A cura di Karina Chichifoi - Ultima revisione: 1 marzo 2026

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*.

Nota bene: gli esempi riportati su questo pdf hanno lo scopo di facilitare la comprensione delle applicazioni di determinati design pattern. Per superare al meglio l'esame occorre proporre nuovi esempi, dimostrando di avere effettivamente capito i concetti trattati.
Copiare non vi aiuterà in alcun modo.

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

Registro delle Modifiche

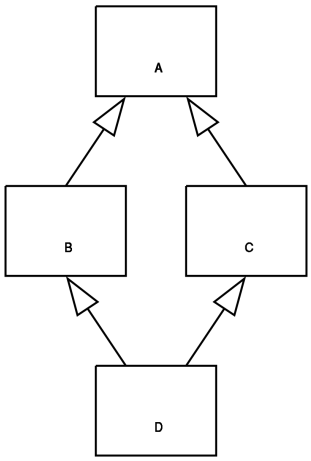
Il presente documento, originariamente redatto da **Karina Chichifoi**, è stato integrato e revisionato da **Davide Chirichella** in data 7 Luglio 2025 con l'aggiunta dei seguenti argomenti:

- **1.26** ADT ed incapsulamento
- **1.27** Differenze tra interfaccia e classe astratta
- **1.28** Ciclo di vita di un oggetto (con esempi OOP)
- **1.29** Fattori di qualità di un buon software
- **1.30** Pattern Abstract Factory
- **1.31** Pattern State
- **1.32** Pattern MVP ed MVC

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");
   }
}

```

Un metodo alternativo all'ereditarietà multipla è il modello **composizione e delega**, dove si sceglie una superclasse significativa e si eredita soltanto da quella; le rimanenti superclassi diventano ruoli e vengono connesse tramite **composizione**. Un esempio è il seguente:

- Si supponga di avere la classe `Pipistrello` che eredita sia dalla classe `Mammifero` che dalla classe `Volatile`.
- `Mammifero` e `Volatile` ereditano da `Animale`.
- Dato che l'ereditarietà multipla non è consentita si può adottare il seguente approccio:
  - `Animale` rimane superclasse, e `Mammifero` eredita da `Animale`.
  - `Pipistrello` deriva da `Mammifero` ed è legato tramite composizione a `Volatile`.

Esiste un ulteriore approccio ed è una combinazione dei metodi *composizione-delega* e *interfacce*.

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**: consente la ridefinizione di metodi/operatori per ogni insieme di argomenti accettati; essa deve avvenire in fase di programmazione.
  - **Coercion**: viene effettuata una conversione implicita del tipo di una variabile. Le conversioni possibili devono essere definite in fase di programmazione.

Un esempio di polimorfismo per inclusione in cui si utilizza *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  }
```

Il codice sopra consente di mostrare un esempio di polimorfismo per inclusione: viene utilizzato *overriding* per ridefinire il metodo `Fun1(int x)` della superclasse `A` in `B`. Il *binding dinamico* viene utilizzato per discriminare il metodo richiamato; la *Virtual Method Table* di ciascuna classe punta al metodo evocato.

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 virtuale di esecuzione delle applicazioni. Il codice che viene eseguito in CLR prende il nome di *codice gestito*.
Il codice sorgente viene trasformato dal compilatore .NET in codice IL (CLI assembly, ovvero .exe o .dll); un assembly contiene, oltre al codice IL, anche un *manifest* che fornisce informazioni come i tipi di assembly, la versione, e requisiti di sicurezza. Il codice IL a sua volta viene convertito dal compilatore *Just In Time* (JIT) in codice nativo, che può essere eseguito.
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:

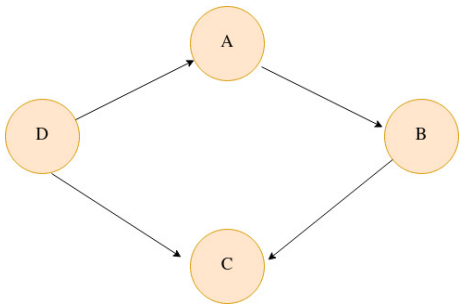


Figura 2: Riferimenti circolari

  Con questo approccio, tuttavia, si verifica la perdita della *distruzione deterministica*, ovvero una richiesta esplicita di liberazione della memoria occupata da un oggetto.
  Tramite *garbage collector* la memoria viene liberata in modo non deterministico, ovvero quando l'oggetto non risulta più raggiungibile.

- **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
Differenza tra tipi valore e tipi riferimento in .NET

Risposta: I *Common Type System* (CTS) sono tipi di dato supportati dal framework .NET, che forniscono un modello di programmazione unificato ai linguaggi orientati agli oggetti, funzionali e procedurali. In CTS tutte le classi ereditano da `System.Object`, ed esistono due categorie di tipi:

- **Tipi riferimento**: sono indirizzi di memoria che rappresentano i riferimenti agli oggetti allocati sull'heap gestito.
- **Tipi valore**: sono allocati sullo stack, o appartengono ad altri oggetti, e contengono direttamente un valore (ovvero una sequenza di byte).
  I tipi valore includono:
  - **Tipi primitivi**: Int32, double, decimal, char, boolean, etc.
  - **Tipi definiti dall'utente**: strutture dati ed enumerativi.

È possibile eseguire conversioni da tipo valore a tipo riferimento mediante un *up cast* **implicito** a `System.Object`: questa procedura prende il nome di *boxing*:

```
double d = 333.3;
Object obj = d;
```

L'*unboxing*, operazione inversa al *boxing*, permette di convertire un tipo riferimento a un tipo valore, tramite un *down cast* **esplicito**:

```
double d2 = (double)obj;
```

Il passaggio dei parametri a un metodo può avere risultati distinti, in base alla tipologia di oggetto passato come argomento. Esistono tre tipologie di argomenti:

- **In**
  - *Tipi valore*: si ha passaggio per copia dell'oggetto e la modifica da parte del metodo invocato avviene solo sulla copia.
  - *Tipi riferimento*: si ha passaggio per copia del riferimento dell'oggetto, e la modifica da parte del metodo invocato avviene sulla copia del riferimento, ma non sul riferimento originale.

  In entrambi i casi gli argomenti passati ai metodi devono essere stati già inizializzati. Un esempio è il seguente:

```
1
2  PostoRistorante posto = new PostoRistorante(1,1);
3  assegnaPosto(posto);
4  Console.writeline(posto); // se PostoRistorante è una classe si ha (3,5)
5                            // se è una struttura si ha (1,1)
6
7  static void assegnaPosto(PostoRistorante posto){
8          posto.NumeroTavolo = 3;
9          posto.NumeroPosto = 5;
10 }
```

- **In/Out**
  - *Tipi valore*: si ha passaggio per riferimento e le modifiche influenzano l'oggetto originale.
  - *Tipi riferimento*: si ha passaggio per riferimento dell'*indirizzo dell'oggetto* e le modifiche influenzano l'oggetto referenziato, ma non l'oggetto originale.

In entrambi i casi gli argomenti passati ai metodi devono essere stati già inizializzati.

```

PostoRistorante posto = new PostoRistorante(1,1);
assegnaPosto(ref posto);
Console.writeline(posto); // sia se PostoRistorante è una classe,
                          // sia se è una struttura si ha (3,5)

static void assegnaPosto(ref PostoRistorante posto){
        posto.NumeroTavolo = 3;
        posto.NumeroPosto = 5;
}
```

- **Out**: si ha passaggio, sia per gli oggetti di tipo riferimento che di tipo valore, dei loro indirizzi, e le modifiche hanno effetto sul chiamante.
  L'inizializzazione degli oggetti deve avvenire nel metodo in cui sono stati passati come argomenti. Un esempio è il seguente:

```

PostoRistorante posto;
assegnaPosto(out posto);

static void assegnaPosto(out PostoRistorante posto){
        posto.NumeroTavolo = 3; posto.NumeroPosto = 5;  // Dà un errore di compilazione
        posto = new PostoRistorante(3,5); // È necessario inizializzarlo
}
```

Domanda 1.5
Garbage Collector in C#

Risposta: Il *Garbage Collector* si occupa del rilascio delle risorse qualora esse non vengano più utilizzate da un oggetto. Consente di avere maggiore stabilità del programma, poiché evita che un programmatore manipoli direttamente puntatori ad aree di memoria.
Il vantaggio dell'utilizzo di un garbage collector è il contrasto delle seguenti problematiche:

- **Dangling pointer**: puntatori ad aree di memoria deallocate precedentemente, che potrebbero essere state successivamente assegnate a un altro oggetto.
- **Doppia deallocazione**: causata da più chiamate consecutive di deallocazione della stessa area di memoria.
- **Memory leak**: un oggetto non più utilizzato non viene deallocato, pertanto continua a occupare memoria.

Gli svantaggi invece sono:

- vengono richieste maggiori risorse di calcolo;
- incertezza del momento in cui viene effettuata la garbage collection;
- il rilascio della memoria è non deterministico, ovvero non si sa il momento esatto in cui il rilascio avviene, né l'ordine di rilascio delle aree non più utilizzate. Ciò può dipendere dall'algoritmo utilizzato dal garbage collector.

