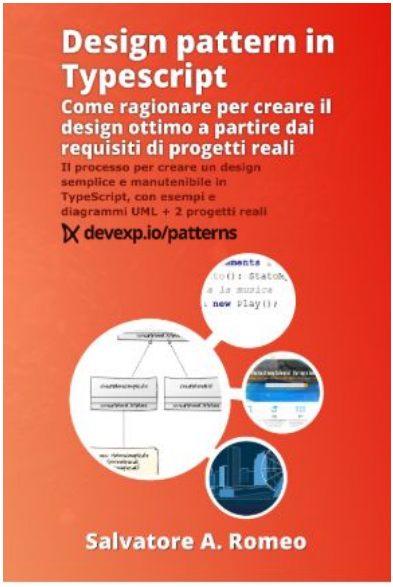
Cos'è la Dependency Injection e come può migliorare il design di un progetto + codici ed esempi



C'è una differenza sostanziale tra un semplice programmatore e un ingegnere del software. E non si tratta di avere un titolo, ma del modo di progettare un software. La Dependency Injection è la base per ottenere un design semplice, manutenibile e, quindi, elegante.

Autore

Ciao, sono Salvatore e trovi tutto su di me sul sito https://linkedin.com/in/salvatoreromeo. Ho scritto questo libro principalmente perché mi piace scrivere e condividere esperienza.

Nella vita mi occupo di sviluppo software fin da quando avevo 12 anni. Oggi è il mio lavoro principale. Insegno Ingegneria del Software all'Università degli Studi di Perugia e un corso di 3 giorni su Angular per le aziende insieme al Codemotion. Il corso è da poco stato trasformato in un libro che puoi trovare su Amazon.

Se vuoi maggiori informazioni su di me contattami pure tramite linkedin o tramite il sito https://devexp.io. Lì potrai anche trovare articoli sul front-end, altri libri del gruppo devexp.io e i codici sorgenti pubblicati su Github.

Sommario

Introduzione

In questo articolo spiegheremo in maniera completa cos'è la Dependency Injection.
Partiremo da un progetto scritto con un codice semplice e itereremo più volte le versioni del codice per arrivare ad ottenere un design elegante, semplice e molto manutenibile. Esploreremo le interfacce e il polimorfismo e vedremo come con questi ingredienti possiamo applicare il pattern Dependency Injection.

Questo articolo è estratto da un capitolo del libro **Design Pattern in Typescript** (scritto da me :-)), disponibile su Amazon all'indirizzo
https://www.amazon.it/Design-pattern-Typescript-ragionare-manutenibile/dp/1719800375
Una descrizione del testo e il motivo per cui sono importanti i Design Pattern è disponibile sul sito qui:
https://devexp.io/blog/design-patterns-typescript/

Comprendere l'utilità delle interfacce e del polimorfismo

Poiché l'obiettivo principale è rendere il codice manutenibile, dobbiamo capire

1. Cosa significa **manutenibile**;
2. Quali sono gli **strumenti** che ci permettono di ottenere questo risultato.

Rendere un software **manutenibile** significa che quando dobbiamo aggiungere funzionalità o modificare quelle esistenti, il *costo* per farlo è basso.

Chiariamo meglio quanto detto: per *costo* si intende il numero di ore di lavoro, detto anche *ore/uomo* o *giorni/uomo*. Se il costo deve essere basso, l'ideale sarebbe lavorare esclusivamente sulla modifica o sulla nuova funzionalità, senza modificare il codice esistente. Se infatti dobbiamo modificare anche il codice esistente, allora sarà maggiore il lavoro da fare, perché magari dovremo adattare del codice alla nuova funzionalità. A volte è necessario, ma spesso dobbiamo farlo solo perché la progettazione iniziale non è stata ottimale.

Quali sono allora gli strumenti che ci permettono di creare del codice in cui la modifica o l'aggiunta di funzionalità ci permette di non intaccare più, per quanto possibile, il codice esistente?

Arriviamo quindi a quali sono gli **strumenti** che ci aiutano ad ottenere il nostro obiettivo:

- le interfacce;
- il polimorfismo.

Ormai tutti i linguaggi di programmazione moderni permettono di definire interfacce e classi perché la programmazione orientata agli oggetti offre numerosi vantaggi. TypeScript non è da meno.

Nella prossima sezione vedremo quindi come utilizzare le interfacce per rendere un codice manutenibile. Introdurremo inoltre la

Dependency Injection, un concetto che sta alla base di tutti i buoni design e che riprenderemo più volte durante tutto il testo.

