A cura di Karina Chichifoi - Ultima revisione: 3 luglio 2020

Ingegneria del Software Q&A

https://github.com/TryKatChup/IngegneriaSoftware_QA

Premessa

Ho scritto questo file in modo da facilitare lo studio e il superamento dell'esame di Ingegneria del Software. Tuttavia è consigliato integrare questo materiale con le slide del professore Marco Patella, disponibili sulla piattaforma *Insegnamenti Online*.

Contribuire alla guida

Se ritieni di poter migliorare la guida, oppure se sono state aggiunte altre domande al di fuori di questo file, o se hai trovato un errore, visita la repository GitHub ed apri una *issue*, oppure inviami un messaggio. Ogni contributo è ben accetto :)

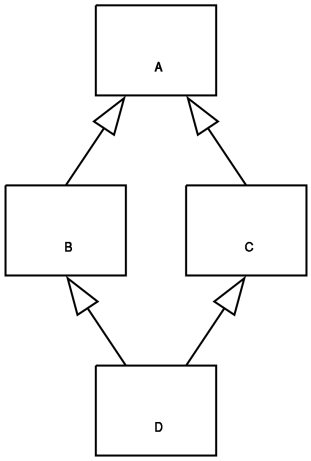
Link Repository: https://github.com/TryKatChup/IngegneriaSoftware_QA

Figura 1: QR Code alla repository di GitHub

1 Modulo 1

Domanda 1.1
Come viene implementata l'ereditarietà multipla?

Risposta: L'ereditarietà multipla si verifica quando, data una gerarchia di classi, almeno una classe della gerarchia deriva da due o più superclassi.
In C# e Java l'ereditarietà multipla non è consentita, in quanto si generano numerose ambiguità e la gestione della gerarchia di classi può diventare complessa; in C++ viene ancora utilizzata.
Un esempio di ambiguità si riscontra nel *problema del diamante* (il nome del problema deriva dalla forma che l'ereditarietà delle classi assume). Siano `A`, `B`, `C`, `D` quattro classi definite nel seguente modo:



- `A` possiede un metodo `doSomething()`
- `B`, `C` sono classi figlie di `A` che ridefiniscono entrambe tale metodo.
- `D` eredita sia da `B` che da `C`.

L'ambiguità si presenta dal momento in cui non è noto quale implementazione di `doSomething()` `D` erediti.
C# e Java non adottano l'ereditarietà multipla delle classi, bensì delle interfacce, poiché queste ultime non specificano il comportamento di un metodo, ma solo la sua firma.
Un esempio di ereditarietà multipla delle interfacce è il seguente:

```
interface flyable() {
   void fly();
}

interface swimmable() {
   void swim();
}

class Seaplane implements flyable, swimmable {
   public void fly {
           System.out.println("I'm flying");
   }
   public void swim {
           System.out.println("I'm swimming");
   }
}
```

Domanda 1.2
Si esegua una classificazione del polimorfismo secondo Cardelli-Wegner e si mostri l'implementazione del polimorfismo per inclusione.

Risposta: Si definisce polimorfismo la capacità di un elemento di apparire in forme diverse in differenti contesti, o di elementi diversi di apparire sotto la stessa forma in uno specifico contesto. La classificazione di Cardelli-Wegner impone due categorie, a loro volta suddivise in due sottocategorie:

- **Universale**: gli elementi assumono infinite forme.
  È suddiviso in:
  - **Per inclusione**: viene utilizzato nella prorgrammazione orientata agli oggetti e utilizza:
    - *Overriding*: consente la ridefinizione di un metodo della superclasse nella sottoclasse. Questo approccio risulta più sicuro nel caso in cui il metodo in questione risulta astratto.
    - *Binding dinamico*: viene consentito grazie all'utilizzo della Virtual Method Table (VMT), posseduta da ogni classe; in particolare una VMT contiene tutti i puntatori ai metodi della classe che la possiede.
  - **Parametrico**: viene utilizzato nella programmazione generica rispetto ai tipi. Consiste nel definire una classe in cui il tipo di una o più variabili è un parametro della classe stessa. Da ogni classe generica si generano classi indipendenti, che non possiedono alcun rapporto di ereditarietà.
- **Ad hoc**: gli elementi assumono un numero finito di forme. È suddiviso in:
  - **Overloading**:
  - **Coercion**: viene effettuata una conversione implicita del tipo di una variabile.

