**Si illustrino le differenze e le similarità tra: la relazione di uso e la relazione di ereditarietà.**

la relazione di ereditarietà e la relazione di composizione tra i moduli di un progetto software:  La relazione di uso e la relazione di ereditarietà A USES B significa che: A puo accedere ai servizi esportati da B tramite la sua interfaccia;  A dipende da B per fornire a sua volta i propri servizi: se B non corretto, allora anche A non corretto. In un sistema orientato agli oggetti, la relazione INHERITS-FROM permette a un componente di estenderne un altro. In pratica, questa relazione una forma più forte di USES, perché l’erede può accedere ad (alcuni) dei servizi privati del componente da cui eredita. INHERITS FROM: per i sistemi orientati agli oggetti.IS-COMPONENT-OF: descrive l’aggregazione di moduli in altri moduli di livello più alto (perché i moduli sono raramente unitari).

**Si descriva che cosa sono componenti e connettori in un progetto software e si diano esempi di componenti e di connettori:**

Componente → è un’unità concettuale di decomposizione di un sistema a tempo di esecuzione. E’ caratterizzata dalla sua interfaccia, definita in termini di porti. Caratteristica delle componenti è la rimpiazzabilità: è possibile sostituire una componente di un sistema con un’altra che abbia la stessa Domande esami 2 interfaccia, senza pregiudicare il funzionamento dell’intero sistema. Connettore → è un canale di interazione tra componenti. Un connettore modella un protocollo, un flusso d’informazione, un modo di accedere a un deposito dati, etc. Un esempio di Componenti e Connettori rispettivamente sono le Classi e le Associazioni nel Diagramma delle Classi relativo all’UML.

**Si illustrino le caratteristiche degli State Diagram in UML:**

 Gli state diagram UML sono diagrammi dinamici che permettono di descrivere automi a stati finiti, usando una notazione grafica derivata da una proposta precedente (StateCharts). I principali elementi di tale notazione sono: stato stato iniziale; stato finale; transizione; decomposizione OR;  decomposizione AND;

**.Si illustrino le caratteristiche dei diagrammi di struttura in UML:**

Vista Strutturale → rappresenta gli elementi strutturali necessari per implementare la soluzione ai requisiti del sistema. Definisce: elementi di analisi a oggetti e di progetto; il vocabolario del dominio e della soluzione; la decomposizione del sistema in strati e sottosistemi e le interfacce del sistema e i suoi componenti.

**Qual è l'obiettivo principale della fase di progettazione "in-the-large"?**

L'obiettivo principale della progettazione "in-the-large" è gestire la complessità del problema nello sviluppo di un prodotto software. Questo viene realizzato decomponendo il problema principale in sotto-problemi più piccoli e gestibili. La somma delle complessità dei sotto-problemi deve risultare inferiore alla complessità del problema iniziale. Questo approccio aiuta a ridurre il rischio di errori, migliorare la manutenibilità e facilitare il lavoro di sviluppo.

**2. Spiega il concetto di "Program Family" e fornisci un esempio.**

Il concetto di "Program Family" si riferisce alla progettazione di un programma come membro di una famiglia di varianti simili. L'obiettivo è progettare l'intera famiglia piuttosto che i singoli membri separatamente. **Esempio:** Un sistema di prenotazione per strutture che può essere adattato per hotel (prenotazione di stanze, ristoranti, sale conferenze) o università (prenotazione di aule o attrezzature). Sebbene molte funzionalità siano comuni, alcune differenze devono essere considerate, come l'uso gratuito o a pagamento delle risorse

**3. Quali sono le principali relazioni tra i moduli e qual è il loro significato?**

**USES:** Indica che un modulo utilizza i servizi esportati da un altro modulo. È una relazione di dipendenza in cui un modulo (cliente) dipende dalla qualità del modulo fornitore. **IS\_COMPONENT\_OF:** Descrive un modulo di livello superiore composto da moduli di livello inferiore. **INHERITS:** Presente nei sistemi orientati agli oggetti, consente a un modulo di estendere un altro, accedendo ad alcune delle sue parti segrete.

**4. Quali sono i principi chiave nella progettazione dei moduli**

**Cohesion e Coupling:** Massimizzare la coesione interna del modulo e minimizzare le interconnessioni con altri moduli. **Information Hiding:** I moduli devono nascondere i propri dettagli interni, esponendo solo un'interfaccia stabile. **Stable Interfaces:** Le interfacce devono rappresentare contratti chiari con i moduli clienti e non essere soggette a cambiamenti frequenti.

**Cosa si intende per "Information Hiding" e perché è fondamentale?**

L'**Information Hiding** si riferisce al principio di progettare un modulo attorno agli elementi che devono rimanere nascosti agli utenti (segreti). Questo principio; Incrementa la manutenibilità: i cambiamenti interni non influenzano i clienti. Migliora la modularità: facilita la sostituzione o il miglioramento di parti del sistema senza interrompere il resto. Stabilizza le interfacce: garantendo che i clienti interagiscano solo con ciò che è essenziale.

**Descrivi il modello architetturale a "Pipes and Filters". Quali sono i suoi vantaggi e svantaggi?**

Il modello "Pipes and Filters" prevede una serie di componenti (filtri) che trasformano i dati in ingresso e li passano attraverso canali (pipe) ad altri filtri.**Vantaggi:** Composizionalità: il comportamento complessivo è una combinazione dei comportamenti dei singoli filtri. Riutilizzo: i filtri possono essere combinati in modi diversi. Facilità di modifica: è possibile aggiungere o sostituire filtri **Svantaggi:** Mancanza di persistenza: i dati devono fluire continuamente attraverso il sistema. Tendenza a strutture batch: difficoltà a gestire sistemi interattivi.

**Qual è la differenza tra meccanismi e stili in un'architettura software?**

**Meccanismi:**Riguardano la costruzione dell'architettura (ad esempio, moduli e loro relazioni). **Stili:** Definiscono il carattere distintivo dell'architettura (ad esempio, un'architettura a livelli rispetto a una pipeline). Un esempio pratico: i meccanismi descrivono i singoli componenti di un'auto, mentre gli stili distinguono tra una coupé e un SUV.

**Quali sono i principali principi di progettazione e come contribuiscono alla qualità del software?**

**Decomposizione:** Suddivide il sistema in sottosistemi più semplici. **Abstraction:** Nasconde i dettagli irrilevanti per concentrarsi sugli aspetti rilevanti. **Modularity:** Facilita la sostituzione e l'aggiornamento delle parti del sistema. **Extensibility:** Permette di aggiungere nuove funzionalità con impatti minimi. **Information Hiding:** Protegge i dettagli implementativi. **Hierarchy:** Organizza il sistema attraverso livelli di astrazione. **Program Families:** Consente la progettazione orientata alla riusabilità. Questi principi riducono la complessità, migliorano la manutenibilità e favoriscono la riusabilità.