Interfacce e polimorfismo

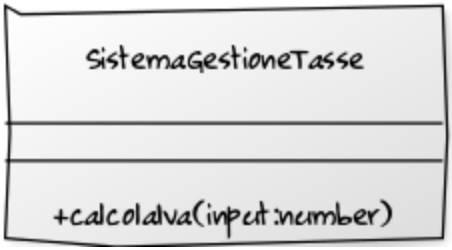
Per comprendere il concetto di interfacce e polimorfismo partiamo da un esempio di codice[1].
Supponiamo di avere un sistema di gestione tasse che esegue varie operazioni. Tra queste c'è un metodo che serve per calcolare l'IVA:

```
class SistemaGestioneTasse {

    //... altre operazioni non visualizzate

    calcolaIva(input:number){
        return input * 0.2
    }

}
```

In questo metodo le istruzioni per il calcolo effettivo sono semplici, ma un sistema che si occupa di gestire le tasse è sicuramente molto complesso. Immaginiamo allora che ci siano altri metodi con istruzioni molto più complicate.

[1] Ricordo che i codici sono scritti in linguaggio TypeScript e che alla fine del testo c'è un'appendice per apprendere la sintassi. Se si hanno le basi di programmazione in altri linguaggi gli esempi saranno comunque chiari.
Due note rapide per non essere spiazzati: rispetto a JAVA o C# il tipo si dichiara dopo un parametro, separato con i due punti; le classi e metodi non hanno bisogno di qualificatori - sono *public*.

Graficamente indicheremo una classe come un rettangolo con tre sezioni (una per il nome, una per i campi di classe, una per i metodi)[2]:



Concentriamoci per ora sul metodo *calcolaIva*. Ad un primo sguardo potrebbe sembrare tutto corretto. Effettivamente a livello sintattico non c'è nessun problema. C'è però un problema a livello di design. Cerchiamo di capire quale.

Un concetto molto importante nel design del codice è cercare di **scrivere in modo che sia possibile estendere il sistema senza modificare le classi esistenti**.

Se però in futuro il tasso IVA varia, dobbiamo necessariamente intervenire modificando il codice esistente. Oggi ad esempio il tasso è allo 0.22%.

La nostra classe diventerebbe:

```
class SistemaGestioneTasse {

   //... altre operazioni non visualizzate

   calcolaIva(input:number){
       return input * 0.22
   }
```

[2] I diagrammi che presenteremo, tranne che per piccole differenze, si chiamano diagrammi UML delle classi. Esiste una sintassi molto formale sul disegno di tali diagrammi, ma in questo testo abbiamo spesso rilassato la sintassi per concentrarci piuttosto sulla comprensione del design.

```
}
```

Abbiamo necessariamente dovuto cambiare il codice esistente.

Potevamo trovare una soluzione che ci avesse permesso di intervenire senza modificare il codice esistente?
Si, con l'uso delle interfacce.

Nei linguaggi di programmazione ad oggetti esiste il concetto di interfaccia e di classe che implementa un'interfaccia. Senza scendere nel dettaglio, l'interfaccia **dichiara cosa** fa una classe, mentre la classe **implementa** i metodi dichiarati, cioè **come** svolgere una determinata funzione. Un'interfaccia può essere implementata da diverse classi, quindi se il **cosa** resta sempre uguale e definito nell'interfaccia, il **come** può variare da classe a classe.
In che modo potremmo allora riscrivere il codice precedente per far si che se in futuro il tasso IVA cambi nuovamente non dovremmo modificare la classe *SistemaGestioneTasse*? (che vi ricordo potrebbe essere molto lunga e complicata)

Possiamo fare in modo che il calcolo effettivo dell'IVA non dipendi più dal *SistemaGestioneTasse*, ma sia delegato ad un'altra entità, definita da un'interfaccia: *CalcoloIva*. Alla classe *SistemaGestioneTasse* non interessa sapere come verrà calcolata l'IVA, ma la cosa che gli interessa è che usando un oggetto di tipo *CalcoloIva* otterrà il valore che gli serve:

```
class SistemaGestioneTasse {

   //... altre operazioni non visualizzate

   calcolaIva(input: number,
              calcolatoreIva: CalcoloIva) {
```

```
        return calcolatoreIva.calcolaIva(input)
    }

}
```