Un esempio di polimorfismo per inclusione in cui si utilizza l'overriding è il seguente:

```
1  public class A {
2     public virtual void Fun1(int x) {
3              ...
4     }
5     public virtual void Fun2(int y) {
6              ...
7     }
8  }
9
10 public class B : A {
11    public override void Fun1(int x) {
12             ...
13    }
14    public virtual void Fun3(int z) {
15             ...
16    }
17 }
```

Domanda 1.3
Procedimento di compilazione ed esecuzione del codice all'interno del framework .NET tramite il CLR.

Risposta: Il *Common Language Runtime* (CLR) viene utilizzato in .NET come ambiente di esecuzione in runtime delle applicazioni.
Il codice che viene eseguito in CLR prende il nome di *codice gestito*.
Il CLR prevede servizi aggiuntivi come:

- **Garbage collector**: si occupa del ciclo di vita degli oggetti; qualora un oggetto non risulti più essere referenziato viene distrutto.
  A differenza di *Component Object Model* (COM), non viene considerato il *reference counting*, ovvero il conteggio dei riferimenti a ciascun oggetto; in questo modo si ha una velocità di allocazione maggiore. Sono inoltre consentiti i *riferimenti circolari*, ovvero più oggetti che puntano nel seguente modo: Con

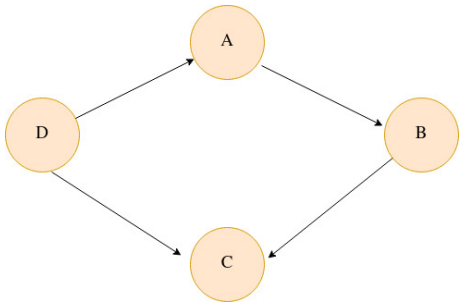


Figura 2: Riferimenti circolari

  questo approccio, tuttavia, si verifica la perdita della *distruzione deterministica*, ovvero una richiesta esplicita di liberazione della memoria occupata da un oggetto.

- **I/O su file**

- **Gestione delle eccezioni**: le eccezioni sono oggetti che ereditano dalla classe *System.Exception*.
  È possibile gestire le eccezioni sfruttando i seguenti tre concetti:
  - *throw*: lancio di un' eccezione
  - *catch*: cattura di un' eccezione
  - *finally*: esecuzione di codice di uscita da un blocco controllato

Domanda 1.4
Tipi di dati in .NET

Risposta:

Domanda 1.5
Differenza tra tipi valore e tipi riferimento in .NET

Risposta:

Domanda 1.6
Garbage Collector in C#

Risposta:

Domanda 1.7
Passaggio dei parametri in C#

Risposta:

Domanda 1.8
Concetto di delegato in C#

Risposta:

Domanda 1.9
Metaprogrammazione e riflessione in C#

Risposta:

Domanda 1.10
Spiegare i quattro bad design (fragilità, immobilità, rigidità, viscosità)

Risposta: La qualità del design risulta fondamentale per rendere un programma:

- Maggiormente affidabile
- Maggiormente efficiente
- Maggiormente manutenibile
- Maggiormente economico

Esistono quattro principi che peggiorano la qualità del software:

1. **Rigidità del software**: rende complesse le modifiche al software, in quanto una piccola modifica influenza gran parte del programma, a causa di modifiche a cascata su moduli dipendenti tra loro.
   Una conseguenza di questo principio è la quantità di tempo necessaria a gestire una modifica del software.
2. **Fragilità del software**: responsabile del malfunzionamento del software di fronte a una modifica di quest'ultimo; il software risulta difficile da mantenere, in quanto per ogni correzione è presente un rischio maggiore di guasto.
3. **Immobilità del software**: rende il software inutilizzabile da parti dello stesso progetto o da altri progetti; ciò avviene quando un modulo software è fortemente dipendente da altri moduli.
   Di conseguenza occorre scrivere nuovo software, anziché riutilizzarlo.
4. **Viscosità del software**: questo principio favorisce l'utilizzo di *hack*, ovvero soluzioni funzionanti, ma che snaturano il design, in quanto non seguono le *best practice*.
   La viscosità è maggiormente presente nel caso in cui utilizzo di hack risulta molto più semplice di una soluzione che preserva la progettazione iniziale.
   Esistono due categorie di viscosità:
   - **Viscosità del design**: l'utilizzo di metodi che rispettano il design risulta più complesso che utilizzare hack.
   - **Viscosità dell'ambiente**: l'ambiente di sviluppo è lento e inefficiente: si possono avere tempi di compilazione molto lunghi, il sistema di controllo del codice sorgente richiede ore per la registrazione di pochi file.

I motivi di una progettazione non corretta sono dovuti a:

- Incapacità dei progettisti nel seguire le *best practice*
- Limiti imposti dall'esterno, in termini di tempo e risorse
- Pratiche obsolete
- Evoluzione del progetto: in seguito a cambiamenti dei requisiti e modifiche al software, non si ha più la manutenibilità del progetto originario

Domanda 1.11
Principio di singola responsabilità con almeno un esempio

Risposta: Il principio di singola responsabilità afferma che ogni elemento di un programma (classe, metodo, variabile) deve avere un'unica responsabilità, interamente gestita dall'elemento stesso. Tutti i servizi offerti dall'elemento stesso dovrebbero essere allineati a tale responsabilità.
Una responsabilità risulta un motivo per cambiare, e ogni classe o modulo deve avere un solo motivo per cambiare.
Un esempio che rispetta il principio di singola responsabilità è un metodo *setter*, in quanto la sua unica responsabilità è quella di impostare il valore di una variabile appartenente a una classe.

Domanda 1.12
Principio di inversione delle dipendenze con almeno un esempio

Risposta: Il principio di inversione delle dipendenze prevede che i moduli di alto livello (i *clienti*) non debbano dipendere dai moduli di basso livello(ovvero i *fornitori dei servizi*). Entrambi devono dipendere da *astrazioni*.
Il motivo della precedente affermazione è dovuto alla presenza di codice e della logica implementativa nei moduli di basso livello. Nel caso in cui i moduli di alto livello dovessero dipendere da moduli di basso livello si avrebbe:

- **Rigidità**: per ogni modifica occorre intervenire su un numero elevato di moduli
- **Fragilità**: per ogni modifica verrebbero introdotti errori nel sistema
- **Immobilità**: non è possibile riutilizzare il codice, in quanto i moduli di alto livello non si riuscirebbero a separarare da quelli di basso livello.

Con il principio di inversione delle dipendenze si eviterebbero i problemi sopra elencati, in quanto le astrazioni possiedono poco codice e sono poco soggetti a modifiche; inoltre è presente una separazione tra moduli astratti e moduli concreti, che permette modifiche limitate al modulo concreto interessato (poichè nessuno dipende da questi moduli). Dato che i dettagli del sistema sono stati isolati da un muro di astrazioni stabili, pertanto:

- I cambiamenti non possono più propagarsi (*design for change*)
- I singoli moduli sono maggiormente riusabili (*design for resue*)

Domanda 1.13
Principio di segregazione delle interfacce con almeno un esempio

Risposta: Questo principio prevede che un cliente non debba dipendere dai metodi che non usa, e che quindi è preferibile avere più interfacce specifiche, anzichè una sola interfaccia con più funzioni (*fat interface*).
Nel caso di utilizzo delle *fat interfaces* sarà più difficile mantenere il sistema, in quanto è rappresentato da un unico blocco; pertanto occorre creare interfacce specifiche per ogni cliente.
Un esempio di mancato utilizzo di principio di segregazione delle interfacce è la creazione, da parte di Xerox (fondatore di *Ethernet*), di una stampante che aveva diverse funzionalità, tra cui mandare fax e pinzare fogli. Le modifiche al software, sebbene piccole, risultavano difficili, in quanto tutte le diverse funzioni eseguite dalla stampante multi-uso erano implementate all'interno di una sola classe.