**Cos’è un’interfaccia in un modulo e perché è importante?**

L’interfaccia di un modulo è l’insieme dei servizi che il modulo esporta verso altri. Essa rappresenta un contratto stabile tra il modulo e i suoi utilizzatori. È importante perché: Definisce cosa un modulo può fare senza esporre come lo fa. Protegge l’implementazione interna, rendendo possibile modificare il modulo senza impatti sui clienti. Favorisce l’interoperabilità e la modularità del sistema.

**Descrivi i vantaggi dell’organizzazione gerarchica di un sistema.**

**Facilità di comprensione:** La struttura a livelli semplifica l’analisi. **Manutenzione incrementale:** È possibile modificare un livello senza influenzare gli altri. **Riusabilità:** I componenti dei livelli più bassi possono essere riutilizzati in altri sistemi. **Sviluppo modulare:** Ogni livello può essere sviluppato e verificato indipendentemente. Un sistema gerarchico si basa su una rappresentazione a **DAG** (grafo aciclico diretto) per rappresentare i livelli.

**Quali sono i pro e contro dei sistemi basati su eventi?**

**Pro:** Scalabilità: facile aggiungere o rimuovere componenti. Disaccoppiamento: i componenti non devono conoscersi direttamente. Ideale per sistemi distribuiti e stimolo-risposta **Contro:** Complessità nella gestione dell’ordine degli eventi. Difficoltà di debug a causa della mancanza di un flusso lineare. Esempi includono interfacce grafiche e sistemi reattivi distribuiti.

**Qual è il ruolo del concetto di “Virtual Machine Structuring”?**

La **Virtual Machine Structuring** consente di organizzare il sistema attraverso strati, in cui ogni strato funge da macchina virtuale per quello superiore. Questo: Riduce la complessità del sistema. Favorisce l’incapsulamento e l’astrazione. Rende più facile l’aggiunta di funzionalità o l’adattamento a nuovi ambienti.

**Qual è la differenza tra un'architettura funzionale e una orientata agli oggetti?**

**Architettura Funzionale:** Si basa su funzioni e dati comuni. Le funzioni sono indipendenti e accedono a dati condivisi. **Architettura Orientata agli Oggetti:** I dati e le funzioni sono incapsulati in oggetti. Gli oggetti comunicano attraverso metodi. L’orientamento agli oggetti facilita l’uso di concetti come ereditarietà e polimorfismo.

Linguaggio descrittivo e operazionali

**Linguaggio Descrittivo** **Scopo**: Descrivere **cosa** fa un sistema, non **come** lo fa.**Caratteristiche**:Focus sui requisiti e sul comportamento. Non specifica dettagli di implementazione. **Esempi**: Diagrammi UML, specifiche matematiche, DSL (Domain-Specific Languages). **Linguaggio Operazionale** **Scopo**: Descrivere **come** funziona un sistema, passo per passo.**Caratteristiche**:Specifica dettagli esecutivi e procedure. Fornisce una sequenza chiara di operazioni. **Esempi**: Codice sorgente (Java, Python), algoritmi pseudocodificati. **Differenza chiave**:**Descrittivo**: Spiega il *cosa* (livello alto).**Operazionale**: Dettaglia il *come* (livello basso).

Architettura funzionale e a eventi.

**Architettura Funzionale** **Definizione**: Sistema suddiviso in funzioni pure che trasformano input in output senza modificare lo stato globale o causare effetti collaterali. **Caratteristiche**: **Determinismo**: Stesso input, stesso output. **Stato immutabile**: Cambiamenti gestiti creando nuovi dati. **Composizione**: Funzioni complesse create componendo funzioni più semplici. **Concorrenza**: Semplice grazie all’assenza di stato condiviso. **Pro**: Facilità di test e debug, parallelizzazione semplice. **Contro**: Meno naturale per applicazioni con interazioni o stati persistenti. **Architettura a Eventi** **Definizione**: Sistema organizzato intorno a eventi che attivano handler per reagire a cambiamenti o azioni. **Caratteristiche**: **Event-driven**: Basata su produttori e consumatori di eventi. **Asincronicità**: Permette di gestire azioni senza bloccare il flusso. **Callback/Handler**: Funzioni eseguite in risposta a eventi. **Message Queue**: Per la gestione di eventi complessi. **Pro**: Ideale per applicazioni interattive, separazione delle responsabilità, scalabilità. **Contro**: Debug complesso, rischio di perdita di eventi e race condition.

**Spiega l’architettura a strati con un esempio pratico.**

L’architettura a strati organizza il sistema in livelli, dove ogni livello utilizza solo i servizi del livello sottostante.**: Livello 1 (Hardware):** Gestisce l’interazione con l’hardware. **Livello 2 (Sistema Operativo):** Fornisce servizi di base come gestione della memoria. **Livello 3 (Applicazioni):** Programmi che utilizzano i servizi del sistema operativo. Questa struttura è tipica nei sistemi complessi come quelli operativi.

**Quali sono le differenze tra il modello "Repository" e quello "Blackboard"?**

**Repository:** I componenti comunicano solo attraverso un archivio centrale passivo. **Blackboard:** Simile al repository, ma il blackboard è attivo, innescando componenti quando il suo stato cambia. Il modello blackboard è ideale per sistemi basati su AI, come riconoscimento vocale o visione artificiale.

**Qual è l'obiettivo dei modelli nella progettazione del software?**

I modelli sono strumenti utilizzati dai progettisti per rappresentare, analizzare e comunicare i concetti di un sistema, sia esso da costruire o già esistente. **Funzioni principali** Taming complexity: aiutano a gestire la complessità rappresentando solo le parti essenziali. Communication vehicle: forniscono un mezzo visivo per descrivere il sistema. Quality analysis: consentono di analizzare specifiche qualità del sistema. Un buon modello deve soddisfare le seguenti proprietà: **Chiarezza:** Deve rappresentare il sistema in modo comprensibile. **Coerenza:** Evita contraddizioni tra i diversi aspetti del sistema. **Precisione:** Descrive esattamente ciò che è richiesto. **Scalabilità:** Può essere esteso per includere nuovi aspetti o dettagli del sistema. **Astrazione:** Rappresenta solo gli elementi rilevanti, ignorando i dettagli irrilevanti.