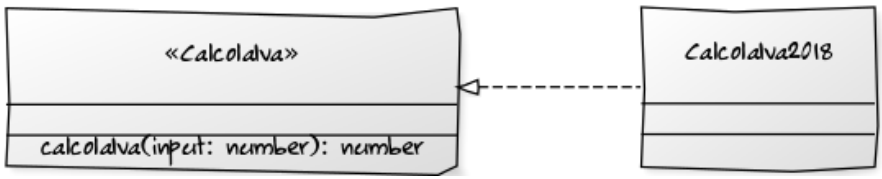
l'interfaccia è molto semplice:

```
interface CalcoloIva {
    calcolaIva(input: number): number
}
```

la classe che la implementa è la seguente:

```
class CalcolaIva2018 implements CalcoloIva {
    calcolaIva(input: number) {
        return input * 0.22
    }
}
```

Graficamente rappresenteremo un'interfaccia come una classe, ma tra doppie parentesi angolari, mentre una classe che implementa un'interfaccia con una freccia tratteggiata la cui punta è bianca:



Come ci aiuta a livello di design questa soluzione che apparentemente sembra più complicata della soluzione precedente?
Effettivamente abbiamo aggiunto qualche riga di codice. Ma c'è un'enorme differenza con la soluzione precedente: supponiamo che in futuro sarà nuovamente necessario modificare il codice perché il calcolo effettivo dell'IVA avrà un'implementazione diversa. Per ottenere questo risultato non dovremo più modificare le classi *CalcoloIva* e *SistemaGestioneTasse*, ma basterà aggiungere una nuova classe *CalcoloIva2019*, con le sue specifiche istruzioni.

Non dovendo intervenire sul codice esistente avremo diversi vantaggi:

1. Il costo di manutenzione è minore poiché dobbiamo solo sapere come funziona l'interfaccia; questo vale per lo sviluppatore terzo[3], che non ha bisogno di sapere come funziona tutto il sistema, ma solo qual è l'input e come restituire l'output;
2. Per aggiungere la nuova funzionalità ci siamo concentrati praticamente solo sul codice inerente quella funzionalità (la classe *CalcoloIva2019*);
3. Un sistema di questo tipo è anche più ordinato e per testare questa specifica funzionalità basterà testare la nuova classe senza considerare il contesto in cui verrà usata.

Usando le interfacce abbiamo quindi permesso allo sviluppatore che dovrà intervenire in futuro sul sistema di aggiornarlo e migliorarlo senza intaccare gran parte del codice esistente.

Nella prossima sezione parleremo della Dependency Injection e vedremo come le interfacce e il polimorfismo aiutino nel configurare sistemi anche molto complessi, estendendoli e modificandoli aggiornando solo pochi punti del codice esistente.

[3] Per sviluppatore terzo si intende colui che ancora non conosce il codice.

Dependency Injection

Per comprendere la dependency injection, cominciamo con qualche esempio.
Leggiamo il seguente requisito:

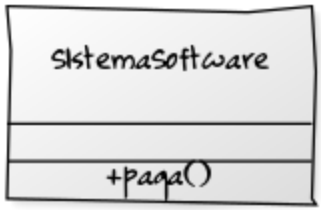
Il sistema deve poter effettuare pagamenti con Paypal

Sembra una funzionalità abbastanza complicata nell'implementazione e per esperienza posso confermare che quando si tratta di pagamenti bisogna fare le cose molto bene: il ***cliente*** non ammette la perdita di denaro se il nostro motore di pagamento non funziona.
Il ***cliente*** è un'entità che impareremo a conoscere nel testo. Si tratta di colui che ha commissionato il software e che vuole ottenere un obiettivo con il software che andremo a sviluppare. Con i ***requisiti*** cercheremo di formalizzare in forma scritta quello che il cliente ha in mente.

Come possiamo tradurre il requisito sopra in codice?
Vediamo una prima implementazione:

```
class SistemaSoftware {

    // ... altri metodi che svolgono le varie
funzionalità del software

   paga(){

       // istruzioni per il pagamento con Paypal
       // immaginiamo circa 400 righe di codice
```

```
    }
}
```



La soluzione proposta sopra è stata implementata inserendo le istruzioni che si occupano del pagamento direttamente dentro la classe principale del sistema.

Che problemi ha questa soluzione? Finché il sistema continuerà ad usare Paypal non ci saranno grossi problemi, se non per il fatto che per ogni bug fix[4] dobbiamo intervenire sempre sulla classe *SistemaSoftware*.

Supponiamo ora che il cliente ci richieda una nuova funzionalità, che traduciamo nel seguente requisito:

Il sistema deve poter effettuare pagamenti con carta di credito

A questo punto le cose si complicano. Come procediamo? Cancelliamo le istruzioni precedenti e implementiamo istruzioni nuove per pagare con carta di credito? E se un domani il cliente volesse mantenere entrambi i metodi di pagamento?
Dobbiamo evidentemente trovare un'altra soluzione.