Domanda 1.14
Principio aperto/chiuso con almeno un esempio

Risposta: Il principio aperto/chiuso prevede un sistema aperto a estensioni software, ma chiuso a modifiche. Per realizzare questo principio vengono utilizzate classi astratte e interfacce; nel caso in cui si vogliano aggiungere nuove funzionalità sarà necessario, anzichè modificare il codice, implementare una nuova classe concreta che implementa le astrazioni.
Questo approccio consente una maggiore modularità del sistema e stabilità, in quanto non si modificano componenti definite precedentemente e non si introducono eventuali errori dovuti alle modifiche.

Domanda 1.15
Principio di sostituibilità di Liskov con almeno un esempio

Risposta: Il principio di sostituibilità di Liskov definisce il rapporto tra una classe e le relative sottoclassi. Esistono due versioni di questo principio:

- **Versione debole**: a ogni riferimento della classe base (classe padre) deve essere associata l'istanza della sottoclasse (classe figlia)
- **Versione forte**: ogni programma che utilizza istanze della classe base non percepisce variazioni logiche nell'utilizzo delle istanze delle relative sottoclassi.

Per essere più specifici, occorre introdurre due concetti definiti dal *design by contract*:

- **Precondizioni**: requisiti minimi che devono essere soddisfatti dal chiamante, affinchè il metodo invocato possa essere eseguito correttamente
- **Postcondizioni**: requisiti soddisfatti dal metodo chiamato, nel caso di esecuzione corretta

Per ogni metodo re-implementato dalla sottoclasse:

- Le precondizioni devono essere ugualmente o meno stringenti
- Le postcondizioni devono essere ugualmente o più stringenti
- La semantica della classe base deve essere conservata
- Non è possibile aggiungere vincoli alla classe base

Ad esempio, data la classe `Uccello` dotata del metodo `vola()` si hanno i due seguenti scenari:

- la sottoclasse `Pinguino`, che eredita da `Uccello`, presenta una violazione del principio di sostituibilità di Liskov, in quanto l'utilizzo del metodo `vola()` deve essere ridefinito, senza mantenere la semantica (ad esempio sollevando un'eccezione, dato che un pinguino non può volare).
- la sottoclasse `Piccione`, che eredita da `Uccello`, non presenta una violazione del principio di sostituibilità di Liskov, in quanto la semantica del metodo `vola()` viene rispettata.

Domanda 1.16
Principi per l'architettura dei package

Risposta:

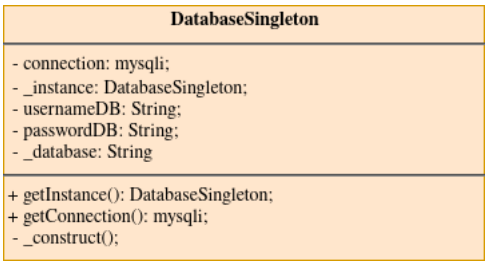
Domanda 1.17
Pattern Singleton con esempi

Risposta: Il pattern Singleton prevede che una classe abbia una sola istanza; chiunque può accedervi attraverso l'unica istanza citata. Una classe che implementa il pattern Singleton è strutturata nel sequente modo:

```
1  public class Singleton {
2
3     private static Singleton istance = null; // riferimento all' istanza
4
5     private Singleton() {} // costruttore
6
7     public static Singleton getIstance() {
8          if (istance == null)
9             istance = new Singleton();
```

```
10      return istance;
11    }
12
13    public void metodo() { ... }
14
15  }
```