**Spiega il concetto di astrazione e il suo ruolo nella modellazione.**

L'astrazione consiste nel descrivere un sistema includendo solo le caratteristiche rilevanti per il contesto, ignorando i dettagli irrilevanti. È fondamentale per: Ridurre la complessità. Concentrarsi sugli aspetti importanti per il problema in esame. Facilitare la comunicazione tra i membri del team. **Esempio:** Per un sistema di prenotazione, l'astrazione può includere nome, data e disponibilità, escludendo dettagli come il design della GUI.

**Quali sono i criteri fondamentali della modularità? Modular decomposability:**

Suddividere un problema complesso in sottoproblemi più semplici. **Modular composability:** Aggregare moduli esistenti per risolvere nuovi problemi. **Modular continuity:** Cambiamenti minimi nelle specifiche devono implicare modifiche minime nei moduli. **Modular protection:** I guasti devono rimanere confinati nel modulo, senza propagarsi.

**Quali sono le caratteristiche ideali di un linguaggio di descrizione?**

**Completezza:** Capacità di descrivere tutti gli aspetti rilevanti del sistema **Accuratezza:** Permette descrizioni precise. **Consistenza:** Aiuta a evitare contraddizioni tra rappresentazioni. **Comprensibilità:** Deve essere facilmente interpretabile da tutti gli stakeholder. **Formalità:** Livello di rigore con cui sono definiti sintassi e semantica.

**Spiega la differenza tra stile descrittivo e operativo nella descrizione di un sistema.**

**Stile descrittivo:** Si concentra sulle proprietà desiderate del sistema, senza specificare il comportamento. È più astratto, ma meno utile per verificare i comportamenti. **Stile operativo:** Descrive il comportamento del sistema tramite una macchina astratta, suggerendo una possibile implementazione

**Quali sono i livelli di formalità dei linguaggi di descrizione?. Qual è il ruolo della formalità nei linguaggi di descrizione?**

La formalità determina il rigore con cui sono definite sintassi e semantica **Informali:** Linguaggi naturali come italiano o inglese. Facili da capire, ma ambigui. **Semi-formali:** Sintassi ben definita, ma semantica ambigua (es. UML). **Formali:** Sia sintassi che semantica sono rigorosamente definite. Permettono analisi automatizzate e verifiche logiche (es. logica, FSM, Petri nets).

**19. Quali sono i benefici di un linguaggio semi-formale come l'UML?**

**Chiarezza visiva:** Facilita la comprensione del sistema tramite diagrammi. **Comunicazione:** Agevola il dialogo tra stakeholder tecnici e non tecnici. **Standardizzazione:** È uno standard riconosciuto, riducendo ambiguità. **Flessibilità:** Supporta molteplici tipi di diagrammi per rappresentare vari aspetti del sistema (es. classi, sequenze, attività).

**Quali sono i vantaggi di utilizzare modelli multipli per rappresentare un sistema complesso?**

Un sistema complesso può essere rappresentato meglio con modelli multipli, ognuno focalizzato su un aspetto specifico: **Struttura statica:** Es. diagrammi delle classi. **Comportamento dinamico:** Es. diagrammi di sequenza. **Organizzazione logica:** Struttura modulare. **Distribuzione fisica:** Architettura dei componenti. Questa diversità migliora la comprensione e facilita il lavoro collaborativo.

**Cosa si intende per "divide et impera" nella modularità**

"Divide et impera" è un principio fondamentale che suggerisce di suddividere un problema complesso in parti autonome, fortemente coese e scarsamente connesse. Questo approccio: Facilita la comprensione e lo sviluppo. Riduce la probabilità di errori. Migliora la manutenibilità.

**Quali sono i pro e i contro dell'utilizzo di linguaggi formali nella descrizione di sistemi software?**

**pro** Eliminano le ambiguità. Consentono verifiche logiche e analisi automatizzate. Facilitano la coerenza tra le descrizioni. **Contro:** Complessi da apprendere e utilizzare. Possono richiedere più tempo e risorse rispetto ai linguaggi semi-formali o informali.

**Spiega il concetto di modularità rispetto alla continuità e protezione.**

**Continuità:** Un piccolo cambiamento nelle specifiche deve causare modifiche minime in uno o pochi moduli, favorendo l'estendibilità. **Protezione:** I guasti devono rimanere confinati al modulo in cui si verificano, senza propagarsi ad altri moduli. Questo migliora la robustezza del sistema.

**Qual è la differenza tra modularità e astrazione?**

**Modularità:** Si concentra sulla decomposizione del sistema in parti autonome, facilitando la manutenibilità e la comprensione. Si basa sul principio "divide et impera". **Astrazione:** Riduce la complessità nascondendo i dettagli implementativi e concentrandosi solo sulle caratteristiche rilevanti. Entrambi migliorano la gestione della complessità, ma la modularità è più focalizzata sulla struttura, mentre l'astrazione riguarda il livello di dettaglio.

**Cosa significa che una descrizione è "completa" in un linguaggio di descrizione?**

Una descrizione è completa quando il linguaggio fornisce gli strumenti per rappresentare tutti gli aspetti del sistema di interesse. Questo include: Struttura statica (es. componenti, relazioni). Comportamento dinamico (es. interazioni, transizioni di stato). Vincoli e proprietà specifiche.

**Quali sono le implicazioni dell’uso dello stile operativo per descrivere un sistema?**

Lo stile operativo descrive il comportamento del sistema come una macchina astratta. Le implicazioni includono: **Pro:** Facilità di esecuzione e prototipazione; utile per verificare il comportamento tramite simulazioni. **Contro:** Difficoltà a ragionare formalmente sulle proprietà del sistema; spinge verso specifiche implementazioni che potrebbero non essere ottimali.

**Perché la comprensibilità è una caratteristica importante nei linguaggi di descrizione?**

La comprensibilità è essenziale perché le descrizioni devono essere utilizzate e modificate da diversi stakeholder, inclusi analisti, sviluppatori e clienti. Se un linguaggio non è facilmente comprensibile: Aumenta il rischio di errori di interpretazione. Rende più difficile la manutenzione e l'evoluzione del sistema.

**Come si applica il concetto di "Information Hiding" alla modularità?**

"Information Hiding" implica che ogni modulo deve nascondere i dettagli interni, esponendo solo un’interfaccia pubblica. Questo: Migliora la manutenibilità, poiché i dettagli possono cambiare senza impatti sui moduli esterni. Favorisce il riutilizzo, poiché altri moduli si interfacciano solo con i servizi pubblici.

**Quali sono le sfide associate alla costruzione di un modello multi-visione?**

**Coerenza:** Garantire che i modelli diversi non siano in conflitto. **Manutenibilità:** Aggiornare tutti i modelli quando cambiano i requisiti. **Integrazione:** Assicurarsi che le diverse visioni possano essere combinate per rappresentare il sistema complessivo.