L'ambiente .NET sfrutta come strategia di *garbage collection* il **tracing**: si stabiliscono quali oggetti sono raggiungibili e si eliminano quelli non raggiungibili.
Quando un processo viene inizializzato il *Common Language Runtime* (CLR) riserva una regione contigua di spazio di indirizzamento, noto come *managed heap* e memorizza l'indirizzo di partenza della regione in un puntatore chiamato `NextObjPtr`. Nel caso in cui venga eseguita una *newobj*, il CLR:

1. Determina la dimensione in byte dell'oggetto e aggiunge a quest'ultimo due campi che possono essere da 32 o 64 bit. Il primo campo contiene un puntatore alla tabella dei metodi, mentre il secondo è un campo `SyncBlockIndex`.
2. Controlla se a partire da `NextObjPtr` ci sia spazio sufficiente; in caso negativo viene utilizzato il Garbage Collector, o viene lanciata `OutOfMemoryException`.
3. Aggiorna i puntatori relativi all'oggetto appena creato e allo spazio libero di memoria:

```
thisObjPtr = NextObjPtr;
NextObjPtr += sizeof(oggetto);
```

4. Invoca il costruttore dell'oggetto.
5. Restituisce il riferimento all'oggetto.

Lo scopo del garbage collector è di individuare quali oggetti non vegono più utilizzati dall'applicazione; quest'ultima presenta un insieme di radici (*root*). Ciascuna radice è un puntatore a un oggetto di tipo riferimento sicuramente attivo, come:

- variabili globali e field statici di tipo riferimento;
- variabili locali o argomenti attuali di tipo riferimento presenti sugli stack dei vari thread;
- registri della CPU contenenti gli indirizzi di oggetti di tipo riferimento.

Vengono distinte di tipologie di oggetti: gli **oggetti vivi** sono raggiungibili (direttamente o indirettamente) dalle radici, mentre gli **oggetti garbage** non lo sono.
Inizialmente il garbage collector marca tutti gli oggetti sul *managed heap* come **garbage**; successivamente viene interpellata la tabella delle radici, per stabilire quali oggetti siano **vivi**, marcandoli come tali.
Terminata la classificazione degli oggetti viene liberata la memoria occupata dagli oggetti garbage; tuttavia ciò causa frammentazione nel managed heap, poichè si hanno aree di memoria libere non contigue.
Pertanto si effettua una compattazione della memoria in uso, modificando di conseguenza i riferimenti agli oggetti spostati; al termine dell'unificazione si aggiorna il valore di `NextObjPtr`.
Il Garbage collector può effettuare tutte le operazioni elencate in precedenza, in quanto conosce il tipo di un oggetto e può sfruttare i metadati per determinare quali campi dell'oggetto contengono riferimenti agli altri oggetti.
Nel caso in cui un oggetto contenga riferimenti a risorse di tipo unmanaged (file, connessione al database, socket, e così via) è responsabilità del programmatore occuparsi della fase di finalizzazione, ovvero il rilascio della risorsa prima della deallocazione.

Domanda 1.6
Passaggio dei parametri in C#

Risposta: Il passaggio dei parametri a un metodo può avere risultati distinti, in base alla tipologia di oggetto passato come argomento. Esistono tre tipologie di argomenti:

- **In**
  - *Tipi valore*: si ha passaggio per copia dell'oggetto e la modifica da parte del metodo invocato avviene solo sulla copia.

  - *Tipi riferimento*: si ha passaggio per copia del riferimento dell'oggetto, e la modifica da parte del metodo invocato avviene sulla copia del riferimento, ma non sul riferimento originale.

In entrambi i casi gli argomenti passati ai metodi devono essere stati già inizializzati. Un esempio è il seguente:

```

PostoRistorante posto = new PostoRistorante(1,1);
assegnaPosto(ref posto);
Console.writeline(posto); // sia se PostoRistorante è una classe si ha (3,5)
                          // se è una struttura si ha (1,1)

static void assegnaPosto(ref PostoRistorante posto){
        posto.NumeroTavolo = 3;
        posto.NumeroPosto = 5;
}
```

- **In/Out**
  - *Tipi valore*: si ha passaggio per riferimento e le modifiche influenzano l'oggetto originale
  - *Tipi riferimento*: si ha passaggio per riferimento dell'*indirizzo dell'oggetto* e le modifiche influenzano l'oggetto referenziato, ma non l'oggetto originale

In entrambi i casi gli argomenti passati ai metodi devono essere stati già inizializzati.

```

PostoRistorante posto = new PostoRistorante(1,1);
assegnaPosto(ref posto);
Console.writeline(posto); // sia se PostoRistorante è una classe,
                          // sia se è una struttura si ha (3,5)

static void assegnaPosto(ref PostoRistorante posto){
        posto.NumeroTavolo = 3;
        posto.NumeroPosto = 5;
}
```

- **Out**: si ha passaggio, sia per gli oggetti di tipo riferimento che di tipo valore, dei loro indirizzi, e le modifiche hanno effetto sul chiamante.
  L'inizializzazione degli oggetti deve avvenire nel metodo in cui sono stati passati come argomenti. Un esempio è il seguente:

```

PostoRistorante posto;
assegnaPosto(out posto);

static void assegnaPosto(out PostoRistorante posto){
        posto = new PostoRistorante(3,5);
}
```

Domanda 1.7
Concetto di delegato in C#

Risposta: I delegati sono oggetti che contengono un riferimento a un metodo da invocare.
Sono equivalenti ai puntatori a funzioni (*functor*) presenti nei linguaggi C/C++, ma orientati agli oggetti ed efficaci.
Eseguono funzionalità di *callback*, tra le quali:

- **Elaborazione asincrona**: permette a un metodo di accettare un delegato come parametro e di chiamare il delegato in un momento successivo. Questo tipo di elaborazione viene eseguita quando un processo impiega molto tempo ad essere completato; quando viene terminato il chiamante viene avvertito.
- **Elaborazione cooperativa**: viene fornita una parte del servizio dal metodo **chiamato** e la parte rimanente viene effettuata dal **chiamante**.
  Un esempio di elaborazione cooperativa è il *merge sort*.
- **Gestione degli eventi**: ogni entità interessata a un determinato evento si registra presso il generatore dell'evento, specificando quale metodo gestirà l'evento.

L'istanza di un delegato può incapsulare uno o più metodi, specificando gli argomenti di ciascun metodo e il valore restituito. Ciascun metodo è riferito a un'entità richiamabile che può essere:

- un metodo, nel caso di metodi statici;
- un'istanza e il metodo relativo a quell'istanza, nel caso di istanze di metodi.

Un delegato contiene soltanto la firma del metodo, e non conosce la classe o il metodo a cui si sta riferendo; è quindi ideale per l'invocazione anonima. Nel caso in cui si abbia l'invocazione di un'istanza delegata, che possiede una o più voci nell'elenco di invocazione, si invocano i metodi dell'elenco in ordine e in modo sincrono. Il delegato è inoltre un'entità type-safe che si pone tra un chiamante e zero o più *call target* e che si comporta come un'interfaccia con un solo metodo. Per ciascun metodo invocato vengono passati come argomenti gli stessi forniti all'istanza del delegato. Si hanno due scenari:

- Nel caso in cui vengono inclusi **parametri di riferimento** nell'invocazione del delegato, ciascuna invocazione del metodo avviene con un riferimento alla medesima variabile e le modifiche alla variabile da parte di un metodo nell'elenco di invocazione saranno visibili ai metodi successivi nell'elenco di invocazione.
- Nel caso di invocazione del delegato con **parametri di output** o **valori di ritorno** il loro valore finale sarà determinato dall'invocazione dell'ultimo delegato presente nell'elenco.

Un esempio di utilizzo di delegati è il seguente:

- Si ha un impiegato che deve effettuare una specifica attività
- Si ha un capo, il cui scopo è di controllare l'attività dei suoi impiegati
- Un impiegato deve comunicare al suo capo l'inizio dell'attività, lo svolgimento e l'eventuale conclusione: per farlo sfrutta un delegato che contiene un riferimento alle sue attività.

Domanda 1.8
Concetto di evento in C#

Risposta: Un evento viene utilizzato al posto di un delegato in quanto quest'ultimo utilizza campi pubblici per la registrazione di metodi.
`Event` consente di rendere automatico il supporto per la pubblica registrazione e l'implementazione privata. Si possono fornire gestori di registrazione eventi definiti dall'utente: un possibile vantaggio di questo approccio è il *controllo*.
Un evento viene scaturito dall'interazione con l'utente o in seguito alla logica del programma. Esistono tre entità relative agli eventi:

- **Event sender**: la sorgente dell'evento è l'oggetto o la classe che scaturisce l'evento.
- **Event receiver**: l'oggetto o la classe per il quale l'evento è determinante e quindi desidera di venire notificato in caso di verifica dell'evento
- **Event handler**: il metodo dell'*event receiver*, eseguito nel momento in cui avviene la notifica.

Quando un evento si verifica, il sender invia un messaggio di notifica a tutti i *receiver*; in genere il *sender* non ha modo di conoscere né i gli *handler*, né i *receiver* e grazie all'utilizzo di delegati si collegano *sender* e *receiver*, consentendo *invocazioni anonime*. L'evento incapsula il delegato, pertanto occorre dichiarare un tipo di delegato prima di dichiarare un evento. I delegati possiedono due parametri:

- la fonte che ha scatenato l'evento;
- dati dedicati all'evento.

Molti eventi non possiedono dati, pertanto è sufficiente utilizzare il delegato presente nella classe `System.EventHandler`. Occorre definire nuovi delegati per gestire gli eventi soltanto quando questi ultimi possiedono dei dati. Nel caso in cui un evento non debba passare ulteriori informazioni ai propri gestori viene sfruttata la classe `System.EventArgs`, aggiungendo i dati necessari e utilizzando il delegato `EventHandler<TEventArgs>`.
Nel caso in cui si dichiari un evento, come

```
Public event EventHandler changed
```

La keyword `event` ne limita sia le possibilità di utilizzo che la visibilità; successivamente l'evento viene trattato come un delegato di tipo speciale, e può diventare `null` se nessun cliente si è registrato; può venire associato a uno o più metodi da invocare.
Dato che un evento può essere visto come un delegato, sono consentite soltanto alcune operazioni del delegato, come l'aggiunta di un delegato all'evento (grazie all'operatore `+=`) oppure sganciarsi da un evento (mediante l'operatore `-=`).
Per ricevere notifiche relative a quell'evento occorre che il cliente definisca un metodo `EventHandler`, che verrà evocato all'atto della notifica dell'evento; dopodiché si crea un delegato dello stesso tipo dell'evento, viene riferito al metodo e successivamente lo si aggiunge alla lista dei delegati di quell'evento.
Grazie alle singole operazioni `+=` e `-=` ammesse nessuno può ottenere o modificare l'elenco dei metodi sottostante dei gestori di eventi.

