1) Descrivere i concetti di accoppiamento e coesione.
Accoppiamento e coesione sono due proprietà dei sottosistemi. L'accopiamento misura la dipendenza tra due sottosistemi e la coesione misura la dipendenza fra le classi di un sottosistema.
L'accoppiamento è il numero di dipendenze tra due sottosistemi. Se due sistemi sono scarsamente accoppiati sono relativamente indipendenti (modifiche su uno dei sottosistemi avranno probabilmente un impatto limitato sull'altro). Se due sistemi sono fortemente accoppiati, sono reciprocamente dipendenti (una modifica su di un sottosistema
può avere notevoli impatti sull'altro). Alta coesione : le classi di un sottosistema realizzano compiti simili e sono collegate le une alle altre(attraverso associazioni). Bassa coesione : il sottosistema contiene certo numero di oggetti non correlati.
La coesione misura le dipendenze all'interno di un singolo sottosistema.
2) Descrivere le attività di System Design
Le attività del System Design consistono nell'individuare degli obiettivi di design(design goals) che verranno utilizzati per ottimizzare la scomposizione del sistema in funzione di questi, inoltre vengono attuate importanti scelte di design come la scelta dell'architettura hardware/software da utilizzare e i supporti per gestire i dati persistenti
Scomposizione in sottosistemi: Il sistema viene decomposto seguendo i design goals massimizzando la coesione tra le classi e diminuendo l'accoppiamento tra i sottosistemi(in casi di modifica si impatta meno se ci sono poche dipendenze)
Concorrenza: Se nel sistema esistono dei thread e come vengono gestiti
Mapping Hardware/Software: Viene deciso come mappare il software su componenti hardware reali
Gestione dei dati persistenti: Si sceglie se utilizzare un database relazionale o un file system a seconda dei requisiti.
Controllo degli accessi e sicurezza: viene deciso come garantire la protezione sui dati e l'accesso a questi tramite una matrice degli accessi
Controllo globale del software: In base all'analisi dei sequence diagram si stabilisce se il controllo è di tipo centralizzato(se i sequence assomigliano ad un fork diagram) o decentralizzato(se i sequence sono stair). Il primo caso può essere procedure-driven, se il controllo è gestito dal programma o event-driven, se è gestito dall'utente.
Boundary Condition: Sono dei casi d'uso eccezionali che vengono conseguentemente all'identificazione di nuovi sottosistemi(ad esempio l'avvio di un server)
3) Descrivere i concetti di stratificazione e partizionamento.
Un sistema grande è di solito decomposto in sottosistemi usando layer e partizioni. Una decomposizione gerarchica di un sistema consiste di un insieme ordinato di layer. Un layer è un raggruppamento di sottoinsiemi che forniscono servizi correlati, eventualmente realizzati utilizzando servizi di altri layer. I sistemi stratificati sono gerarchici. La gerarchia riduce la complessità. Le architetture chiuse sono più portabili. Le architetture aperte sono più efficienti. Se un sottosistema è un layer, spesso è anche chiamato macchina virtuale.
Il partizionamento è un altro approccio per gestire la complessità. Il sistema viene partizionato in sottosistemi paritari, ciascuno dei quali è responsabile di differenti classi di servizi.
4) Descrivere la differenza tra architetture aperte e architetture chiuse.
Le architetture chiuse hanno il vantaggio di una maggiore manutenibilità e portabilita' in quanto per accedere a dei livelli inferiori si deve attraversare necessariamente un sottosistema intermedio.
*In architetture aperte al contario ogni layer può accedere a tutti i layer sottostanti, rendendo il sistema più efficiente in quanto si risparmia l'overhead delle chiamate in cascata.*Se un sottosistema è un layer, spesso è chiamato macchina virtuale.