**. Qual è il ruolo delle viste nella modellazione di sistemi complessi?**

Le viste permettono di rappresentare un sistema complesso tramite diverse prospettive, ognuna focalizzata su un aspetto specifico, come: Struttura statica. Dinamica dei comportamenti. Distribuzione fisica. Questo approccio migliora la comprensione, la gestione e l’analisi del sistema, specialmente in team di grandi dimension

**Qual è la differenza tra vista implementativa e vista ambientale in UML?**

**Vista implementativa:** Descrive i componenti logici (es. file, librerie) e le loro dipendenze. **Vista ambientale:** Focalizzata sulla topologia hardware e sull’installazione dei componenti software.

**Cos’è UML e perché è importante e quali limiti ha ?**

UML (Unified Modeling Language) è un linguaggio visuale standard utilizzato per definire, progettare, realizzare e documentare sistemi software orientati agli oggetti. **Importanza:** Riunisce diversi approcci di modellazione preesistenti  È indipendente dal linguaggio di programmazione. Copre l’intero ciclo di vita del software. È standardizzato dall’OMG (Object Management Group). **Limiti: Ambiguità:** I diagrammi semi-formali possono essere interpretati diversamente. **Complessità:** La creazione e gestione di molti diagrammi può essere impegnativa. **Supporto limitato:** Non copre tutti gli aspetti di alcune applicazioni specializzate. **Soluzioni:** Utilizzare UML insieme a linguaggi formali per eliminare ambiguità. Automatizzare la generazione e l’aggiornamento dei diagrammi con strumenti software. Estendere UML con profili personalizzati per domini specifici.

**Si illustrino le caratteristiche delle famiglie di diagrammi (ovvero le “viste”) di UML**

Le principali viste di un sistema in UML sono pensate per rappresentare i diversi aspetti di un sistema software. **Vista dei casi d’uso:** Mostra il sistema dal punto di vista dell'utente finale, evidenziando i requisiti funzionali. Serve per analizzare i requisiti, comunicare con gli stakeholder e validare le funzionalità principali del sistema. Lo fa usando: **Use Case Diagram**: Rappresenta i casi d’uso (funzionalità principali) e le interazioni tra gli attori (utenti o sistemi esterni). **Sequence Diagram**: Mostra la sequenza temporale delle interazioni tra gli attori e il sistema per un caso d’uso specifico. **Activity Diagram**: Rappresenta il flusso di attività all’interno di un caso d’uso, evidenziando decisioni e iterazioni. **Vista strutturale:** Descrive gli aspetti statici del sistema, come la sua organizzazione interna e le sue componenti. È fondamentale per la progettazione e lo sviluppo del sistema, fornendo una mappa chiara degli elementi costitutivi e delle loro relazioni. Lo fa usando: **Class Diagram**: Illustra le classi, i loro attributi, metodi e relazioni (associazioni, generalizzazioni, aggregazioni, composizioni) **Package Diagram**: Rappresenta i pacchetti (raggruppamenti di classi o componenti) e le loro dipendenze. **Vista comportamentale:** Mostrare il comportamento dinamico del sistema, ovvero come le componenti interagiscono e si evolvono durante l'esecuzione. Serve per comprendere e documentare il funzionamento dinamico e per analizzare scenari di utilizzo. Lo fa usando: **Sequence Diagram**: Mostra lo scambio di messaggi tra oggetti o componenti nel tempo, evidenziando l’ordine delle operazioni. **State Diagram**: Descrive gli stati di un oggetto o componente e le transizioni tra di essi, basate su eventi o condizioni. **Activity Diagram**: Dettaglia il flusso di controllo e delle attività tra oggetti o componenti.Communication diagram. Timing diagram.  **Vista implementativa:** Documentare gli aspetti tecnici del sistema, come la sua suddivisione in moduli e le dipendenze tra di essi.  È utile durante la fase di implementazione e manutenzione per comprendere l’architettura tecnica del sistema. Lo fa usando: **Component Diagram**: Mostra i componenti software (come librerie, file eseguibili, moduli) e le loro interazioni o dipendenze. **Vista ambientale:** Rappresentare l’infrastruttura hardware e la distribuzione del software su di essa. Serve nella fase di deployment per pianificare e ottimizzare l’installazione del sistema e garantirne l’interoperabilità. **Deployment Diagram**: Mostra i nodi hardware e come i componenti software sono distribuiti su di essi, includendo informazioni su protocolli di comunicazione. EXTRA: **Relazioni tra le viste:** Queste viste non sono isolate, ma si intersecano per fornire una descrizione completa del sistema: La **vista dei casi d'uso** guida la progettazione della **vista comportamentale**. La **vista strutturale** informa la **vista implementativa**. La **vista comportamentale** e la **vista strutturale** insieme supportano la progettazione della **vista ambientale**.

**Spiega l’architettura a livelli di UML (M0-M3).**

**M0:** Oggetti reali, istanze del sistema (es. un biglietto o un passeggero). **M1:** Modelli utente definiti dagli sviluppatori (es. classi come "Biglietto", "Passeggero"). **M2:** Metamodelli che definiscono la struttura di UML (es. concetti come "Classe", "Relazione"). **M3:** Meta-metamodelli che definiscono le regole generali per creare metamodelli. Questo approccio aiuta a organizzare e standardizzare la modellazione.

**Qual è la differenza tra diagrammi statici e dinamici in UML?**

**Diagrammi statici:** Descrivono la struttura statica del sistema, come class diagram, object diagram e package diagram. **Diagrammi dinamici:** Mostrano il comportamento e le interazioni tra componenti nel tempo, come sequence diagram, activity diagram e state diagram.

**Quali sono i vantaggi dell’utilizzo di UML in un progetto software?**

**Standardizzazione:** Permette una comunicazione uniforme tra team. **Chiarezza:** Facilita la comprensione di sistemi complessi. **Riusabilità:** Favorisce il riutilizzo di modelli per progetti futuri **Estensibilità:** Può essere adattato a diversi domini applicativi. **Indipendenza:** Non è vincolato a linguaggi di programmazione specifici.

**Come UML supporta la rappresentazione di sistemi distribuiti?**

Per rappresentare sistemi distribuiti, UML utilizza: **Deployment diagram:** Per descrivere la topologia hardware e l’assegnazione di componenti software ai nodi. **Sequence diagram:** Per mostrare la sequenza di messaggi tra componenti distribuiti. **State diagram:** Per rappresentare stati e transizioni in ambienti distribuiti.