Domanda 1.9
Metaprogrammazione e riflessione in C#

Risposta: La metaprogrammazione viene utilizzata per programmare un sistema in modo che abbia accesso alle informazioni relative al sistema stesso, e che possa manipolare tali informazioni. Essa viene realizzata mediante la **riflessione**, implementata in C# tramite la classe `System.Reflection`. La riflessione sfrutta i **metadati**, ovvero dati che descrivono altri dati; infatti, se un componente dispone di informazioni sufficienti per descrivere se stesso, le sue interfacce possono essere esplorate dinamicamente. In Java e .NET i dati sono generati nel momento in cui si definisce un tipo, vengono salvati assieme alla sua definizione e sono disponibili a *runtime*; in COM e CORBA i metadati sono definiti da *Interface Definition Language* (IDL).
La riflessione viene utilizzata per:

- esaminare i dettagli di un *assembly*;
- istanziare oggetti e chiamare metodi scoperti a runtime;
- creare, compilare ed eseguire assembly, ove necessario.

La chiave per la *riflessione* in .NET è la classe `System.Type`:

- tutti gli oggetti sono istanze di tipi, e i tipi stessi sono istanze di `System.Type`;
- il tipo di un oggetto può essere scoperto tramite il metodo `GetType()`;

- è presente un solo oggetto `Type` per ogni tipo definito.

Ad esempio, è possibile enumerare i tipi di un *assembly* tramite i seguenti comandi:

- `Assembly.Load(...)` restituisce un oggetto `Assembly`;
- `Assembly.GetModules(...)` restituisce un array di oggetti `Module`;
- `Module.GetTypes(...)` restitusice un array di `Type`.

È possibile creare istanze di tipi e accedere ai loro membri mediante *very late binding*:

- `Activator.CreateInstance(type, ...)` invoca il costruttore specificato;
- `methodInfo.Invoke(...)` invoca i metodi;
- `propertyInfo.GetValue(...)/.SetValue(...)` invocano *getter* e *setter*.

I membri pubblici sono sempre accessibili, mentre i membri non pubblici lo sono solo se il chiamante ha permessi sufficienti. Per creare dinamicamente istanze di oggetti si usa la classe `System.Activator`, il cui metodo `CreateInstance(Type type, Object[] args)` è equivalente all'operazione `new`.

Per aggiungere informazioni ai metadati è possibile utilizzare **attributi personalizzati**, ovvero classi visibili via riflessione che derivano da `System.Attribute` e che possono contenere proprietà e metodi.
Per creare attributi personalizzati occorre:

- dichiarare la classe dell'attributo;
- dichiarare i costruttori;
- dichiarare le proprietà;
- opzionalmente applicare `AttributeUsageAttribute`, che specifica alcune caratteristiche della classe, ovvero a quali elementi l'attributo è applicabile, quando l'attributo può essere ereditato e quando possono esistere molteplici istanze di un attributo.

Il metodo `GetCustomAttributes()` restituisce la lista degli attributi personalizzati.
La classe `System.Reflection.Emit` consente di scrivere il codice IL necessario per creare e compilare un *assembly* che potrà essere chiamato direttamente dal programma che lo ha creato, e che potrà essere archiviato su disco in modo che altri programmi possano utiizzarlo.

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 grande 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 rispettino 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.
Il principio prevede di sviluppare classi o moduli indipendenti tra loro e semplici. Un esempio di applicazione del seguente principio è il seguente:

- si ha una classe `MacchinaFotografica`:
- `MacchinaFotografica` possiede due funzionalità indipendenti tra loro, ovvero `Foto` e `Video`;
- unire le due funzionalità in un'unica classe risulta poco portabile e la modifica a una funzionalità influenza l'altra, pertanto si creano due classi separate `Fotocamera` e `Videocamera`.

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 (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 (poiché nessuno dipende da questi moduli). Dato che i dettagli del sistema sono stati isolati da un muro di astrazioni stabili:

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

Un esempio di principio di inversione delle dipendenze è il seguente:

- Si considerino le classi concrete `BigliettoNormale` e `GestoreVenditaBiglietti`; quest'ultima ha una dipendenza verso `BigliettoNormale`.
- Tale dipendenza è verso una classe concreta, pertanto il codice di `GestoreVenditaBiglietti` è poco manutenibile.
  Infatti, nel caso si volesse creare la classe `BigliettoVIP` occorrerebbe modificare il codice di `GestoreVenditaBiglietti`.
- Per risolvere il problema, si introduce l'interfaccia `IBiglietto` e fa sì che `GestoreVenditaBiglietti` dipenda da essa; inoltre, si modifica la classe `BigliettoNormale` in modo che implementi `IBiglietto`.
- A questo punto, se si volesse aggiungere una nuova tipologia di biglietto, non sarebbe più necessario modificare `GestoreVenditaBiglietti` ma basterebbe creare una classe concreta che implementi `IBiglietto` (ad esempio, `BigliettoVIP`).

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 (creatore dello standard *Ethernet*), di una stampante avente 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. Tramite l'utilizzo del principio di segregazione delle interfacce si avevano più interfacce, ciascuna specifica per il ruolo della stampante, come ad esempio le interfacce `IScanFogli` e `IPinzaFogli`.
Ciacsuna modifica di una funzione risulta più semplice, in quanto *autocontenuta*.

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, creare una nuova classe concreta che implementi le astrazioni.
Per soddisfare il principio aperto/chiuso si occorre all'ereditarietà:

- **di interfaccia**: le classi derivate ereditano da una classe base astratta con funzioni virtuali. In questo modo l'interfaccia è chiusa alle modifiche e il suo comportamento può essere modificato implementando nuove classi derivate.
- **dell'implementazione**: si creano nuove sottoclassi che estendono la classe base, mantenendo il codice comune in quest'ultima.
  In questo modo si evitano ripetizioni nelle sottoclassi.

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. Un esempio di utilizzo OCP è il seguente:

- Si supponga di avere una classe `CalcolatoreSpeseAziendali` la quale, data una collezione di oggetti di tipo `Impiegato`, restituisca le spese totali dovute al loro salario, tramite metodo `CalcolaSpese(Collection<Impiegato> impiegati)`.
- Si definisce un'interfaccia o una classe astratta `Impiegato` che espone il metodo `getSalario()`.
- Si definiscono le classi concrete `Manager`, `AmministratoreDelegato`, `Programmatore` che implementano `Impiegato` e che possiedono un salario specifico al ruolo ricoperto.
- A questo punto `CalcolaSpese(Collection<Impiegato> impiegati)` attraverserà la collezione e sommerà i salari dei vari dipendenti, senza conoscere il loro ruolo.
- Di conseguenza, in caso venga aggiunto un nuovo ruolo come `Venditore` con OCP non sarà necessario modificare il codice, ma soltanto aggiungere una nuova classe e ridefinire il metodo `getSalario()` al suo interno.

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: L'architettura dei package prevede i seguenti tre principi:

- **Principio di equivalenza di riuso/rilascio (REP)**: un elemento riutilizzabile deve essere periodicamente fornito e rilasciato da un sistema apposito, poiché chi rilascia il software non è la stessa persona che lo utilizza.
  Se non è garantita la manutenzione del software l'utilizzo di quest'ultimo non dovrebbe essere permesso ai clienti. Occorre pertanto raggruppare le classi in *package*, in modo da modo da garantire il riutilizzo di una o più classi.
- **Principio di chiusura comune (CCP)**: occorre ridurre al minimo l'utilizzo di *package*, in modo che risulti più semplice gestire e modificare ciascuna classe e ridistribuire ciascun package.
  Occorre quindi raggruppare più classi che vengono modificate insieme nello stesso *package*.
- **Principio di riutilizzo comune (CRP)**: occorre non raggruppare nello stesso package le classi che non vengono riutilizzate insieme.
  La motivazione è dovuta al fatto che, per ciascuna modifica del package, sia necessario controllare ogni singola classe per verificare il corretto funzionamento dell'aggiornamento; vengono controllate anche le classi che non presentano alcuna modifica.

Risulta complesso soddisfare tutti e tre i principi contemporaneamente, poichè il *principio di chiusura comune* suggerisce di rendere i package **i più grandi possibili**, semplificando l'utilizzo di questi ultimi da parte dello sviluppatore, mentre il *principio di equivalenza di riuso/rilascio* e il *principio di riutilizzo comune* incentivano l'utilizzo di **package piccoli**, semplificandone l'utilizzo da parte dei clienti.
Una buona prassi è applicare il *principio di chiusura comune* durante la progettazione dell'architettura, in modo da semplificare lo sviluppo e la manutenzione. Una volta ottenuta maggiore stabilità del sistema viene eseguito un *refactoring*, diminuendo i package e soddisfando i rimanenti due principi per permettere maggiore riutilizzo.

Sono presenti ulteriori tre principi che descrivono le relazioni tra i *package*:

- **Principio delle dipendenze acicliche (ADP)**: afferma che non devono essere presenti dipendenze tra package che possano creare dei cicli. Nel caso in cui si voglia modificare una classe occorre ricompilare l'intero package di appartenenza e i package che dipendono da quest'ultimo; nel caso di dipendenza ciclica una modifica di una classe potrebbe portare a ricompilare l'intero progetto.
  Per evitare questo problema occorre introdurre un'interfaccia per la classe responsabile delle dipendenze cicliche, e sfruttando il *principio di inversione delle dipendenze*, oppure trasferire le dipendenze in un nuovo package.
- **Principio delle dipendenze stabili (SDP)**: stabilisce che un package dovrebbe dipendere soltanto da package più stabili di esso.
  Un package stabile rispetta il principio di chiusura comune, poiché le modifiche a una classe non causano effetti a cascata sull'intero progetto, bensì rimangono interne al package di appartenenza.
- **Principio delle astrazioni stabili:** i package stabili dovrebbero essere astratti, poiché facili da estendere. Un package astratto non contiene codice soggetto a modifiche.