Il costruttore è privato, in quanto assicura che viene creata al più un'istanza della classe `Singleton`. Il metodo statico `getInstance()` controlla se è già esistente un'istanza della classe: in caso affermativo restituisce l'istanza creata in precedenza, altrimenti crea una nuova istanza.
Una classe con soli membri statici non rappresenta un'alternativa al pattern Singleton, in quanto non permette di creare istanze personalizzate in base al contesto; non permette inoltre di utilizzare un numero arbitrario di interfacce.
Un esempio di utilizzo del pattern Singleton è l'accesso a un database, in quanto si vuole garantire atomicità.



Domanda 1.18
Pattern Observer con esempi

Risposta: Il pattern Observer viene utilizzato nel caso in cui il cambiamento di un oggetto preveda un aggiornamento degli altri oggetti (*observer*). È previsto un accoppiamento uno a molti: l'oggetto che vuole ricevere un aggiornamento (`Observer`) si iscrive a `Subject`, che prevede un metodo `notify()`, il quale notifica tutti gli Observer dell'aggiornamento effettuato. Il diagramma UML previsto per il pattern Observer è il seguente:

Domanda 1.19
Pattern Strategy con esempi

Risposta:

Domanda 1.20
Pattern Adapter con esempi

Risposta:

Domanda 1.21
Pattern Decorator con esempi

Risposta:

Domanda 1.22
Pattern Composite con esempi

Risposta:

Domanda 1.23
Modello LMU nei VCS con vantaggi e svantaggi

Risposta:

2 Modulo 2

Domanda 2.1
Spiegare il modello a cascata e le sue criticità.

Risposta: Il modello a cascata (waterfall model) è un modello di processo di sviluppo software che prevede fasi sequenziali distinte tra loro:

- Studio di fattibilità
- Analisi dei requisiti
- Analisi del problema
- Progettazione
- Implementazione
- Collaudo
- Manutenzione

Ciascuna fase di sviluppo deve essere svolta in maniera esaustiva, prima di passare alla successiva, in modo da non tornare più indietro. Per questo modello è importante definire:

- **Semilavorati**: consistono in documentazione di tipo cartaceo, codice dei singoli moduli, sistema nel suo complesso.
  Vengono prodotti da una fase, e utilizzati dalla fase successiva; in questo modo viene garantito un controllo della qualità del lavoro eseguito in ogni fase.
- **Date**: stabiliscono una scadenza entro la quale devono essere prodotti i semilavorati, in modo da tracciare il progresso del lavoro (workflow).

L'efficacia del modello a cascata è determinata dai seguenti fattori:

- **Immutabilità dell'analisi**: i clienti sono in grado di esprimere tutte le loro richieste sin da subito, pertanto nella fase iniziale del progetto si possono definire tutte le funzionalità che il software deve eseguire
- **Immutabilità del progetto**: progettare l'intero sistema prima di avere scritto codice risulta possibile

Un importante vantaggio di questo approccio risulta essere un maggiore controllo dell'andamento del progetto; tuttavia la rigidità di questo modello rappresenta un grosso svantaggio, in quanto:

- Man mano che il sistema prende forma le sue specifiche cambiano in continuazione, così come la visione che i clienti hanno del sistema
- Spesso, per avere prestazioni migliori, occorre revisionare il progetto.

Per risolvere parzialmente i problemi sopra citati si è introdotto un modello a cascata con forme limitate di retroazione a un livello. Una possibile soluzione al problema consiste nel realizzare un prototipo che, una volta terminato il compito, viene abbandonato (*throw-away prototyping*); successivamente viene costruito il sistema reale rispettando il modello a cascata.
Tuttavia quest'ultimo approccio risulta talmente dispensioso da eliminare i vantaggi economici del modello a cascata.