**Quali sono i principali vantaggi dei diagrammi UML dinamici?**

**Chiarezza nelle interazioni:** Mostrano come i componenti collaborano per realizzare un caso d’uso. **Identificazione dei colli di bottiglia:** Aiutano a individuare problemi di performance o accoppiamento. **Validazione dei requisiti:** Consentono di verificare che i comportamenti desiderati siano rispettati.

**Qual è il significato di "stile misto" in UML?**

Lo stile misto di UML combina approcci dichiarativi e operazionali: **Dichiarativo:** Si concentra sulle proprietà statiche del sistema (es. class diagram). **Operazionale:** Rappresenta il comportamento dinamico del sistema (es. sequence diagram, activity diagram). Questo approccio bilancia la rappresentazione statica e dinamica per offrire una visione completa del sistema.

**Come UML supporta la modellazione orientata agli oggetti?**

UML supporta la modellazione orientata agli oggetti grazie a: **Class diagram:** Per rappresentare classi e relazioni. **Inheritance e polymorphism:** Per descrivere gerarchie di classi. **Encapsulation:** Tramite visibilità pubblica, privata e protetta.

**Quali sono i vantaggi dell’indipendenza del linguaggio di programmazione in UML?**

**Adattabilità:** Può essere usato in qualsiasi progetto, indipendentemente dal linguaggio scelto. **Comunicazione universale:** Facilita la collaborazione tra team con competenze diverse. **Riutilizzo:** I modelli possono essere tradotti in diversi linguaggi di implementazione.

**Qual è la differenza tra activity diagram e state diagram?**

**Activity diagram:** Rappresenta il flusso di attività o processi, con enfasi sul parallelismo e sulle decisioni. **State diagram:** Mostra gli stati di un oggetto e le transizioni causate da eventi. **Esempio:** Un activity diagram può descrivere il processo di login, mentre un state diagram può rappresentare gli stati di autenticazione (es. "Autenticato", "Non autenticato").

**Quali sono le differenze tra component diagram e composite structure diagram?**

**Component diagram:** Mostra i componenti fisici del sistema (es. file, librerie) e le loro relazioni **Composite structure diagram:** Descrive l’organizzazione interna di un componente o una classe, evidenziando le parti e le interazioni interne.

**PARLA DELLO USE CASE ?**

Un Use Case Diagram è una rappresentazione grafica che descrive le **funzionalità principali di un sistema** e le sue interazioni con attori esterni. il suo scopo è quello di definire il comportamento del sistema, inclusi i **requisiti funzionali**(Attività che il sistema deve supportare, Attori che interagiscono con il sistema, Relazioni tra attività e attori), per facilitare la comunicazione tra sviluppatori e stakeholder. **Gli elementi principali della documentazione** sono **:** attori (entità esterna che interagisce con il sistema per raggiungere un obiettivo, fornendo imput e ricevendo output dal sistema), relazioni ( semplice, inclusione, estensione,generalizzazione) ,casi d uso, scenari ( si suddividono in sequenza tipica e sequenza alternativa( descrive eventuali varaizioni o eccezioni). lo use case è utile utilizzarlo quando bisogna chiarire cosa deve fare il software, suddividendo le esecuzioni. Le limitazioni dello use case sono: Requisiti non funzionali (es. prestazioni, sicurezza), Strutture interne o architetture del sistema,Richiedono descrizioni testuali aggiuntive per chiarire molti dettagli.

**Cosa sono i Diagrammi di Interazione e a cosa servono?**

I Diagrammi di Interazione sono strumenti UML utilizzati per descrivere il **comportamento dinamico** di un gruppo di oggetti che collaborano per risolvere un problema o eseguire un’operazione. Servono a: Modellare le interazioni tra oggetti in termini di scambio di messaggi. Rappresentare la logica interna di un caso d’uso o di uno scenario specifico. i diagrammi di iterazione si suddividono in: **Sequence Diagram** ( evidenziano la sequenza temporale dei messaggi scambiati tra gli oggetti), **Communication Diagram** (detto anche Collaboration Diagram): Evidenziano le connessioni tra gli oggetti e l’ordine dei messaggi.

**Parla del Sequence**

è composto da : asse orizzontale( rappresenta gli oggetti) e asse verticale(rappresenta il tempo), lifeline(rappresenta una vita di un oggetto/attore), messaggi(sincroni, asincroni/ di risposta, autodiretti), condizioni(alt(scelte tra 2 o piu operazioni),opt(condizione eseguita o ignorata),loop(basate su una condizione booleane )

**Parla del Communication Diagram e definisci le differenze con sequence**

Si concentra sulle connessioni tra gli oggetti e sullo scambio di messaggi. I messaggi sono numerati per rappresentare l’ordine di esecuzione. Non c’è un asse temporale esplicito; l’ordine è determinato dalla numerazione.I **Sequence Diagram** enfatizzano la sequenza temporale e la vita degli oggetti. I **Communication Diagram** enfatizzano le relazioni strutturali e le connessioni tra oggetti.

**Come si modellano eventi persi (lost) o trovati (found) in UML?**

**Eventi persi (lost)**: Un messaggio che parte da un mittente noto ma non arriva al destinatario. **Eventi trovati (found)**: Un messaggio che arriva a un destinatario noto ma il mittente è sconosciuto.

**Cosa sono i Diagrammi di Classe in UML?**

I Diagrammi di Classe sono uno strumento UML che rappresenta la struttura statica di un sistema software, modellando:Classi: Componenti principali con attributi e metodi.Relazioni: Connessioni tra classi, come associazioni, aggregazioni, composizioni ed ereditarietà. I principali elementi di un class diagram sono: nome,attributi,metodi, visibilita(pubblico,privato,protetto, visibilita package), vincoli (su attributi, associazioni e relazioni, notazione {x>0}),molteplicità  associazione(definisce il numero di istanze che possono partecipare a un’associazione( esattamente 1, 0…\**,1….*\*,n…..m), relazioni(aggregazioni(Le parti possono esistere indipendentemente dal tutto.),composizione( Le parti non possono esistere senza il tutto.), relazione semplice, ereditarietà)

Scopo principale:

**Cos’è un’interfaccia in UML e come si rappresenta?**

Un’interfaccia è una classe astratta che specifica un contratto di comportamento per le classi che la implementano. Contiene solo metodi astratti e attributi costanti. Una classe può implementare più interfacce.