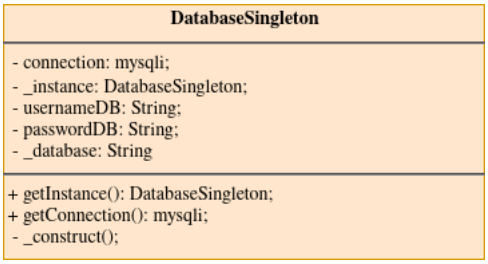
Occorre inoltre considerare i sistemi di package a livelli: i package nei livelli superiori sono instabili e facilmente modificabili, mentre nei livelli inferiori sono presenti package difficilmente modificabili. Una buona prassi consiste nel possedere package stabili **astratti**, in modo che risultino difficili da modificare, ma facili da estendere (sfruttando il principio aperto/chiuso).

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 }
16
```

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 un oggetto (detto *subject*) debba notificare un cambiamento del proprio stato interno a uno o più oggetti (detti *observer*). Si definiscono le seguenti classi:

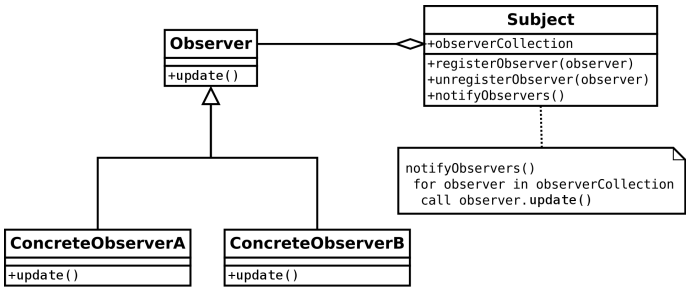
- `Subject` dichiara i metodi `Attach()` (o `Register()`), `Detach()` (o `Unregister()`) e `Notify()`.
- Gli Observer dichiarano il metodo `Update()`.

Per registrarsi presso *Subject*, un *Observer* invoca il metodo `Attach()`. Successivamente, in caso di evento, *Subject* chiama il metodo `Notify()`, che esegue `Update()` di tutti gli Observer che si sono registrati.

Infine, un `Observer` si può disiscrivere a un `Subject`, sfruttando il metodo `Detach()`.
Il pattern Observer è una delle cinque possibili soluzioni nel caso in cui si voglia aggiornare elementi in base al cambiamento del *subject*. Altri possibili quattro approcci sono:

- **Class based**: Ogni subject ha un riferimento diretto all'observer da notificare
- **Interface based**: Ogni subject possiede un riferimento all'interfaccia implementata dagli *observer*, aumentando il disaccoppiamento tra subject e observer
- **Delegate based**: Ogni subject contiene un delegato pubblico, in modo che gli observer possano registrarsi a tale delegato e venire notificati indirettamente dal subject
- **Event based**: un event si occupa di effettuare `Unregister()` degli observer.

Un esempio di utilizzo di pattern Observer potrebbe essere l'iscrizione da parte di diversi dispositivi elettronici a un sensore di temperatura; nel caso in cui la temperatura della stanza cambi, tutti i dispositivi registrati al subject `Sensore` vengono avvertiti.
Diagramma UML:



Domanda 1.18.1
Pattern Flyweight con esempi

Risposta: Il pattern *Flyweight* consente la condivisione di oggetti leggeri tra più entità, in modo tale che il loro utilizzo non sia troppo costoso. L'oggetto condiviso (`flyweight`) viene utilizzato da più client in modo efficace nello stesso momento. In particolare:

- l'oggetto non deve essere distinguibile da un oggetto non condiviso;
- l'oggetto non deve fare ipotesi sul contesto in cui opera.

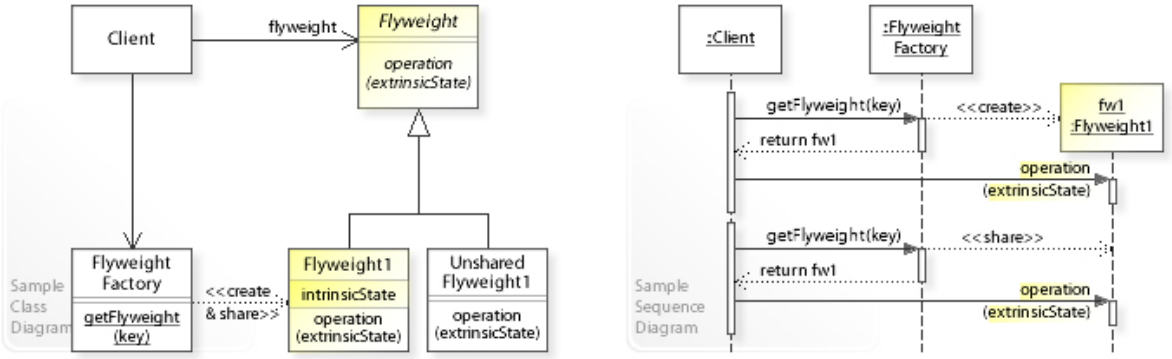
Una corretta condivisione dei *flyweight* tra i clienti viene garantita dall'utilizzo di *FlyweightFactory*. Esistono due stati associati al pattern:

- **intrinseco**: stato memorizzato in `flyweight` e non dipende dal contesto di utilizzo
- **estrinseco**: stato memorizzato dal cliente, dipende dal contesto di utilizzo e non può essere condiviso da tutti i clienti. Viene passato a `flyweight` da parte del client quando viene invocata una operazione relativa a flyweight.

Un esempio di pattern *Flyweight* potrebbe essere la rappresentazione dei caratteri in un *word processor*:

- Si consideri la classe `Carattere`. Senza pattern *flyweight* ogni carattere scritto dall'utente risulterebbe in un nuovo oggetto `Carattere`, causando un'alta occupazione di memoria.
- Applicando il pattern *flyweight* si riutilizza lo stesso carattere, senza ridefinire un nuovo oggetto a ogni sua occorrenza.
- La classe `Carattere` può contenere i campi nome, font, dimensione, che rappresentano lo stato intrinseco dell'oggetto `Flyweight`.
- Lo stato estrinseco è rappresentato dalla posizione del carattere (come riga e colonna) all'interno del testo, pertanto va gestito dal *client*.

Diagramma UML:

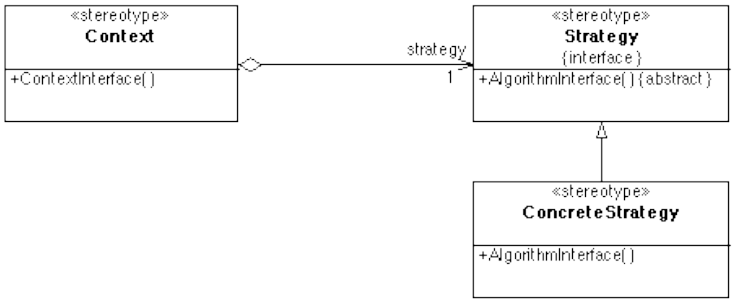


Domanda 1.19
Pattern Strategy con esempi

Risposta: Il pattern *Strategy* è un pattern di tipo comportamentale e viene utilizzato quando si ha una famiglia di algoritmi, intercambiabili tra loro, che implementano una determinata funzionalità. A questo scopo si dichiara un'interfaccia dotata di metodi che svolgono la funzionalità richiesta; si dichiara inoltre, per ogni algoritmo, una classe concreta che implementa l'interfaccia e i relativi metodi definiti precedentemente. Nel caso in cui un *cliente*, che dipende unicamente dall'interfaccia, voglia eseguire le relative funzionalità, non deve fare assunzioni su quale strategia sia stata adottata.
Un esempio di utilizzo del pattern Strategy è il seguente:

- Interfaccia `ISortStrategy` che espone il metodo `sort()`.
- Classe concreta `MergeSortStrategy` che implementa l'interfaccia `ISortStrategy` e realizza il metodo `sort()`, usando l'algorirmo di *merge sort*.
- Classe concreta `QuickSortStrategy` che implementa l'interfaccia `ISortStrategy` e realizza il metodo `sort()`, usando l'algorirmo di *quick sort*.
- Classe concreta `BubbleSortStrategy` che implementa l'interfaccia `ISortStrategy` e realizza il metodo `sort()`, usando l'algorirmo di *bubble sort*.
- Classe `Cliente` che ha un riferimento a un oggetto di tipo `ISortStrategy`.

Diagramma UML:

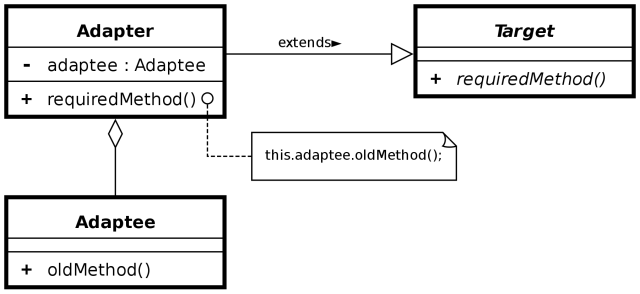


Domanda 1.20
Pattern Adapter con esempi

Risposta: Il pattern *Adapter* è un pattern di tipo strutturale. Si supponga che un Cliente dipenda da un'interfaccia Target già definita, la quale espone un certo metodo. Esiste una certa classe `Adaptee` che realizza tale metodo, ma ha un'interfaccia incompatibile con l'interfaccia `Target`. Di conseguenza, per risolvere questo problema, si utilizza una classe Adapter, che implementa Target e richiama il metodo di `Adaptee`, mascherandolo come una chiamata al metodo di `Target`. Un esempio di utilizzo del pattern Adapter è il seguente:

- Si ha un'interfaccia MediaPlayer che espone il metodo play().
- Si ha una classe MP3Player fornita da una libreria esterna, che riproduce file di tipo MP3, e che non è compatibile con MediaPlayer.
- Si ha una classe Cliente che richiede un servizio a MediaPlayer per riprodurre un file di tipo MP3.
- Si realizza una classe MediaAdapter, che implementa MediaPlayer e che mantiene un riferimento a MP3Player; in particolare quando il client einvocherà il metodo play() di MediaPlayer, verrà invocato da MediaAdapter il metodo play() di MP3Player, senza che il cliente se ne accorga.

Diagramma UML:



Domanda 1.21
Pattern Decorator con esempi

Risposta: Il pattern *Decorator* è un pattern di tipo strutturale. Consente di aggiungere dinamicamente responsabilità a un oggetto, e risulta essere una valida alternativa all'ereditarietà, qualora si abbia una gerarchia troppo estesa. I vantaggi del pattern *Decorator* rispetto all'ereditarietà sono i seguenti:

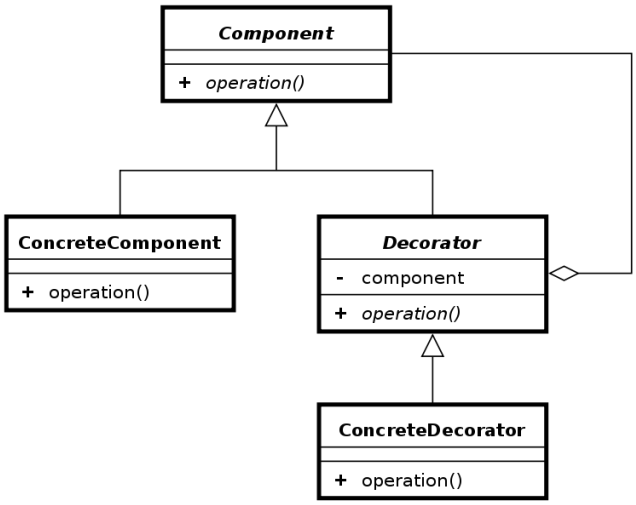
- Consente di aggiungere dinamicamente nuove funzionalità a un oggetto a *run-time*, mentre l'ereditarietà estende il comportamento delle classi padre alle classi figlie durante la fase di compilazione
- Codice pulito e più semplice da testare

In particolare viene soddisfatto il principio di singola responsabilità, in quanto il pattern *Decorator* permette di suddividere una funzionalità in più classi che svolgono uno specifico compito.
Si supponga di avere una classe `Component` che rappresenta l'interfaccia dell'oggetto a cui aggiungere dinamicamente nuove funzionalità, e la classe `ConcreteComponent` che rappresenta l'oggetto in questione. Per aggiungere la funzionalità all'oggetto si dichiara una classe astratta `Decorator`, che mantiene un riferimento all'oggetto `Component` e definisce un' interfaccia ad esso conforme, e una classe `ConcreteDecorator` che rappresenta l'oggetto che aggiunge nuove funzionalità a `ConcreteComponent` Un esempio di utilizzo del pattern *Decorator* è il seguente:

- Si ha un'interfaccia `ITea` che rappresenta `Component`

- Si hanno le classi `GreenTea`, `BlackTea` che implementano l'interfaccia `ITea`, e rappresentano i `ConcreteComponent`