5) Descrivere lo stile architetturale Repository.
I sottosistemi accedono e modificano una singola struttura dati chiamata repository. I sottosistemi sono loosely coupled (interagiscono solo attraverso il repository). Il flusso di controllo è determinato o dal repository o dai sottosistemi. I repository sono adatti per applicazioni con task di elaborazione dati complessi e soggetti a frequenti cambiamenti. Una volta che un repository centrale è stato definito, nuovi servizi nella forma di sottosistemi addizionali possono essere definiti facilmente. Problema : il repository centrale può facilmente diventare un collo di bottiglia per le prestazioni e la modificabilità; l'accoppiamento tra i sottosistemi ed il repository è elevato, ciò rende difficile modificare il repository senza impattare su tutti gli altri sottosistemi.
6) Descrivere lo stile architetturale Model/View/Controller.
In questo stile architetturale i sottosistemi sono classificati in 3 tipi differenti:

• Sottosistema Model (modello) mantiene la conoscenza del dominio di applicazione;

• Sottosistema View (vista) visualizza all'utente gli oggetti del dominio dell'applicazione;

• Sottosistema Controller (controllore) è responsabile della sequenza di interazioni con l'utente.

MVC è un caso particolare di architettura di tipo repository.
MVC è appropriato per i sistemi interattivi, specialmente quando si utilizzano viste multiple dello stesso modello.
Introduce lo stesso collo di bottiglia visto per lo stile architetturale Repository.
7) Descrivere lo stile architetturale Client/Server.
Un sottosistema, detto Server, fornisce servizi ad istanze di altri sottosistemi detti Client. I Client chiamano il Server che fornisce i servizi richiesti:

• I Client conoscono l'interfaccia del Server (servizi);

• Il Server non conosce le interfacce dei Client.

La risposta è immediata. Gli utenti interagiscono solo con il Client. È un caso speciale dell'architettura Repository, in cui la struttura dati centrale è gestita da un processo.
È usata spesso nei sistemi informativi con un Database centralizzato. Problemi : i sistemi stratificati non forniscono comunicazioni di tipo Peer – to – Peer (P2P). Spesso le comunicazioni di tipo P2P sono indispensabili.
8) Descrivere lo stile architetturale Peer – to – Peer.
È una generalizzazione dell'Architettura Client/Server. Ogni sottosistema può agire sia come client sia come server : può richiedere e fornire servizi. Il flusso di controllo di ciascun sottosistema è indipendente dagli altri, eccetto che per la sincronizzazione sulle richieste.
9) Descrivere gli stili architetturali a tre e quattro livelli.
Architettura Three – tier (tre livelli): i sottosistemi sono organizzati in tre strati :

• L'Interface Layer, include tutti i boundary object che gestiscono l'interazione con l'utente;

• L'Application Logic Layer, include tutti gli oggetti relativi al controllo e alle entità;

• Lo Storage Layer (o Data Layer) effettua la memorizzazione, il recupero e l'interrogazione degli oggetti persistenti.

La separazione dell'interfaccia dalla logica applicativa consente di modificare e/o sviluppare diverse interfacce utente per la stessa logica applicativa.
Stili architetturali a quattro livelli:

vengono implementati in sistemi client server, e presentano uno strato extra di presentation layer se il client è thin, oppure uno strato extra di data layer se abbiamo un fat client. Es:web browser, forum, connection, query.

10)Descrivere i design goals relativi alle Performance.

Includono i requisiti imposti sul sistema in termini di spazio e velocità:

• Tempo di risposta : il tempo entro quanto una richiesta da parte di un utente deve essere soddisfatta.

• Troughput : il numero di task che il sistema porta a compimento in un periodo di tempo prefissato.

• Memoria : lo spazio richiesto affinchè il sistema possa funzionare.

11) Descrivere i design goals relativi ai costi.

Vanno valutati i costi di sviluppo del sistema, alla sua installazione ed al training degli utenti, eventuali costi per convertire i dati di un sistema precedente, costi di manutenzione e di amministrazione.