Domanda 2.2
Spiegare il modello a cascata e il modello iterativo

Risposta: Per il modello a cascata si veda la domanda 2.1.
Il modello iterativo prevede un numero elevato di passi nel ciclo di sviluppo che iteramente aumentano il livello di dettaglio del sistema. Uno svantaggio di questo modello è che non può essere utilizzato nella realizzazione dei progetti significativi. Un esempio di processo di sviluppo che utilizza il modello iterativo è *Rational Unified Process* (RUP).

Domanda 2.3
Illustrare RUP

Risposta: Il *Rational Unified Process* (RUP) rappresenta un modello di processo software **iterativo** (si veda domanda 2.2) e **ibrido** (contiene elementi di tutti i modelli di processo generici) pensato per software di grandi dimensioni.
Esistono tre aspetti importanti del processo di sviluppo:

- **Prospettiva dinamica**: mostra l'evoluzione del modello nel tempo. È composta da 4 fasi:

  1. **Avvio**: lo scopo di questa fase è di delineare il *business case*, ovvero comprendere il tipo di mercato a cui si rivolge, le entità esterne (persone e sistemi) che interagiscono con il sistema.
     Durante la fase di avvio si utilizzano modelli di caso d'uso e si effettua una valutazione dei rischi.
  2. **Elaborazione**: questa fase definisce la struttura complessiva del sistema; comprende l'analisi del dominio e una prima fase di progettazione dell'architettura.
     Occorre soddisfase alcuni criteri, tra i quali:
     - Modello dei casi d'uso completo all' 80%
     - Descrizione dell'architettura del sistema
     - Sviluppo dell'architettura del sistema
     - Sviluppo di un'architettura eseguibile adatta agli use case significativi
     - Revisione dei business case e dei rischi
     - Pianificazione del progetto complessivo

     Nota bene: al termine di questa fase modificare il progetto risulterà più difficile e dannoso.
  3. **Costruzione**: durante questa fase avviene la progettazione, la programmazione e il collaudo del sistema. Lo sviluppo delle diverse parti del sistema avviene in parallelo; successivamente vengono integrate.
     Al termine di questa fase il sistema software dovrebbe essere funzionante e la relativa documentazione dovrebbe risultare pronta.
  4. **Transizione**: il sistema passa dall'ambiente di sviluppo a quello dell'utente finale. Quest'ultimo viene istruito nell'utilizzo del sistema, e si effettua *beta testing* del sistema a scopo di verifica e validazione.

- **Prospettiva statica**: si focalizza sulle attività di produzione del software, note come *workflow*; la descrizione di questi ultimi è orientata ai modelli associati a UML. Esistono sei workflow principali:

  1. **Modellazione delle attività aziendali**: i processi aziendali vengono modellati, sfruttando il *business case*
  2. **Requisiti**: vengono sviluppati i casi d'uso per la stesura dei requisiti; avviene l'identificazione degli attori che interagiscono con il sistema
  3. **Analisi e progetto**: attraverso l'utilizzo dei modelli architetturali e sequenziali degli oggetti e delle componenti viene creato e documentato un *modello di progetto*
  4. **Implementazione**: i componenti vengono implementati; grazie alla generazione automatica del codice a partire dai modelli precedentemente definiti
  5. **Test**: vengono testati i sottocomponenti e il sistema finale
  6. **Rilascio**: il prodotto viene distribuito agli utenti

  Oltre ai 6 workflow principali vengono definiti 3 workflow di supporto:

  1. **Gestione della configurazione e delle modifiche**: gestisce i cambiamenti del sistema
  2. **Gestione del progetto**: gestisce lo sviluppo del sistema
  3. **Ambiente**: fornisce agli sviluppatori degli strumenti adeguati

- **Prospettiva pratica**: suggerisce le *buone prassi* da seguire nello sviluppo dei sistemi.
  Esistono sei fasi fondamentali:

  1. **Sviluppare ciclicamente il software**: pianificare (??) e consegnare le funzioni aventi la priorità più alta
  2. **Gestire i requisiti**: documentare ogni richiesta esplicita del cliente e ogni cambiamento effettuato, analizzandone l'impatto
  3. **Usare architetture basate sui componenti**: strutturare l'architettura del sistema in più componenti
  4. **Usare modelli visivi del software**: utilizzare grafici UML per la rappresentazione statica e dinamica del software
  5. **Verificare la qualità del software**: assicurarsi che vengano raggiunti gli standard di qualità previsti dall'organizzazione
  6. **Controllare le modifiche del software**: utilizzare strumenti e pratiche che permettono di gestire modifiche al software

Domanda 2.4
Tipologie di analisi dei requisiti (in particolare quelli della sicurezza)

Risposta:

Domanda 2.5
Si illustri brevemente il ciclo di vita della valutazione del rischio

Risposta: L'analisi del rischio si occupa di bilanciare eventuali perdite, dovute ad attacchi informatici, con i costi richiesti per assicurare la protezione dei beni.
Un'importante componente dell'analisi del rischio è la valutazione del rischio, composta da pù fasi:

- **Valutazione preliminare del rischio**: determina i requisiti di sicurezza dell'intero sistema
- **Ciclo di vita della valutazione del rischio**: avviene parallelamente al ciclo di vita dello sviluppo del software.
  In questa fase occorre conoscere l'architettura del sistema e l'organizzazione dei dati.
  La scelta della piattaforma e del middleware è stata già effettuata, così come la strategia di sviluppo del sistema; ciò consente di conoscere meglio cosa è necessario proteggere e quali sono le possibili vulnerabilità del sistema, alcune delle quali determinate da scelte progettuali precedenti.
  In questa fase vengono effettuate l'*identificazione* e la *valutazione* della vulnerabilità, ovvero quali beni hanno la maggiore probabilità di essere colpiti.
  Il risultato della valutazione del rischio è un insieme di decisioni ingegneristiche che influenzano la progettazione o l'implementazione del sistema.

Domanda 2.6
Principali categorie di requisiti per la sicurezza

Risposta: Lo scopo dei requisiti di sicurezza è di definire quali comportamenti risultano inaccettabili per il sistema, senza definire le funzionalità richieste al sistema. I requisiti di sicurezza specificano il contesto, i beni da proteggere e il valore che questi ultimi hanno per l'organizzazione.
Le categorie dei requisiti per la sicurezza sono:

- **Requisiti di identificazione**: specificano se un sistema deve eseguire l'identificazione dei clienti, prima di una qualsiasi interazione con loro
- **Requisiti di autenticazione**: specificano le modalità di autenticazione degli utenti

- **Requisiti di autorizzazione**: specificano i permessi e i privilegi che gli utenti possiedono una volta identificati
- **Requisiti di immunità**: specificano i meccanismi di difesa che il sistema deve adottare per difendersi da eventuali malware
- **Requisiti di integrità**: specificano come evitare le corruzioni dei dati
- **Requisiti di scoperta delle intrusioni**: specificano quali meccanismi vengono adottati per la rilevazione degli attacchi
- **Requisiti di non-ripudiazione**: specificano che una parte interessata in una transazione non può negare il proprio coinvolgimento
- **Requisiti di riservatezza**: specificano come deve essere mantenuta la riservatezza delle informazioni
- **Requisiti di controllo della protezione**: specificano come deve essere controllato e verificato l'utilizzo del sistema
- **Requisiti di protezione della manutenzione del sistema**: specificano come un' applicazione può evitare modifiche autorizzate, nel caso in cui accidentalmente vengano annullati i meccanismi di protezione

Domanda 2.7
Commentare eventuali errori di un diagramma UML

Risposta:

Domanda 2.8
Linee guida di progettazione nella sicurezza

Risposta:

Domanda 2.9
White box e black box testing

Risposta:

Domanda 2.10
Capacità di sopravvivenza del sistema

Risposta: