiglia, promuove il focus, e mantiene un ritmo di lavoro costante.

In Kanban, la metrica principale è il Cycle Time: il tempo medio che un task impiega per attraversare tutta la pipeline, dal primo stadio a "Done". Il WIP (Work in Progress) è invece la quantità attuale di lavoro in ogni stadio, e serve a monitorare il carico di lavoro del team.

14 Modello di Analisi

L'obiettivo è capire cosa deve fare il sistema, comunicare lo sviluppo al cliente e definire una serie di requisiti validabili.

Gli **artefatti** analizzano punti diversi dello sviluppo: richieste, specifiche, glossario e modello dominante.

I **requisiti** specificano quello che il software deve fare e i vincoli che deve rispettare. Sono indipendenti dall'implementazione.

Per scrivere un requisito formale si usa un linguaggio il meno ambiguo possibile.

1. Se l'utente ha inserito le credenziali corrette (condition)
2. il sistema (Subject)
3. deve consentire l'accesso (Action)
4. al pannello di controllo (Object)
5. entro 3 secondi (Constraint)

Un requisito deve essere non ambiguo, coinciso, completo, singolare, fattibile, verificabile e tracciabile.

Un gruppo di requisiti deve essere: completo, consistente, coveniente e vincolato.

	Tipi di requisiti	
	Funzionali	Non Funzionali
Obiettivo	Descrive cosa fa il prodotto	Descrive come funziona il prodotto
Risultato finale	Definisce le funzionalità del prodotto	Definisce le proprietà del prodotto
Focus	Requisiti dell'utente	Aspettative dell'utente
Origine	Utente	Sviluppatore
Testing	Prima dei non funzionali	Dopo i funzionali

Tassonomia **FURPS+** per i requisiti:

- *Functional*;
- *Usability*: sforzo cognitivo di un utente per giungere all'obiettivo;
- *Reliability*;
- *Performance*;
- *Supportability*: manutenibilità;
- *+*: Implementazione, interfaccia e packaging.

14.1 Object Oriented Principles

- Astrazione: ci si concentra sulle caratteristiche essenziali di un oggetto;
- Incapsulamento: si nascondono i dettagli di implementazione e si espongono solo le interfacce necessarie;
- Ereditarietà: il comportamento e lo stato può essere specializzato, riutilizzando il codice e creando una gerarchia;
- Polimorfismo: si possono usare oggetti di classi diverse in modo intercambiabile, purché implementino la stessa interfaccia.

	Descrizione	SOLID					Grasp					Related Patterns	Pattern Type	
		Single Responsibility GoF	Open-Closed	Liskov Substitution	Interface Segregation	Dependency Inversion	Low Coupling	High Cohesion	Pure Fabrication	Indirection	Polymorphism			Protected Variations
Facade	Interfaccia comune e unificata	✓	?			?	✓			✓	✓		Singleton, Adapter	Structural
Proxy	Intercettare e controllare l'accesso ad un oggetto		?	?	?		✓			✓	✓			Structural
Decorator	Aggiungere responsabilità ad un oggetto tramite incapsulamento	✓			?					✓				Structural
Adapter	Permettere a classi con interfacce incompatibili di collaborare	✓	?	?		?	✓		✓	✓	✓		Facade	Structural
Bridge	Separare un'astrazione dalla sua implementazione		?	?		?							Adapter	Structural
Composite	Gestire un gruppo di oggetti allo stesso modo										✓	✓		Structural
Flyweight	Ridurre l'uso di memoria condividendo dati tra oggetti simili		?	?			✓							Structural
Abstract Factory	Creare famiglie di oggetti correlati che implementano un'interfaccia comune	✓	?	?	?	?	✓					✓		Creational
Factory	Delegare l'istanziazione di un unico oggetto complesso a sottoclassi						✓		✓				Singleton, Abstract Factory	Creational
Singleton	Sola istanza di una classe con punto di accesso globale	✓							✓				Factory, Facade	Creational
Builder	Costruire oggetti complessi passo per passo													Creational
Prototype	Uso istanze di oggetti esistenti per creare nuovi tipi													Creational
Template Method	Lo scheletro dell'algoritmo è definito in un metodo che a sua volta chiama metodi astratti	✓		✓										Behavioural
Strategy	Sostituire un algoritmo con un altro a runtime in base a condizioni esterne	✓		✓			✓		✓		✓	✓		Behavioural
State	Come strategy ma con condizioni interne	✓					✓	✓					Strategy	Behavioural
Observer	I subscriber ricevono la notifica del cambiamento dal publisher: si propagano le modifiche	✓					✓				✓	✓	Facade	Behavioural
Memento	Esternalizzare lo stato di un oggetto per ripristinarlo in un momento successivo													Behavioural
Iterator	Accedere agli elementi di un oggetto aggregato senza esporre la sua rappresentazione interna													Behavioural
Mediator	Oggetto che incapsula come insieme di oggetti interagiscono.						✓			✓				Behavioural
Visitor	Iterator ma con IoC											✓	Iterator	Behavioural
Command	Richiesta incapsulata come oggetto command	✓							✓				Visitor, Composite	Behavioural
Chain of Responsibility	Si passa una richiesta attraverso una catena di handler													Behavioural
Interpreter	Si definisce la grammatica di una lingua con una struttura ad albero													Behavioural
Hollywood Principle	Scrivere codice che dipende da interfacce o astrazioni											✓		Modern
Null Object	Specifica che un parametro non è inizializzato													Modern
Dependency Injection	Fornire le dipendenze ad un oggetto dall'esterno						✓					✓		Modern
Clean Dependency Injection	Implementazione di DI aderendo a SRP e DIP		✓			✓							Dependency Injection	Modern