**Che cos'è un package in UML e quali sono le sue principali caratteristiche?**

Un package in UML è un "contenitore" che raggruppa elementi UML come classi, use case e altri oggetti. Le principali caratteristiche sono: **Definizione di namespace**: i nomi degli elementi all’interno devono essere univoci. **Visibilità**: la visibilità degli elementi può essere controllata per limitare l’accesso da altri package. le relazioni tra i package possono includere **Generalizzazione**: rappresenta relazioni di ereditarietà tra elementi di package diversi.**Dipendenza**: un package dipende da un altro quando necessita di accedere a elementi definiti nel secondo package. le relazioni tra i package possono essere: **Aggregazione**: un package può contenere altri package. **Merge**: simile alla generalizzazione, combina i contenuti di due package con nomi o concetti condivisi.**Import**: aggiunge gli elementi di un package target al namespace del package client.**Access**: permette di referenziare elementi in un package target senza modificarne il namespace.  È utilizzata per riassumere molteplici dipendenze a livello di elementi. Utilizzati per: **Gestire la complessità**: permettono di suddividere un sistema complesso in componenti più semplici.**Organizzazione gerarchica**: modellano le relazioni tra classi (ereditarietà, aggregazione, composizione) a livello più alto. **Alta coesione e interfacce precise**: favoriscono una progettazione modulare con chiari confini tra i moduli.

**Qual è la differenza tra una relazione di tipo <<import>> e <<access>>?**

**<<import>>**: gli elementi importati entrano nel namespace del package client. Se ci sono conflitti di nome, il modello risulta non valido. **<<access>>**: garantisce la possibilità di referenziare gli elementi di un package target senza introdurli nel namespace del client.

**Cos'è un package merge e in quali casi viene utilizzato?**

Un package merge è una relazione che combina i contenuti di due package. È utilizzato quando: Gli elementi di package diversi hanno lo stesso nome e rappresentano lo stesso concetto. Si vuole estendere incrementando il significato base di un concetto comune, senza modificare gli elementi di base.

**In che modo le dipendenze tra package influenzano l'accesso agli elementi?**

Le dipendenze regolano l’accesso agli elementi tra package. Regole principali: Un package può accedere solo agli elementi visibili (public o protected) del package target. Le dipendenze non sono transitive: se A dipende da B e B dipende da C, non significa che A possa accedere a C.

**Quali approcci possono essere adottati per l'uso dei package?**

**Top-down**: si parte identificando i componenti principali come package vuoti, che verranno poi riempiti con classi e altri elementi. **Bottom-up**: si raggruppano elementi già esistenti in package per semplificare modelli complessi e migliorare la leggibilità.

**Che cosa sono gli Activity Diagram?**

Gli Activity Diagram sono una rappresentazione grafica che descrive la sequenza di operazioni per eseguire attività complesse. Sono un caso particolare di diagrammi di macchine a stati, dove: Ogni stato rappresenta un’attività o un'azione. Le transizioni sono causate dal completamento delle azioni. hanno lo scopo di: Evidenziare processi guidati dall’elaborazione interna (workflow), anziché eventi esterni. I principali elementi grafici sono: nodi finali e iniziali,attività ( azioni), fork/join (gestiscono la concorrenza e sincoronizzazioni), branch/merge(definiscono la biforcazioni basate su condizioni), flussi(micro attivita( di solito dentro ad un rettangolo)), corsie(swimlanes)(ogni corsia rappresenta un attore o un ruolo suddividendo le responsabilità ), regione interrompibile, eccezioni

**Che cos'è un Timing Diagram ?**

Il Timing Diagram è un diagramma UML che rappresenta esplicitamente il tempo. È utilizzato per:  Mostrare come lo stato o il valore di un elemento cambia nel tempo.Rappresentare interazioni tra eventi temporizzati. Definire vincoli di tempo e durata tra eventi. Ad esempio, può essere usato per descrivere l'interazione tra un utente e un sistema automatizzato con vincoli temporali specifici. Composto da Lifeline( State Lifeline: rappresenta i cambiamenti di stato nel tempo/ Value Lifeline\*\*: rappresenta i cambiamenti di valore nel tempo), Transizioni: indicano eventi che cambiano lo stato o il valore.Vincoli temporali\*\*: definiscono i limiti entro cui devono avvenire determinati eventi o stati.

**Che cos'è un Component Diagram ?**

Un \*\*Component Diagram\*\* descrive i componenti modulari di un sistema e le loro relazioni. Ogni componente:\*\*Incapsula\*\* i suoi contenuti, rendendoli sostituibili. È definito da interfacce fornite e richieste, che stabiliscono le sue funzionalità e dipendenze. Viene usato per rappresentare un sistema a un livello di astrazione superiore rispetto al Class Diagram. Gli elementi principali sono: \*\*Componenti\*\*: rappresentano parti modulari del sistema. \*\*Interfacce fornite\*\*: funzioni o servizi offerti dal componente.  \*\*Interfacce richieste\*\*: funzionalità di cui il componente ha bisogno.  \*\*Connettori\*\*: collegano interfacce fornite e richieste. \*\*Artifatti\*\*: elementi concreti legati ai componenti, come file o librerie.

**Cosa rappresenta un Deployment Diagram?**

Un \*\*Deployment Diagram\*\* rappresenta l’architettura di esecuzione del sistema, mostrando come i componenti software (artifacts) sono distribuiti sui nodi hardware o software. Serve a: Definire i dettagli di distribuzione fisica. Modellare reti complesse e connessioni tra nodi. Gli elementi principali includono: \*\*Nodi\*\*: risorse computazionali, come server o PC.  \*\*Artifatti\*\*: elementi concreti come eseguibili, script, database. \*\*Canali di comunicazione\*\*: connessioni fisiche o virtuali tra nodi.  \*\*Associazioni\*\*: relazioni tra i nodi e gli artifatti che ospitano.Gli \*\*artifatti\*\* rappresentano entità fisiche prodotte durante il processo di sviluppo, come: File sorgenti.Script di configurazione. Database o documenti. Sono assegnati ai nodi e definiscono cosa viene installato o eseguito su ogni nodo. Le relazioni principali includono: \*\*Connessioni fisiche\*\*: rappresentano cavi o reti che collegano nodi hardware. \*\*Connessioni virtuali\*\*: come protocolli TCP/IP. Nodi annidati: un nodo può contenere altri nodi o componenti (es. un server che ospita una VM).