12)Descrivere i design goals relativi di Dependability.

Quanto sforzo deve essere speso per minimizzare i crash del sistema e le loro conseguenze?

Rispondono alle seguenti domande :

• Robustness (robustezza) : Capacità di resistere ad input non validi immessi dall'utente;

• Reliability (affidabilità) : Differenza tra comportamento specificato e osservato;

• Availability (disponibilità) : Percentuale di tempo in cui il sistema può essere utilizzato per compiere normali attività;

• Fault Tolerance (tolleranza ai guasti) : Capacità di operare sotto condizioni di errore;

• Security (sicurezza del sistema) : Capacità di resistere ad attacchi di hacker;

• Safety (sicurezza nell'utilizzo del sistema) : Capacità di evitare di danneggiare vite umane,
anche in presenza di errori e di fallimenti.

13)Descrivere i design goals relativi di Maintenance.

Determinano quanto deve essere facile modificare il sistema dopo il suo rilascio:

• Estensibilità : Facilità di aggiungere funzionalità o nuove classi al sistema;

• Modificabilità : Facilità di modificare le funzionalità del sistema;

• Adattabilità : Facilità di portare il sistema in un differente dominio di applicazione;

• Portabilità : Facilità di portare il sistema su una differente piattaforma;

• Leggibilità : Facilità di comprendere il sistema dalla lettura del codice;

• Tracciabilità dei requisiti : Facilità di mapping del codice sui relativi requisiti.

14)Descrivere le diverse strategie per la gestione della memorizzazione.

La decisione dipende molto spesso dai vincoli non funzionali.

• File: Questa strategia da una parte richiede una logica più complessa per la lettura/scrittura, dall'altra permette un accesso ai dati più efficiente.

• DBMS Relazionale: fornisce un interfaccia di più alto livello rispetto ai file. I dati vengono memorizzati in tabelle ed è possibile utilizzare un linguaggio standard per le operazioni. Gli oggetti devono essere mappati sulle tabelle per poter essere memorizzati, e viene occupato circa tre volte lo spazio che sarebbe stato utilizzato con i file.

• DBMS ad Oggetti: simile al relazionale con la differenza che non è necessario mappare gli oggetti in tabelle in quanto questi vengono memorizzati così come sono. Con questa strategia si risparmia sulle decisioni di mapping ma è lento e le query sono di difficile comprensione.

15)Descrivere vantaggi e svantaggi nella scelta tra Flat file, Database Relazionali ed Object – Oriented.

I flat file si scelgono quando : si hanno a disposizione dati voluminosi e non strutturati, dati temporanei, bassa densità di informazioni.
Scegliere un database relazionale o orientato agli oggetti quando : si hanno accessi multipli
concorrenti, piattaforme o applicazioni multiple per gli stassi dati.
Scegliere un database relazionale quando : si hanno query complesse sugli attributi, grandi
insiemi di dati.
Scegliere un database Object – Oriented quando : si ha un intenso uso di associazioni per recuperare i dati, insiemi di dati di taglia media, associazioni fra oggetti irregolari.

16)Descrivere i diversi approcci per la specifica del controllo degli accessi agli oggetti di
un sistema software.

• La tabella globale degli accessi: rappresenta esplicitamente ogni cella nella matrice come
una tripla (attore, classe, operazione) . Se tale tripla non è presente l'accesso è negato.

• Lista di controllo degli accessi: associa una coppia (attore, operazione) a ciascuna classe.

• Capability: associa una coppia (classe, operazione) a ciascun attore.

17)Descrivere i tre meccanismi principali per il flusso di controllo.

• Controllo guidato dalla procedura (Procedure driven control) : le operazioni aspettano gli input ogni volta che hanno bisogno di dati da parte di un attore. I controlli risiedono nel codice del programma. È principalmente utilizzato nei sistemi legacy, e nei sistemi scritti con linguaggi procedurali. È difficile da usare nei linguaggi Object – Oriented.

• Controllo guidato dagli eventi (Event driven – control) : un loop principale attende un evento esterno; quando l'evento si verifica, è spedito all'oggetto appropriato. Il controllo risiede in un dispatcher che chiama le funzioni del sottosistema. È flessibile e particolarmente adatto per gestire le interfacce utenti grafiche.

• Controllo basato su Thread : rappresentano la versione concorrente del controllo Procedure
– driven. Il sistema può creare un numero arbitrario di threads, ciascuno dei quali risponde ad un differente evento. Se un thread necessita di informazioni addizionali aspetta un input da uno specifico attore. Il debug del software che utilizza questo tipo di flusso di controllo è complesso.

18) Spiegare la differenza tra Ereditarietà di Specifica ed Ereditarietà di Implementazione
e fornire un esempio per ciascun tipo di relazione.

L'ereditarietà di specifica (chiamata anche Interface Inheritance) eredita da una classe astratta i metodi non implementati(implementa una interfaccia) mentre l'ereditarietà di implementazione eredita tutti i metodi dalla superclasse, ma è sconsigliato usarla, si usa la delegazione, altrimenti qualunque utilizzatore della classe potrebbe chiamare un metodo della superclasse.

19) Spiegare l'importanza delle interfacce. Per quale motivo si dovrebbe programmare
sempre riferendosi ad un'interfaccia, piuttosto che ad un'implementazione?
Ci sono due vantaggi fondamentali nel gestire gli oggetti facendo riferimento esclusivamente alle interfacce di classe astratte:

• i client non sono a conoscenza dei tipi specifici degli oggetti che usano;

• i client non sono a conoscenza della classe che effettivamente implementa questi oggetti. I
client conoscono infatti solo la classe astratta che definisce l'interfaccia.
Non si dovrebbero dichiarare variabili che sono istanze di classi concrete specifiche, ma ci si
dovrebbe affidare a interfacce definite da classi astratte.

20) Descrivere il concetto di delega e fornire un esempio di applicazione di tale relazione.
La delega è un'alternativa all'ereditarietà che dovrebbe essere utilizzata quando l'unico obiettivo è il riuso di componenti. La delega rende esplicite le dipendenze tra la classe riutilizzata e la nuova classe.
Un esempio potrebbe essere: immaginiamo una classe che estende un altra, questa è un implementazione con ereditarietà, mentre se la stessa classe invece di estendere un altra classe la utilizza nel costruttore avremo un implementazione con delega.

21)Descrivere il Bridge Design Pattern e fornire un problema di progettazione in cui potrebbe essere adottato.
È un pattern strutturale, ovvero si occupa delle modalità di composizione di classi e oggetti per formare strutture complesse.
Problema : sviluppare, testare e integrare sottosistemi realizzati da differenti sviluppatori in maniera incrementale. Per risolvere questo problema utilizziamo il design pattern bridge.
Nome : Bridge
Descrizione del problema : separare un'astrazione da un'implementazione così che una diversa implementazione possa essere sostituita eventualmente a runtime
Soluzione : una classe Abstraction definisce l'interfaccia visibile al codice client. Implementor è una classe astratta che definisce i metodi di basso livello disponibili ad Abstraction. Un'istanza di Abstraction mantiene un riferimento alla sua istanza corrispondente di Implementor. Abstraction ed Implementor possono essere raffinate indipendentemente.
Conseguenze : disaccoppiamento tra interfaccia ed implementazione (un'implementazione non è più legata in modo permanente ad un'implementazione. L'implementazione di un'astrazione può essere configurata durante l'esecuzione. La parte di alto livello di un sistema dovrà conoscere soltanto le classi Abstraction ed Implementor); maggiore estendibilità (le gerarchie Abstraction ed Implementor possono essere estese indipendentemente); mascheramento dei dettagli dell'implementazione ai client (i client non
devono preoccuparsi dei dettagli implementativi).

22)Descrivere l'Adapter Design Pattern e fornire un problema di progettazione in cui potrebbe essere adottato.
È un pattern strutturale.
Nome : Adapter
Descrizione del problema : convertire l'interfaccia utente di una classe legacy in un'interfaccia diversa che il cliente si aspetta, in maniera tale che classi diverse possano operare insieme nonostante abbiano interfacce incompatibili.
Soluzione : ogni metodo dell'interfaccia verso il client è implementato in termini di richieste

alla classe legacy. Ogni conversione tra strutture dati o variazioni nel comportamento sono realizzate dalla classe Adapter. Viene usata per fornire una nuova interfaccia a componenti
legacy esistenti.
Conseguenze : se il Client utilizza ClientInterface allora può utilizzare qualsiasi istanza dell'Adapter in maniera trasparente senza dover essere modificato. L'Adapter lavore con la LegacyClass e con tutte le sue sottoclassi. L'adapter pattern viene utilizzato quando l'interfaccia e la sua implementazione esistono già e non possono essere modificate.

23)Descrivere l'Abstract Design Pattern e fornire un problema di progettazione in cui
potrebbe essere adottato.

È un pattern creazionale ovvero fornisce un'astrazione del processo di istanziazione degli oggetti e aiuta a rendere un sistema indipendente dalle modalità di creazione, composizione
e rappresentazione degli oggetti utilizzati
Problema : Consideriamo un'applicazione per un'abitazione intelligente: l'applicazione riceve eventi da sensori ed attiva comandi per dispositivi. L'interoperabilità in questo dominio è debole, di conseguenza è difficile sviluppare una singola soluzione SW per tutte le aziende.
Nome: Abstract Factory (Kit).
Descrizione del problema: Fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le loro classi concrete.
Soluzione: Una piattaforma (es., un sistema per la gestione di finestre) è rappresentato con
un insieme di AbstractProducts, ciascuno dei quali rappresenta un concetto (es., un bottone).
Una classe AbstractFactory dichiara le operazioni per creare ogni singolo prodotto. Una piattaforma specifica è poi realizzata da un ConcreteFactory ed un insieme di ConcreteProducts.
Conseguenze : Isola le classi concrete (Poiché Abstract Factory incapsula la responsabilità e
il processo di creazione di oggetti prodotto, rende i client indipendenti dalle classi effettivamente utilizzate per l'implementazione degli oggetti). E' possibile sostituire facilmente famiglie di prodotti a runtime (una factory concreta compare solo quando deve essere istanziata). Promuove la coerenza nell'utilizzo dei prodotti. Aggiungere nuovi prodotti è difficile poiché nuove realizzazioni devono essere create per ogni factory.

24)Descrivere lo Stategy Design Pattern e fornire un problema di progettazione in cui
potrebbe essere adottato.

È un pattern comportamentale ovvero si occupa di algoritmi e dell'assegnamento di responsabilità tra oggetti collaboranti.
Problema : Consideriamo un'applicazione mobile che si deve connettere a diversi tipi di rete, facendo lo switch in base alla posizione ed al costo della rete disponibile. Si vuole che
l'applicazione possa essere adattata a futuri protocolli di rete senza dover ricompilare l'applicazione.
Nome: Strategy (Policy)
Descrizione del problema: Definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Permette agli algoritmi di variare indipendentemente dai client che ne fanno
uso.

Soluzione: Un Client accede ai servizi forniti da un Context. I servizi del Context sono realizzati utilizzando uno dei diversi meccanismi, come deciso da un oggetto Policy. La classe astratta Strategy descrive l'interfaccia che è comune a tutti i meccanismi che Context
può usare. La classe Policy configura Context per usare un oggetto ConcreteStrategy.
Conseguenze : Un'alternativa all'ereditarietà (L'ereditarietà legherebbe staticamente il comportamento nel Context e mescolerebbe l'implementazione del Context con l'implementazione dell'algoritmo. Inoltre sarebbe impossibile modificare l'algoritmo dinamicamente). Policy decide quale Strategy è la migliore, in base alle circostanze correnti
Scelta dell'implementazione (Attraverso Strategy è possibile fornire diverse implementazioni dello stesso comportamento). Le strategie eliminano i blocchi condizionali

25)Descrivere il Proxy Design Pattern e fornire un problema di progettazione in cui potrebbe essere adottato.

26)Descrivere i concetti di Design Pattern, Framework e librerie di classi, in riferimento
all'attività di riuso.

I design pattern sono dei template di soluzioni a problemi comuni, raffinate nel tempo dagli sviluppatori.
Un framework è una struttura di supporto su cui un software può essere organizzato e progettato. Lo scopo di un framework è di risparmiare allo sviluppatore la riscrittura di codice già steso in precedenza per compiti simili.
Le librerie di classe sono il più generico possibile e non si focalizzano su un particolare dominio di applicazione fornendo solo un riuso limitato.

27)Descrivere i diversi tipi di contratto che possono essere specificati su una classe.

Spesso la specifica del tipo non è sufficiente a specificare il range dei valori consentiti per un attributo.
Si possono aggiungere contratti su una classe (consentono a class users, implementors ed
extenders di condividere le stesse assunzioni sulla classe).
I contratti possono essere usati per specificare senza ambiguità casi speciali o eccezionali
I contratti includono 3 tipi di vincoli:

• Invariante : Un predicato che è sempre vero per tutte le istanze di una classe. Gli invarianti
sono vincoli associati a classi o interfacce.

• Precondizione : Le precondizioni sono predicati associati ad una specifica operazione che
devono essere veri prima che l'operazione sia invocata. Le precondizioni sono usate per specificare vincoli che un chiamante deve soddisfare prima di chiamare un'operazione.

• Postcondizione : Le postcondizioni sono predicati associati ad una specifica operazione che
devono essere veri dopo che l'operazione è stata invocata. Le postcondizioni sono usate per
specificare vincoli che l'oggetto deve assicurare dopo l'invocazione dell'operazione.

28)Descrivere i diversi tipi di trasformazione tra i modelli (modello a oggetti e codice sorgente).

Una trasformazione ha lo scopo di migliorare un aspetto del modello (ad es. la sua modularità) mentre preserva tutte le altre proprietà (ad es. le funzionalità). Generalmente una trasformazione è localizzata, impatta su un numero ristretto di attributi, classi e

operazioni e viene eseguita in una serie di semplici passi.
Le trasformazioni del modello operano sul modello a oggetti (Es.: conversione di un attributo semplice: un indirizzo rappresentato da una stringa potrebbe essere trasformato in
una classe contenente via, codice postale, comune, provincia e nazione)

• Refactoring: operano sul codice sorgente, migliorando la sua leggibilità o la sua modificabilità, senza cambiare il comportamento del sistema.

• Forward Engineering. Produce un template del codice sorgente corrispondente al modello ad
oggetti. Molti costrutti (associazioni, attributi,...) possono essere mappati meccanicamente nei costrutti del codice sorgente (es. classi e dichiarazioni di campi in java), mentre il corpo dei metodi ed altri metodi privati vanno aggiunti dagli sviluppatori.

• Reverse Engineering. Produce un modello che corrisponde al codice sorgente (Nonostante il
supporto dei CASE, è necessario un forte intervento umano).

**29)Descrivere in che modo è possibile mappare un'associazione uno – a – molti tra due
oggetti e come memorizzare tale relazione in un database relazionale (fornire un esempio).**

Le associazioni uno – a – molti sono implementate usando una chiave esterna sul lato molti.
Le associazioni uno – a – molti possono essere realizzate con una tabella di associazione invece che con chiavi esterne. Usare le tabelle separate rende lo schema più modificabile. Se
cambia la molteplicità dell'associazione non dobbiamo cambiare lo schema.
Es : supponiamo che ad un cliente possano corrispondere più conti. Poiché i conti non hanno
un ordine specifico possiamo usare un insieme di riferimenti per modellare la parte “molti” dell'associazione.

**30)Descrivere in che modo è possibile mappare un'associazione molti – a – molti tra due
oggetti e come memorizzare tale relazione in un database relazionale (fornire un esempio).**

Entrambe le classi hanno campi che sono collezioni di riferimenti ed operazioni per mantenere queste collezioni consistenti. Le associazioni molti – a – molti sono implementate
usando una tabella separata contenente due colonne con chiave esterna per entrambe le classi dell'associazione.

**31)Descrivere in che modo è possibile mappare una relazione di ereditarietà su di uno
schema relazionale (mapping verticale e mapping orizzontale).**

I database relazionali non supportano l'ereditarietà. Esistono due opzioni per mappare l'ereditarietà in uno schema di un database :

• Mapping verticale : la superclasse e la sottoclasse sono mappate in tabelle distinte. Quella
della superclasse mantiene una colonna per ogni attributo definito nella superclasse e una colonna che indica quale tipo di sottoclasse rappresenta quell'istanza. La tabella della sottoclasse include solo gli attributi aggiuntivi e una chiave esterna che la collega alla tabella

della superclasse.

• Mapping orizzontale : non esiste una tabella per la superclasse ma una tabella per ciascuna
sottoclasse.

32)Descrivere i concetti di Fallimento, Stato erroneo e Fault (difetto).

Fallimento : qualsiasi deviazione del comportamento osservato dal comportamento atteso.
Stato erroneo : il sistema è in uno stato tale che ogni ulteriore elaborazione da parte del sistema conduce ad un fallimento.
Fault (difetto) : la causa meccanica o algoritmica di un fallimento

33)Descrivere la differenza tra Fallimento Meccanico ed Algoritmico.

Fallimento Meccanico : è un fallimento che è causato da eventi indipendenti dall'implementazione, sia software che hardware.
Fallimento algoritmico : è un fallimento che dipende da un errata implementazione.

34)Descrivere le differenze tra Testing Blackbox e Testing Whitebox.

Blackbox testing indica un testing per le funzionalità esterne del sistema(si focalizza sui valori di input/output). Whitebox testing indica invece il testing del codice indipendentemente dall'input, ogni statement viene testato almeno una volta e vengono testate ogni stato del modello dinamico ed ogni interazione tra gli oggetti(statement, loop, branch e path,si focalizza sulla completezza)

35) Elencare i diversi tipi di test di usabilità e descrivere una delle tecniche.

Scenario test :

• Viene presentato a uno o più utenti un Visionary Scenario

• Gli sviluppatori determinano quanto velocemente gli utenti comprendono lo scenario, la bontà del modello e come reagiscono gli utenti alla descrizione del nuovo sistema.

Lo scenario selezionato dovrebbero essere il più realistico possibile.
Vantaggi: sono economici da realizzare e da ripetere
Svantaggi: gli utenti non possono interagire direttamente con il sistema.

Test di prototipo :

Agli utenti finali viene presentato una parte del software che implementa gli aspetti chiave del sistema :

• Prototipo verticale. Implementa completamente uno use case.

• Prototipo orizzontale. Implementa un singolo layer nel sistema (per esempio un prototipo dell'interfaccia utente)

• Prototipo funzionale. Usato per valutare le richieste cruciali (es: tempo di risposta)

Vantaggi: forniscono una vista realistica del sistema all'utente ed il prototipo può essere concepito per collezionare informazioni dettagliate
Svantaggi: richiede un impegno maggiore nella costruzione rispetto agli scenari cartacei

Test di prodotto:

Simile al test di prototipo, eccetto per il fatto che viene utilizzata una versione funzionale del
sistema. Il test può essere affrontato solo dopo che una buona parte del sistema sia stata sviluppata.

36) Elencare i diversi tipi di test di unità e descrivere una delle tecniche.

Il testing di unità si concentra sui blocchi base del sistema: gli oggetti e i sottosistemi.
Ci sono vari tipi di test di unitò:

• Equivalence Testing: è una tecnica black-box che minimizza il numero di casi di test. I casi

di test sono divisi in classi di equivalenza e viene preso un solo caso di test da ogni classe.

• Boundary Testing: è un caso specifico di test di test di equivalenza, che si focalizza sui casi
limite delle classi di equivalenza.

• Path Testing: è una tecnica white-box che identifica fault nell'implementazione di una componente. L'assunzione dietro al path testing è che provando tutti i possibili percorsi del codice almeno una volta tutte le fault scateneranno delle rispettive failure.

• State-based Testing: è una tecnica di testing recente per rilevare failure relative ad caratteristiche degli ambienti ad oggetti, come polimorfismo, binding dinamico e distribuzione di funzionalità su un grande numero di piccoli metodi.

37) Elencare i diversi tipi di test di integrazione e descrivere una delle tecniche.

Big-Bang Integration: consiste nel testare tutte le componenti individualmente e poi unirle; questa tecnica è sconsigliata perché in caso di fault non si può capire facilmente in quale sottosistema risiede.

Bottom-Up Testing: Testa prima i sottosistemi risiedenti agli strati inferiori, avvalorandosi dell'uso dei Driver per simulare quelli superiori, poi vengono testati quelli che chiamano i sottosistemi. Quest'approccio non è consigliato per sistemi decomposti funzionalmente, perché testa i sottosistemi importanti solo alla fine.

Top-Down Testing: Questa strategia testa prima i sottosistemi che risiedono nello strato superiore e poi l'insieme di quelli testati e quelli che sono chiamati dal componente testato

Sandwich Testing: Combina le strategie Bottom-Up e Top-Down. Si seleziona un livello target (minimizzando il numero di stub e di driver) poi vengono eseguiti I test sui livelli superiori e sui livelli inferiori contemporaneamente e successivamente integrati con il livello
target.

Sandwich Modificato: Viene testato anche il livello di target, e quindi il target con il livello superiore(che viene sostituito al driver) ed il target con il livello inferiore(che viene sostituito
allo stub)

38) Elencare i diversi tipi di test di sistema e descrivere una delle tecniche.

Testing di unità e di integrazione si focalizzano sulla ricerca di bug nelle componenti individuali e nelle interfacce tra le componenti. Il testing di sistema assicura che il sistema completo sia conforme ai requisiti funzionali e non funzionali. Attività :

• Testing funzionale. Test dei requisiti funzionali

• Pilot Testing. Test di funzionalità comuni, fra un gruppo selezionato di utenti finali, nell'ambiente target

• Testing di prestazioni. Test dei requisiti non funzionali

• Testing di accettazione. Test di usabilità, delle funzionalità e delle prestazioni effettuato dal
cliente nell'ambiente di sviluppo

• Testing di installazione. Test di usabilità, delle funzionalità e delle prestazioni effettuato dal
cliente nell'ambiente operativo

39)Descrivere i diversi tipi di test di accettazione.

Tre modi in cui il cliente può valutare un sistema durante
il testing di accettazione :

• Benchmark test. Il cliente prepara un insieme di test case che rappresentano le condizioni
tipiche sotto cui il sistema dovrà operare

• Competitor testing. Il nuovo sistema è testato rispetto ad un sistema esistente o un prodotto
competitore

• Shadow testing. Una forma di testing a confronto, il nuovo sistema e il sistema legacy sono
eseguiti in parallelo ed i loro output sono confrontati

Se il cliente è soddisfatto, il sistema è accettato, eventualmente con una lista di cambiamenti
da effettuare.