- Si vuole dare la possibilità di aggiungere ulteriori ingredienti in ciascun tè; pertanto si dichiara la classe astratta `TeaExtra`, che rappresenta il `Decorator`;
- Si realizzano le classi che concretizzano la classe `TeaExtra`, come `TeaWithMilk`, `TeaWithHoney`, `TeaWithIce`

Diagramma UML:



Domanda 1.22
Pattern Composite con esempi

Risposta: Il pattern *composite* è un pattern di tipo strutturale.
Nasce per facilitare la manipolazione delle strutture ad albero, che risulta essere complessa e prona ad errori, poiché bisogna distinguere tra un nodo (oggetto composto) e una foglia (oggetto singolo). La soluzione al problema citato è la definizione di un'interfaccia che permetta di trattare allo stesso modo oggetti singoli e composti. Per realizzare il pattern *composite* si suppone di avere le seguenti componenti:

- `Component`: classe astratta che dichiara un'interfaccia che consente l'accesso e la manipolazione degli oggetti della composizione.
- `Client`: accede agli oggetti che derivano da `Component` e li manipola attraverso l'interfaccia di `Component`.
- `Leaf`: descrive il comportamento degli oggetti singoli, ovvero le foglie, e che non possono avere figli.
- `Composite`: definisce il comportamento degli oggetti aventi figli.

È necessario che il contenitore dei figli sia un attributo di `Composite`, e può essere di qualsiasi tipo (come array, lista, hashtable ecc). Il `Client` utilizza soltanto `Component`, pertanto quest'ultimo deve possedere tutti i metodi di cui il `Client` necessita e la definizione base dei metodi che dovranno essere ridefiniti dalle sottoclassi. Alcune operazioni definite in `Component` risultano prive di significato per gli oggetti senza figli, come `add`, `remove`, pertanto sono possibili due possibili approcci:

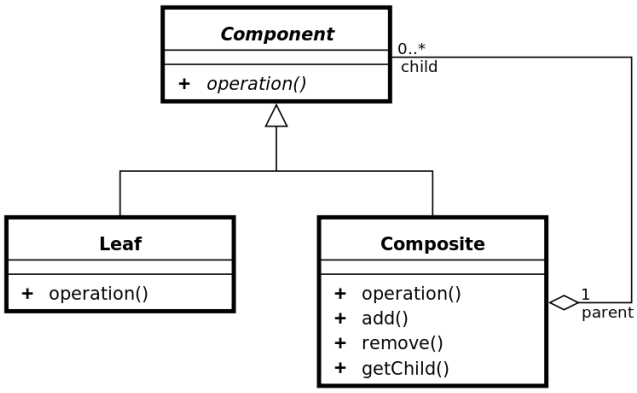
- **Trasparenza**: si definiscono le operazioni di `add` e `remove` in `Component`. Tuttavia è possibile che il `Client` definisca operazioni illegali come l'aggiunta di figli alle foglie; per evitare questo problema è necessario che le foglie sollevino eventualmente un'eccezione per i metodi precedentemente citati.

- **Sicurezza**: i metodi di gestione dei figli come `add` e `remove` vengono implementati in `Composite`; occorre predisporre di un'implementazione che verifichi se l'oggetto che si sta trattando è un `Composite` o `Leaf`.

Un esempio di utilizzo del pattern *composite* è il seguente:

- Si supponga di avere come `Client` un oggetto che deve accedere alla gerarchia di impiegati di un'azienda informatica.
- Si ha una classe astratta `Impiegato`, che rappresenta il `Component`.
- Si consideri la classe `Manager`, che rappresenta `Composite`, e che può avere come sottoposti altri impiegati.
- Si consideri la classe `Programmatore`, che rappresenta `Leaf` e che non ha sottoposti.
- Una eventuale operazione di `aggiungiSottoposto()` può essere eseguita da `Manager`, ma non da `Programmatore`.

Diagramma UML:



Domanda 1.23
Pattern Visitor con esempi

Risposta: Il pattern *Visitor* è un pattern comportamentale che separa in un'apposita classe l'operazione relativa a una struttura di oggetti.
In questo modo è possibile l'aggiunta di nuove operazioni definendo appositamente delle classi; non vengono quindi modificate le classi appartententi a una struttura che utilizzerà le nuove operazioni. Vengono soddisfatti due principi:

- il principio aperto/chiuso, in quanto l'intero sistema è aperto per aggiungere nuove funzionalità, ma risulta chiuso alle modifiche.
- il principio di singola responsabilità, poiché ogni funzionalità è in un'apposita classe.

Per realizzare il pattern *visitor* si suppone di avere le seguenti componenti:

- **Visitor**: classe astratta o un'interfaccia, che dichiara il metodo `VisitElement`.
- **ConcreteVisitor**: classe concreta che estende o implementa `Visitor` e che consente di percorrere la struttura di oggetti coinvolta.
- **Element**: classe astratta o interfaccia che dichiara il metodo `Accept`, che ha come parametro un oggetto di tipo `Visitor`.

- **ConcreteElement**: classe concreta che estende o implementa `Element`.
- **ObjectStructure**: è realizzata come *Composite* o come una collezione (array, lista, etc.) e deve contentire l'enumerazione dei suoi elementi ; per permettere di essere visitabile da `Visitor`, `ObjectStructure` deve implementare un'interfaccia apposita `Visitable`.

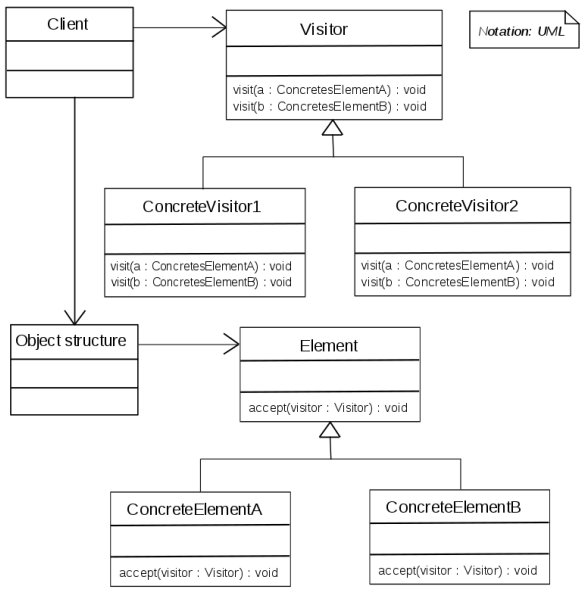
Occorre considerare i seguenti aspetti:

- Per definire una nuova operazione occorre implementare un nuovo `ConcreteVisitor`; quest'ultimo, durante un'operazione, può modificare il proprio stato.
- Per permettere a un `Visitor` l'accesso allo stato degli elementi della struttura, è necessario sfruttare l'**incapsulamento**.
- La gerarchia composta da `Element` deve essere stabile, poichè non è semplice aggiungere nuovi `ConcreteElement`: ciò implicherebbe definire in tutti i `Visitor` esistenti un metodo apposito `visit` per quel `ConcreteElement`.
- L'operazione `Accept` è di tipo *double dispatch*, poiché dipende sia da `Visitor` che da `Element`.

Si consideri il seguente esempio:

- Si suppone di avere un negozio che vende libri e CD musicali.
- Si definisce la classe `Negozio` che contiene una collezione di elementi.
- Si definisce una classe astratta `Elemento`, che presenta il metodo `accept(Visitor v)`, e le classi concrete `Libro` e `CD`, che derivano da `Elemento`.
- Si definisce l'interfaccia `Visitor` che presenta i metodi `visitLibro(Libro libro)` e `visitCD(CD cd)`.
- Si definisce la classe `VisitorVendita`, che implementa visitor e che racchiude rispettivamente la logica di vendita di libri e CD nei metodi `visitLibro(Libro libro)` e `visitCD(CD cd)`.
- Il negozio vuole consentire il prestito di libri e CD; per farlo sarà sufficiente creare una classe `VisitorPrestito`, che implementa `Visitor` e ridefinisce i metodi `visitCD` e `visitLibro`.

Diagramma UML:



Domanda 1.24
Modello LMU nei VCS con vantaggi e svantaggi

Risposta: Il modello *Lock-Modify-Unlock* (LMU) viene utilizzato in alcuni sistemi di controllo delle versioni e consente a una sola persona alla volta di modificare file di una *repository*. Ogni volta che un utente voglia modificare un determinato file della cartella di lavoro deve prima **bloccarlo** (*lock*); una volta completata la modifica occorre **sbloccare** il file. Questo modello presenta diversi svantaggi:

- Esiste la possibilità che un utente si dimentichi di sbloccare il file, non consentendo a nessun altro di modificare il file.
  Si generano quindi notevoli ritardi nello sviluppo.
- Si crea una serializzazione non necessaria, poiché la modifica contemporanea dello stesso file da parte di due persone non implica la generazione di conflitti.
- È presente un falso senso di sicurezza da parte degli sviluppatori nel blocco di un file: nel caso in cui un utente modifichi un file, e un altro utente apporti cambiamenti su un secondo file si possono generare conflitti se i due file hanno dipendenze tra loro.
  A causa dei conflitti generati i due file non funzionano più correttamente insieme.
- Il lavoro può esssere eseguito soltando offline.

Un vantaggio che il modello offre è la possibilità di lavoro nel caso in cui si hanno file non unibili, come ad esempio i file immagine

Domanda 1.25
Modello CMM nei VCS con vantaggi e svantaggi

Risposta: Il modello *Copy-Modify-Merge* viene utilizzato in alcuni sistemi di controllo delle versioni e prevede l'assenza di meccanismi di blocco dei file in caso di modifica (*lock*). Ogni client dell'utente esegue l'accesso alla *repository* del progetto e crea una copia personale e locale su cui lavorare. Gli utenti possono lavorare indipendentemente e in parallelo, modificando le loro copie private. Nel momento del *check in*, le modifiche effettuate da un utente al progetto vengono unite alle modifiche degli altri utenti: l'operazione appena descritta prende il nome di *merge*. L'operazione appena descritta può avere due esiti:

- **Successo**: le modifiche effettuate non causano problemi di congruenza del codice
- **Conflitto**: due o più utenti hanno modificato lo stesso blocco di codice, generando incongruenze.
  In questo caso occorre risolvere i conflitti a mano.

Anche in caso di conflitto il tempo di risoluzione del problema risulta minore rispetto al ritardo di sviluppo introdotto da un sistema di blocco come previsto nel modello *Lock-Modify-Unlock*. L'assenza di conflitti non garantisce sempre il corretto funzionamento del programma dopo le modifiche: l'operazione di *merge* infatti non è in grado di rilevare conflitti logici, concettuali o semantici.

Domanda 1.26
ADT ed incapsulamento

Risposta: Un **Tipo di Dato Astratto** (ADT, Abstract Data Type) è un modello matematico per strutture dati che consente di definire:

- **Cosa** fa un certo tipo di dato, senza specificare **come** lo fa.
- Le operazioni applicabili agli oggetti di quel tipo e il loro comportamento.

L'ADT è utile per favorire l'astrazione, la modularità e la riusabilità del codice.
Per definire un ADT, sono necessari:

- **Interfaccia pubblica**: specifica i metodi pubblici applicabili agli oggetti di quel tipo.

- **Implementazione privata**: include i campi privati e i metodi (sia pubblici che privati) che realizzano il comportamento definito dall'interfaccia.

Incapsulamento L'**incapsulamento**, noto anche come *information hiding*, consiste nel nascondere agli utilizzatori finali i dettagli relativi alla struttura e al funzionamento interno di un oggetto. Questo approccio consente di:

- Nascondere le scelte progettuali, rendendo il sistema più robusto e meno soggetto a errori.
- Limitare l'accesso diretto ai campi privati, garantendo che le modifiche avvengano solo attraverso metodi controllati.
- Favorire la manutenibilità del codice, poiché eventuali modifiche all'implementazione interna non influenzano il codice che utilizza l'ADT.

Un esempio pratico di incapsulamento è l'utilizzo di **getter** e **setter** per accedere e modificare i campi privati di una classe, anziché esporre direttamente tali campi.

Domanda 1.27
Differenze tra interfaccia e classe astratta

Risposta: In sintesi, un'**interfaccia** è un contratto puro, senza nessuna implementazione né stato, mentre la **classe astratta** ha una base con implementazione e stato, ma consente eredità singola. Analizzando più nel dettaglio le differenze:

Interfaccia

Descrive un contratto che una classe deve rispettare, definendo un insieme di metodi che devono essere implementati. È utilizzata per rappresentare una funzionalità semplice e specifica che può essere condivisa tra classi **eterogenee** che devono esibire comportamenti comuni.
Esempio: tutte le classi che implementano l'interfaccia `ICloneable` sono clonabili, indipendentemente dalla loro natura.

- Contiene solo **metodi astratti** (senza corpo) e **costanti**.
- Non può avere stato interno (non può contenere campi di istanza, ma solo costanti statiche e finali).
- Può contenere **metodi predefiniti** (*default methods*) con implementazione e **metodi statici**.
- Non può avere costruttori.
- Deve essere stabile, secondo il principio aperto/chiuso (OCP), poiché ogni modifica può influenzare tutte le classi che la implementano.

Per quanto riguarda le relazioni con altre classi:

- Una classe può implementare più interfacce (ereditarietà multipla di interfaccia).
- Un'interfaccia può estendere altre interfacce.
- Non può estendere una classe astratta o una classe concreta.

Classe astratta

Una classe astratta rappresenta un concetto generico o una base comune per un insieme di classi **omogenee**, con comportamenti comuni. Può fornire sia metodi astratti (da implementare nelle sottoclassi) sia metodi concreti (già implementati).
Esempio: la classe astratta `Shape` può definire metodi comuni come `getArea()` e `getPerimeter()`, lasciando alle sottoclassi come `Circle` e `Rectangle` il compito di implementare i dettagli specifici.

- Può contenere **metodi astratti** (senza corpo) e **metodi concreti** (con corpo).
- Può avere stato interno (campi di istanza) e costruttori.
- Può contenere metodi statici.
- Può fornire implementazioni predefinite per alcuni metodi, che le sottoclassi possono ridefinire.
- È aperta alle modifiche, ma ogni modifica deve essere gestita con attenzione per evitare di compromettere le sottoclassi.

Per quanto riguarda le relazioni con altre classi:

- Una classe astratta può estendere un'altra classe astratta o concreta (ma solo una, se l'ereditarietà multipla non è permessa).
- Può implementare una o più interfacce.
- Le sottoclassi concrete devono implementare tutti i metodi astratti della classe astratta.

Domanda 1.28
Ciclo di vita di un oggetto esemplificandone i passi utilizzando uno specifico linguaggio orientato agli oggetti

Risposta: Consideriamo il linguaggio **Java** e il funzionamento della **Java Virtual Machine (JVM)** per descrivere le fasi del ciclo di vita di un oggetto.

Inizio del programma

All'avvio, la JVM alloca un'area di memoria continua detta **managed heap**, destinata a contenere gli oggetti creati dinamicamente durante l'esecuzione del programma. Tale area può aumentare dinamicamente nel tempo, in base alle esigenze. Un puntatore chiamato `NextObjPtr` mantiene l'indirizzo del primo spazio libero nel managed heap.

Nascita dell'oggetto

Quando si invoca un'istruzione come `new Oggetto()`, la JVM esegue i seguenti passaggi:

1. Calcola la dimensione in memoria dell'oggetto, in base ai suoi campi e alla struttura della classe.
2. Verifica se nel managed heap c'è spazio sufficiente:
   - Se lo spazio è disponibile, si procede normalmente.
   - In caso contrario, viene attivato il **Garbage Collector (GC)** per liberare memoria.
   - Se dopo il GC lo spazio resta insufficiente, viene sollevata un'eccezione `OutOfMemoryError`.
3. Assegna `ThisObjPtr = NextObjPtr`; aggiorna `NextObjPtr = NextObjPtr + sizeof(obj)`.
4. Invoca il costruttore della classe, passando come riferimento implicito `this = ThisObjPtr`.

5. Restituisce al chiamante il riferimento alla memoria allocata per l'oggetto.

A questo punto l'oggetto è stato correttamente creato, inizializzato e pronto per essere utilizzato nel programma. Rimarrà in memoria fino a quando sarà considerato "vivo" secondo le regole di gestione della memoria.

Vita dell'oggetto

Durante l'esecuzione, l'oggetto può essere utilizzato da più componenti del programma. Finché esistono riferimenti attivi ad esso (sullo stack, nei registri o altrove), l'oggetto viene considerato raggiungibile e non può essere deallocato.

Morte dell'oggetto

Un oggetto diventa "morto" quando non è più raggiungibile da nessuna parte nel programma. In tal caso, la JVM può deallocarne la memoria (prima si utilizzava il metodo `finalize()`, ora deprecato) secondo una specifica strategia di Garbage Collection. Le principali strategie sono:

Reference Counting

Ogni oggetto mantiene un contatore che rappresenta il numero di riferimenti attivi. Quando il contatore scende a zero, l'oggetto può essere liberato. È semplice ma non gestisce correttamente i riferimenti circolari.

Tracing (Tracciamento)

È la strategia utilizzata dalla JVM. Si compone di due fasi:

1. **Marcatura**: a partire da oggetti radice (root), come quelli referenziati dallo stack, registri o riferimenti globali, il GC marca ricorsivamente tutti gli oggetti raggiungibili.
2. **Liberazione**: tutti gli oggetti non marcati vengono considerati inaccessibili. Viene invocato il loro (eventuale) metodo `finalize()` e la memoria viene resa nuovamente disponibile.

Domanda 1.29
Fattori di qualità di un buon software

Risposta: Un **software di qualità** non si limita a soddisfare i requisiti funzionali, ma rispetta anche una serie di criteri fondamentali per la progettazione, lo sviluppo e la valutazione di un sistema:

- **Correttezza**: il software deve produrre risultati coerenti con le specifiche e i requisiti funzionali. La correttezza può essere verificata tramite test, ispezioni del codice o metodi formali.
- **Robustezza**: il software deve gestire correttamente situazioni anomale o impreviste senza andare in crash o produrre comportamenti incoerenti. Include la gestione "pulita" di tutte eccezioni.
- **Affidabilità**: misura la capacità del software di mantenere un comportamento corretto e stabile nel tempo.
- **Estendibilità**: indica quanto sia semplice aggiungere nuove funzionalità al software senza modificare radicalmente la struttura esistente. Favorita da un design modulare basato su astrazioni e interfacce ben definite.
- **Riusabilità**: si riferisce alla capacità di componenti software di essere riutilizzati in contesti diversi. Un componente riusabile è autonomo, ben documentato e presenta interfacce chiare.
- **Facilità d'uso**: il software deve essere intuitivo, coerente e accessibile per l'utente finale
- **Efficienza**: riguarda l'uso ottimale delle risorse computazionali nell'hardware specifico di utilizzo.

- **Verificabilità**: rappresenta la facilità con cui si possono eseguire test sul software per accertarne il corretto funzionamento.
- **Portabilità**: misura la capacità del software di essere eseguito su piattaforme diverse con modifiche minime. Si ottiene tramite l'uso di linguaggi portabili, l'aderenza agli standard e l'astrazione delle dipendenze dalla piattaforma.

In conclusione, un buon software è il risultato di un equilibrio tra questi fattori, spesso in competizione tra loro (es. efficienza vs portabilità, robustezza vs semplicità). La consapevolezza di questi aspetti guida scelte progettuali informate e migliora l'intero ciclo di vita del software.

Domanda 1.30
Pattern Abstract Factory con esempi

Risposta: È un design pattern di tipo *creazionale*, il cui scopo è fornire un'interfaccia per creare famiglie di oggetti tra loro correlati, senza specificarne esplicitamente le classi concrete. Questo pattern consente al client di astrarsi completamente dalla logica di istanziazione, favorendo l'indipendenza rispetto alla rappresentazione concreta e alla modalità di creazione degli oggetti.

È particolarmente utile in sistemi che devono supportare più varianti o configurazioni di una stessa struttura di oggetti. In questo contesto, la "configurazione" del sistema consiste nella scelta di una **famiglia coerente di componenti**, che lavorano insieme in modo compatibile.

Un caso d'uso tipico è la creazione di componenti grafici (GUI) che devono comportarsi in modo uniforme ma adattarsi all'ambiente grafico sottostante (es. Windows, macOS, Linux). In questo scenario, si vuole che il client lavori con elementi astratti (es. `RadioButton`, `Checkbox`), senza preoccuparsi del loro stile o comportamento specifico.

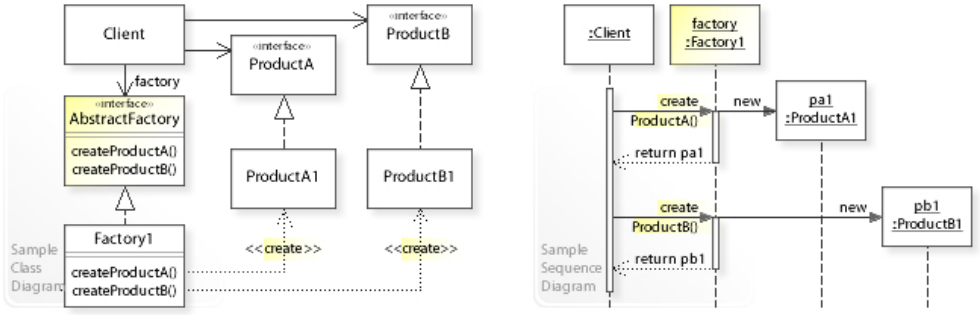
- **AbstractFactory**
  Definisce l'interfaccia per la creazione dei vari oggetti appartenenti alla famiglia. Ogni metodo corrisponde alla creazione di un componente astratto. (es. `GUIFactory`)
- **ConcreteFactory**
  Implementa l'interfaccia di `AbstractFactory`, fornendo la logica concreta per creare le varianti specifiche degli oggetti. (es. `WindowsFactory`, `LinuxFactory`).
- **AbstractProduct**
  Specifica le interfacce comuni per ciascun tipo di oggetto che può essere creato. (es. `RadioButton` con il metodo "render()")
- **ConcreteProduct**
  Sono le implementazioni concrete degli oggetti, che realizzano le interfacce definite dagli `AbstractProduct`. Ogni concrete product rappresenta una versione concreta coerente con la famiglia di riferimento (es. `LinuxRadioButton`).

  Il **Client** interagisce solo con le interfacce astratte (`AbstractFactory` e `AbstractProduct`), rimanendo del tutto indipendente dalla specifica implementazione concreta. La factory viene scelta in base al contesto (es. tipo di OS).

Diagramma UML:

Domanda 1.31
Pattern state con esempi

Risposta: È un design pattern *comportamentale* che consente a un oggetto di modificare il proprio comportamento quando cambia il suo stato interno, come se cambiasse la sua classe. Questo permette di localizzare



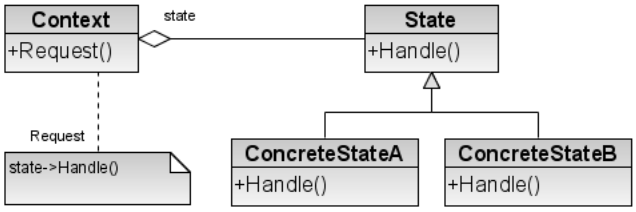
la logica specifica di ogni stato in classi dedicate, semplificando la gestione delle transizioni di stato e migliorando la manutenibilità del codice.

Il suo utilizzo è opportuno quando un oggetto deve comportarsi in modi diversi a seconda del suo stato interno, e la semplice gestione con istruzioni condizionali (if-else o switch) può diventare difficile da mantenere e estendere.

Il pattern State risolve questo problema incapsulando ogni comportamento in una classe separata e delegando l'azione allo stato corrente.

- **Context**
  Mantiene un riferimento al suo stato corrente, che rappresenta il comportamento attuale. Ad esempio, la classe `Light` rappresenta il contesto. Contiene un riferimento allo stato corrente (`state`) e delega l'azione del metodo `pressSwitch()` allo stato corrente. Inoltre, consente di cambiare stato tramite il metodo `setState()`.
- **State** (interfaccia o classe astratta)
  Definisce i metodi comuni a tutti gli stati possibili. Questi metodi rappresentano le azioni che variano a seconda dello stato. Ad esempio, l'interfaccia `State` definisce il metodo `pressSwitch(Light context)`, che rappresenta l'azione di premere l'interruttore.
- **ConcreteState**
  Implementa i comportamenti specifici per ogni stato concreto. Gestisce anche le transizioni verso altri stati, chiamando `setState()` sul contesto. Ad esempio, le classi `OffState` e `OnState` rappresentano gli stati concreti. `OffState` accende la luce e cambia lo stato a `OnState`, mentre `OnState` spegne la luce e cambia lo stato a `OffState`.

Diagramma UML:



Domanda 1.31
Pattern MVP ed MVC

Risposta: Sono due pattern architetturali per separare la logica di business dalla presentazione.

Model-View-Presenter (MVP)

Consente di separare le responsabilità di un'applicazione in tre componenti principali: il **Model**, la **View** e il **Presenter**.

Quest'ultimo gestisce direttamente la logica di interazione tra il *Model* e la *View*. La *View* è passiva e delega completamente al *Presenter*.

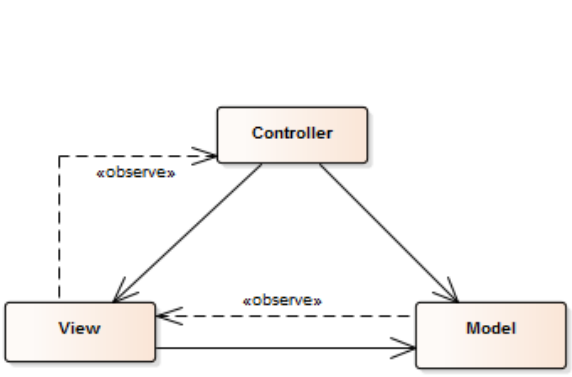
- **Model**: rappresenta i dati e la logica di business. Si occupa di:
  - Gestire un insieme di dati logicamente correlati.
  - Rispondere alle interrogazioni sui dati e alle istruzioni di modifica dello stato.
  - Generare eventi quando lo stato cambia.
  - Registrare gli oggetti interessati alla notifica degli eventi.
- **View**: è responsabile della visualizzazione dei dati e dell'interfaccia utente. Si occupa di:
  - Presentare all'utente una vista dei dati gestiti dal *Model*.
  - Mappare i dati del *Model* in oggetti virtuali e visualizzarli su un dispositivo di output.
  - Aggiornarsi automaticamente quando il *Model* subisce modifiche.
- **Presenter**: funge da intermediario tra il *Model* e la *View*. Si occupa di:
  - Ricevere gli input dell'utente dalla *View*.
  - Tradurre gli input in comandi per il *Model*.
  - Aggiornare la *View* in base ai cambiamenti del *Model*.

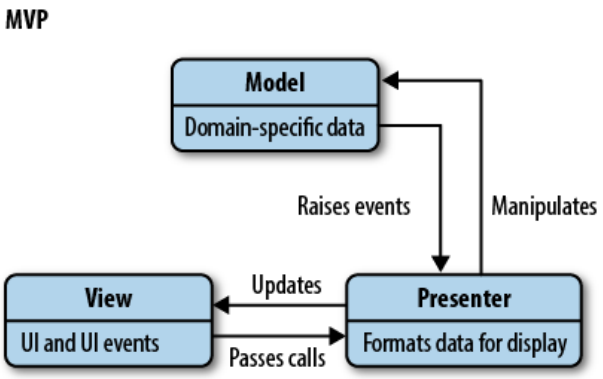
Model-View-Controller (MVC)

Molto simile al MVP, ma con una differente gestione delle responsabilità. Anche in questo caso, l'applicazione è suddivisa in tre componenti principali: il **Model**, la **View** e il **Controller**.

QUi però il *Controller* agisce come intermediario, ma la *View* può interagire direttamente con il *Model* per ottenere i dati.

- **Model**: come nel pattern MVP, rappresenta i dati e la logica di business. Si occupa di:
  - Gestire i dati e lo stato dell'applicazione.
  - Rispondere alle interrogazioni e alle modifiche dello stato.
  - Notificare i cambiamenti agli osservatori registrati.
- **View**: è responsabile della visualizzazione dei dati e dell'interfaccia utente. Si occupa di:
  - Mostrare i dati del *Model* all'utente.
  - Aggiornarsi automaticamente quando il *Model* cambia.
- **Controller**: gestisce gli input dell'utente e li traduce in comandi per il *Model* e la *View*. Si occupa di:
  - Ricevere gli input dell'utente (ad esempio, eventi di mouse o tastiera).
  - Mappare gli input in comandi per il *Model* e/o la *View*.
  - Coordinare le interazioni tra il *Model* e la *View*.





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 requisiti

Risposta: I requisiti rappresentano la descrizione dei servizi che vengono forniti dal sistema e dei suoi vincoli. Esistono due tipologie di requisiti:

- *Requisiti utente*: specificano i servizi offerti dal sistema e i vincoli su cui opera. Risultano molto astratti e di alto livello, e sono espressi solitamente in linguaggio naturale assieme a diagrammi.
- *Requisiti di sistema*: specificano nel *documento dei requisiti del sistema* le funzioni, i servizi e i vincoli del sistema in modo dettagliato.
Questi ultimi sono ulteriormente diviso in tre categorie:
  - **Requisiti funzionali**: elenca i servizi che il sistema deve fornire, in particolare per ciascun servizio viene specificato come reagire a particolari input, come comportarsi in specifiche situazioni, e cosa il sistema non dovrebbe fare.
  - **Requisiti non funzionali**: descrivono altre caratteristiche del sistema, tra le quali:
    * *Requisiti del prodotto*: limitano le proprietà del sistema.
    * *Requisiti organizzativi*: vincolano il processo di sviluppo.
    * *Requisiti esterni*: sono vincoli che derivano da sistemi esterni o da contesti come la legislazione sulla privacy o requisiti etici.

Altre caratteristiche che riguardano i requisiti non funzionali è la difficoltà di verifica di questi ultimi, poiché risultano poco chiari e vaghi. Spesso risultano in contraddizione con i requisiti funzionali: un esempio è la protezione della privacy che può essere in contrasto con la facilità d'uso o con i tempi di risposta di un sistema.

– **Requisiti del dominio**: derivano dal dominio di applicazione del sistema e indicano il funzionamento del sistema all'interno di uno specifico dominio.
Il dominio di applicazione del sistema viene specificato agli esperti del dominio.

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 e limitano il suo utilizzo.

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
Linee guida di progettazione nella sicurezza

Risposta: Le linee guida per la sicurezza nella progettazione sono le seguenti:

1. Basarsi su una politica esplicita per le decisioni inerenti la sicurezza.
   Le politiche di sicurezza vengono esplicitate in documenti di alto livello, che definiscono quali siano i criteri della sicurezza, ma non specificano come ottenerla. Occoore incorporare politiche di sicurezza in modo da specificare quali sono le informazioni che possono essere accedute e quale ente può accedervi, e quali precondizioni occorre soddisfare per l'accesso.
2. Evitare ogni singolo punto di fallimento, poiché il fallimento di una parte del sistema può compromettere l'intero sistema.
3. Fallire in modo controllato.
4. Bilanciare sicurezza e usabilità del sistema, dato che l'aggiunta di ulteriori livelli di sicurezza possono richiedere tempo maggiore di apprendimento dell'uso del sistema da parte dell'utente.
5. Essere consapevoli delle conseguenze che l'ingegneria sociale può portare, poiché tramite quest'ultima è semplice ottenere alcuni dati personali dell'utente, e convincerlo a rivelare ulteriori dati sensibili.
6. Utilizzare per ridurre rischi i seguenti fattori:
   - **Ridondanza**: mantenimento di più versioni dello stesso software e di dati nel sistema.
   - **Diversità**: consiste nell'utilizzo di tecnologie diverse per le varie versioni del sistema, in modo che una vulnerabilità di una tecnologia non consenta un punto di fallimento comune.
7. Validare l'input, ovvero applicare controlli ai dati di ingresso, in modo da non potere sfruttare vulnerabilità come *SQL Injection*, *Buffer Overflow*, etc.
8. Organizzare le informazioni di sistema in compartimenti, in modo che gli utenti possano accedere alle informazioni di loro competenza.
9. Progettare per il *deployment*, includendo nel sistema programmi di utilità che semplifichino il processo di *deployment*.
   Alcune indicazioni sono elencate in seguito:
   - includere supporto per la visione e analisi delle configurazioni;
   - ridurre al minimo i privilegi di default;
   - rendere le impostazioni di configurazione locali;
   - fornire metodi per rimediare a vulnerabilità.
10. Progettare per il ripristino, includendo meccanismi diretti o automatici per aggiornare il sistema e risolvere le vulnerabilità scoperte.

Domanda 2.8
White box e black box testing

Risposta: Il *Black box testing* consente di trovare le vulnerabilità di un sistema, senza sapere come esso è stato implementato. Gli aspetti che vengono migliorati principalmente sono la velocità del sistema, l'affidabilità, la velocità. I tester non possiedono il codice sorgente, ma cercano di intuire la struttura del sistema, per poi attaccarlo in modo mirato.
Sfruttano molte vulnerabilità inerenti ai linguaggi di programmazione utilizzati, alle configurazioni delle reti, degli host e delle maccchine virtuali.
Il *White box testing* prevede la conoscenza completa dell'applicazione da parte dei tester, dalle informazioni di configurazione delle reti e delle macchine virtuali fino al codice sorgente. In questo modo i tester possono effettuare una revisione del codice, e la creazione di test *ad hoc* per il sistema, traendo vantaggio dalle debolezze scoperte. Questo approccio consente di valutare, in primo luogo, la leggibilità e la modularità del codice.

Domanda 2.9
Capacità di sopravvivenza del sistema

Risposta: La capacità di sopravvivenza del sistema indica la capacità di effettuare servizi agli utenti che ne hanno il permesso, qualora il sistema sia sotto attacco o nel caso in cui presenti componenti danneggiate. Essa riguarda l'intero sistema e non le singole componenti, e risulta importante in quanto sia l'economia che la società dipendono da servizi digitali. Nel processo di ingegnerizzazione dei processi sicuri occorre tenere conto della capacità di sopravvivenza, in quanto esistono servizi critici, soggetti ad attacchi. Occorre quindi conoscere:

- quali servizi risultano essere critici;
- in quali modi i servizi critici possono essere attaccati;
- lo standard di qualità minimo da mantenere nei servizi;
- come proteggere i servizi, qualora siano sotto attacco;
- come ripristinare il sistema nel tempo minore possibile, nel caso in cui i servizi siano sotto attacco.

Per potere ideare un sistema che supporti la capacità di sopravvivenza e per valutare le vulnerabilità di quest'ultimo è stato ideato il *Survivable Analysis System*. Questo metodo struttura la sopravvivenza di un sistema in un processo a quattro fasi e dipende dalle seguenti strategie complementari:

- **Identificazione**: permette di individuare problemi grazie a un sistema che riconosce eventuali attacchi e fallimenti, valutandone il danno.
- **Resistenza**: permette al sistema di respingere attacchi.
- **Ripristino**: consente, nonostante i problemi del sistema, di garantire il funzionamento dei componenti essenziali, e di ripristinare tutti i servizi dopo un attacco.

Le fasi sono:

1. **Capire il sistema**: esaminare l'architettura del sistema, i requisiti e gli obiettivi.
2. **Identificare i servizi critici**: capire quali servizi sono critici, e quali sono le componenti che li gestiscono.
3. **Simulare attacchi**: individuare i casi d'uso e gli scenari di eventuali attacchi e i componenti soggetti agli attacchi.
4. **Analizzare la sopravvivenza**: identificare i componenti essenziali e vulnerabili, e sfruttare strategie di sopravvivenza come l'identificazione, la resistenza e il ripristino.

Sfortunatamente l'analisi della sopravvivenza non viene effettuata nella maggior parte dei processi di ingegnerizzazione, in quanto molte aziende che non hanno subito attacchi risultano scettiche nell'investire sulla sicurezza. È tuttavia consigliato quest'ultimo investimento, prevenendo eventuali attacchi, piuttosto che subirli, in quanto le perdite conseguenti risulterebbero gravi in termini di risorse e, nei casi peggiori, di vite.