**Quali sono le differenze tra un diagramma dinamico e un class diagram?**

**Class Diagram**: Rappresenta la struttura statica del sistema, mostrando classi, attributi, metodi e relazioni. È utile per modellare il dominio del problema. **Diagrammi Dinamici**: Mostrano il comportamento del sistema nel tempo, includendo gli stati delle classi (state diagram) o le interazioni tra oggetti (sequence diagram). Sono utili per comprendere il flusso degli eventi e il cambiamento degli stati

**Si illustrino le motivazioni per l’introduzione e l’uso dei design pattern e se ne descriva uno in dettaglio:**

Un design pattern fornisce al progettista software: 1. una soluzione consolidata, già testata, per un problema ricorrente; 2. un livello di astrazione solitamente più elevato di una classe; 3. un modo per progettare software con caratteristiche già note. 4. un supporto alla progettazione di sistemi complessi. MVC → un pattern architetturale che struttura l’applicazione mediante tre tipi principali di componenti: **Model**: rappresenta i dati veri e propri, funzionali all’applicazione, in modo indipendente da come vengono visualizzati sullo schermo e dalle modalit di input da parte dell’utente. **View**: l’interfaccia utente (o con altri sistemi), che visualizza sullo schermo i dati rappresentati dal model. **Controller**: gestisce la comunicazione tra view e model, ricevendo gli input dall’utente e traducendoli in richieste di servizio per il modello o la vista a cui è associato. In questo modo, si realizza un sistema a strati, ciascuno dei quali comunica solo con quelli adiacenti. Grazie al disaccoppiamento di viste e modelli, la presentazione dei dati all’utente può essere modificata indipendentemente dalla logica applicativa. È possibile (e comune) avere più rappresentazioni grafiche per gli stessi dati, mediante view e controller diversi che fanno riferimento allo stesso modello

**Che cos'è un design pattern ?**

Un design pattern è una soluzione riutilizzabile e collaudata per un problema ricorrente nel contesto della progettazione software, utile per oggetti condivisi come database o log. I vantaggi principali includono:**Riutilizzo**: Consente di applicare soluzioni standardizzate a problemi comuni.**Abstratione di alto livello**: Fornisce modelli progettuali più astratti rispetto a singole classi. **Gestione della complessità:** Aiuta a progettare sistemi complessi rendendoli più comprensibili e modificabili. **Comunicazione tra progettisti:**Fornisce un vocabolario comune per discutere di soluzioni.  i principali svantaggi sono : puo rendere la struttura del codice piu complessaLe categorie principali  sono:**Creazionali:** Gestiscono la creazione degli oggetti, spesso separandola dalla logica del programma (es. Factory, Singleton). **Strutturali**: Si concentrano sulle composizioni e le relazioni tra oggetti o classi (es. Composite, Adapter).**Comportamentali**: Gestiscono interazioni e comunicazioni tra oggetti (es. Strategy, Observer).

**Spiega il pattern Factory Method e fornisci un esempio del suo utilizzo ?**

Il pattern \*\*Factory Method\*\* definisce un'interfaccia per creare oggetti in una classe, ma lascia alle sottoclassi il compito di specificare il tipo di oggetti da creare.quindi Il codice di un programma ad oggetti per lo più non dipende dalla precisa classe cui appartiene un certo oggetto  \*\*Esempio\*\*: Un'app di logistica che supporta diversi tipi di trasporti (es. Camion, Nave). Il metodo factory crea dinamicamente l'oggetto corrispondente al tipo di trasporto richiesto senza conoscere i dettagli specifici della classe concreta.

**Descrivi il pattern Singleton e fornisci un esempio di utilizzo**

Il pattern Singleton assicura che una classe abbia solo un’istanza e fornisce un punto di accesso globale a tale istanza. Questo modello è spesso utilizzato per gestire risorse condivise come connessioni al database o logger, dove è importante garantire che esista un solo oggetto di tale risorsa. In una classe **Singleton il costruttore è protetto o privato** e un metodo statico, o una factory, forniscono l’accesso alla sola copia dell’oggetto **Esempio**: Implementare un gestore di connessioni al database in un’applicazione può essere fatto con il Singleton per assicurarsi che tutte le connessioni utilizzino lo stesso oggetto di connessione. Il **Singleton** è **utile** quando è necessario garantire che una classe abbia una sola istanza globale, ad esempio per una configurazione di sistema o un logger.**\*\*Rischi\*\***: Può violare il principio di singola responsabilità. Difficoltà nei test unitari a causa della sua natura globale. Rischio di creare dipendenze non necessarie tra le componenti.

**Si illustri il design pattern Flyweight:**

Quando molti oggetti identici e immutabili vengono utilizzati contemporaneamente, è utile costruire e condividere un solo oggetto per ogni classe di equivalenza. Questi oggetti condivisi vengono chiamati flyweight (pesi mosca), perché spesso sono molto piccoli. Un esempio comune è il tipo String, che in molti linguaggi (compreso Java) è gestito appunto come un flyweight. Flyweight è un pattern strutturale che permette tenere in memoria RAM un numero maggiore di oggetti che condividono parti comuni. Viene dunque usato per affrontare problem legati allo spreco di risorse. Per implementare questo pattern, si usa una factory che tiene gli oggetti già creati in una tabella. Quando viene richiesto un oggetto, la factory controlla se questo esiste già nella tabella, e in tal caso restituisce un riferimento a esso, altrimenti costruisce un nuovo oggetto (chiamando un costruttore privato) e lo aggiunge alla tabella. Gli oggetti flyweight devono essere immutabili per evitare problemi di aliasing: se uno di essi venisse modificato, tale modifica sarebbe visibile da tutti i client che lo condividono (cosa che invece non accadrebbe senza l’uso dei flyweight, perché allora i client avrebbero copie separate dell’oggetto). Usando questo pattern, se c’ un alto grado di condivisione degli oggetti: • si risparmia memoria; • non si perde tempo a inizializzare oggetti duplicati; • si pu usare ‘==’ per il confronto, al posto di equals

**Parla del Pattern State:**

Il design pattern **State** gestisce la variabilità del comportamento di un oggetto che cambia configurazione in base al proprio stato, evitando codice complesso e pieno di condizioni. Introduce un'interfaccia o una classe astratta per rappresentare i comportamenti variabili, con ogni stato implementato come una classe separata. La **classe principale** si occupa di: Scegliere lo stato appropriato e Delegare il comportamento alla classe corrispondente. Questo approccio migliora la modularità, elimina strutture condizionali come if o switch, e permette all'utente di usare l'oggetto senza percepire i cambiamenti di stato. **Componenti principali:** Interfaccia o classe astratta per i comportamenti comuni. Classi concrete per i comportamenti specifici degli stati. Classe principale per la gestione degli stati. Esempio: in un sistema di gestione degli accessi, il comportamento dei metodi cambia a seconda che l'utente sia autenticato o meno. Tuttavia, il pattern può aumentare il numero di classi e non è sempre adatto per oggetti con stati semplici.

**In quali casi è utile il pattern Strategy?\*\***

 Il \*\*Strategy\*\* è utile quando è necessario definire una famiglia di algoritmi intercambiabili e rendere il loro utilizzo flessibile.

- \*\*Esempio\*\*: Un'app di navigazione che supporta percorsi a piedi, in auto o con trasporto pubblico. Ogni modalità di calcolo del percorso è implementata come una strategia separata.

**Quali problemi risolve il pattern Adaptor e come funziona?\*\***

 L'\*\*Adapter\*\* permette a classi con interfacce incompatibili di collaborare.L’oggetto esterno è l’Adapter, quello interno l’Adaptee. le librerie di classi per l’interfaccia grafica, come AWT o Swing, non sono altro che enormi raccolte di oggetti Adapter  \*\*Problemi risolti\*\*: Integrare librerie di terze parti con interfacce non compatibili. Uniformare l'accesso a classi eterogenee.

**Quando è consigliabile utilizzare il pattern Proxy?\*\***

Il \*\*Proxy\*\* è utile per controllare l'accesso a un oggetto o ritardarne la creazione fino a quando non è necessario. crea un  intermediario, che effettua pre- e post-

processing (controllo di accessi, caching, …).Quando l’oggetto interposto espone esattamente la stessa interfaccia dell’oggetto separato, di cui fa le veci, esso prende il nome di Proxy. alcuni esempi di proxy sono:  
**Remote Proxy** fornisce un’interfaccia locale che maschera la comunicazione con un oggetto localizzato in un altro spazio di indirizzamento **Virtual Proxy** crea oggetti “costosi” (che occupano per esempio molta

memoria) solo quando è necessario. **Protection Proxy** gestisce livelli di accesso diversi ad un oggetto da parte di client diversi **Smart Reference** viene usato in sostituzione ad un semplice riferimento. Effettua azioni aggiuntive quali, per esempio, il lock/unlock dei dati che vengono acceduti **OMG-CORBA, Java RMI** meccanismo di comunicazione remota fra oggetti basata su “proxy”  per ogni oggetto remoto esiste un proxy locale

**HTTP Proxy**:  intermediario fra web browser e http server, caching di documenti a cui accedono frequentemente diversi browser Firewall di rete controllo degli accessi

**Parla del Design Pattern decorator**

Il design pattern **Decorator** permette di aggiungere dinamicamente funzionalità a un oggetto senza modificarne il codice. Utilizza una struttura a composizione, in cui i decoratori avvolgono l'oggetto originale, mantenendo l'interfaccia comune e consentendo un uso trasparente.**Componenti chiave sono :**Interfaccia o classe astratta condivisa; L'oggetto di base da decorare e il Decoratore**:** Classe che aggiunge funzionalità mantenendo un riferimento all'oggetto decorato

**Parla dell'Abstract factory**

L’**Abstract Factory** è un design pattern creazionale che consente di creare **famiglie di oggetti correlati o dipendenti** senza specificare le loro classi concrete. Questo pattern fornisce un’interfaccia per creare oggetti, lasciando alle classi concrete il compito di definire quale oggetto specifico creare. **Quando usarlo:** Quando il sistema deve essere indipendente da come i suoi oggetti sono creati, compositi o rappresentati. Quando si desidera creare oggetti correlati o interdipendenti, garantendo che appartengano alla stessa famiglia.Quando il sistema deve essere configurabile con diverse implementazioni di oggetti.

**Cos'è il pattern Composite e quali problemi risolve?**

Il pattern \*\*Composite\*\* permette di comporre oggetti in strutture ad albero per rappresentare gerarchie intero-parte. Consentire ai client di trattare oggetti singoli e compositi in modo uniforme Minimizzare il più possibile la complessità di una gerarchia intero-parte  riducendo il numero di tipologie di oggetti che possono trovarsi nei diversi nodi dell’albero

Problemi risolti\*\*:Trattare oggetti primitivi e compositi in modo uniforme. Ridurre la complessità delle gerarchie e semplificare l'aggiunta di nuovi tipi di componenti.

**Cos'è il pattern facade e quali problemi risolve?**

Il **Facade Pattern** è un design pattern strutturale che fornisce un’interfaccia semplificata e unificata per un sistema complesso di classi o sotto-sistemi. L’obiettivo del pattern è rendere più semplice e chiaro l’accesso a funzionalità complesse, nascondendo i dettagli di implementazione e riducendo il numero di interazioni dirette con le classi interne.

**Descrivi il pattern Observer e un suo possibile utilizzo pratico.**

 Il \*\*Observer\*\* stabilisce una relazione uno-a-molti tra oggetti, in modo che quando uno cambia stato, tutti gli osservatori registrati vengano notificati. \*\*Esempio pratico\*\*: Un'app di stock market che notifica i clienti su variazioni di prezzo di un'azione. Ogni cliente si registra come osservatore dell'azione desiderata e riceve notifiche automatiche.

**Parla dei Pattern Architetturali:**

I **pattern architetturali** sono una tipologia di pattern utilizzati per organizzare la struttura di un sistema software, suddividendolo in sottosistemi con responsabilità definite e relazioni ben gestite. Sono utili per affrontare la complessità su larga scala e differiscono da altre tipologie di pattern: **Design Pattern:** Operano a livello di sottosistema, evidenziando le associazioni tra classi. **Idioms:** Operano a basso livello, specifici di un linguaggio di programmazione. Esempi di pattern architetturali: **Layers:** Divide il sistema in livelli con ruoli distinti. **Pipes & Filters:** Gestisce i dati tramite trasformazioni indipendenti. **Broker:** Media tra componenti distribuiti. **Model-View-Controller (MVC):** Separa il sistema in modello, vista e controller. Questi pattern garantiscono modularità, scalabilità e manutenibilità, rappresentando una guida per la progettazione dell’intero sistema

***Spiega il pattern MVC (Model-View-Controller)(pattern mining) e la sua relazione con altri pattern.***

MVC → un pattern architetturale che struttura l’applicazione mediante tre tipi principali di componenti: **Model**: rappresenta i dati veri e propri, funzionali all’applicazione, in modo indipendente da come vengono visualizzati sullo schermo e dalle modalit di input da parte dell’utente. **View**: l’interfaccia utente (o con altri sistemi), che visualizza sullo schermo i dati rappresentati dal model. **Controller**: gestisce la comunicazione tra view e model, ricevendo gli input dall’utente e traducendoli in richieste di servizio per il modello o la vista a cui è associato. In questo modo, si realizza un sistema a strati, ciascuno dei quali comunica solo con quelli adiacenti. Grazie al disaccoppiamento di viste e modelli, la presentazione dei dati all’utente può essere modificata indipendentemente dalla logica applicativa. È possibile (e comune) avere più rappresentazioni grafiche per gli stessi dati, mediante view e controller diversi che fanno riferimento allo stesso modello