[4] Un bug è un problema che compromette il corretto funzionamento del software. Un bug fix consiste nella modifica delle istruzioni software per risolvere il problema.

Cos'hanno in comune i due metodi di pagamento? Hanno oggettivamente una cosa in comune tra tutte: a livello astratto sono metodi di pagamento. Un metodo di pagamento deve permettere di pagare.

Nella sezione precedente abbiamo visto che esiste un concetto nei linguaggi ad oggetti per definire in maniera astratta cosa farà un componente e poi le implementazioni ci forniranno la soluzione di come verrà svolta la funzione del componente: le interfacce e le classi.
Un'interfaccia che esprima il concetto di pagamento visto sopra potrebbe essere la seguente:

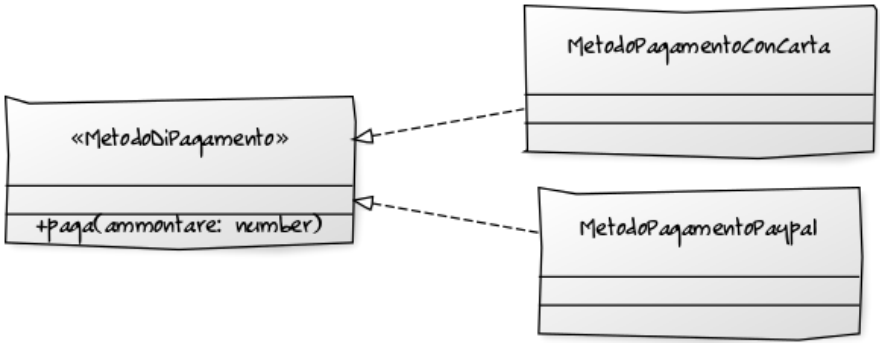
```
interface MetodoDiPagamento {
    paga(ammontare:number)
}
```

Le classi concrete relative ai nostri requisiti potrebbero essere le seguenti:

```
class MetodoPagamentoPaypal
                implements MetodoDiPagamento {
    paga(ammontare: number) {
        // istruzioni per il pagamento con Paypal
    }
}

class MetodoPagamentoConCarta
                implements MetodoDiPagamento {
    paga(ammontare: number) {
        // istruzioni per il pagamento con carta
```

```
    }
}
```

Questa la rappresentazione grafica:

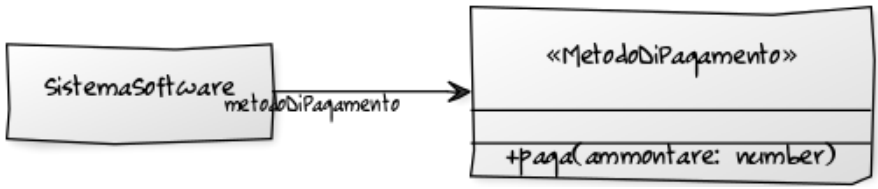


Nella realtà potrebbero essere necessarie molte altre informazioni per un pagamento, ma ai fini didattici supponiamo che sia sufficiente conoscere l'ammontare del pagamento.

Con la soluzione appena vista, come potremmo modificare il nostro *SistemaSoftware* per gestire i pagamenti?
Possiamo spostare la gestione dei pagamenti al di fuori della classe *SistemaSoftware*:

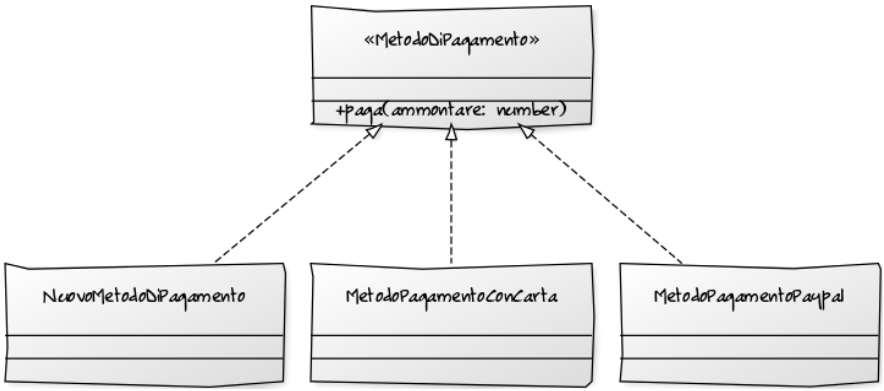
```
class SistemaSoftwareV2 {

   private metodoDiPagamento: MetodoDiPagamento

   constructor(
       metodoDiPagamento: MetodoDiPagamento) {
                       this.metodoDiPagamento    =
metodoDiPagamento
```

```
    }

    paga(ammontare: number) {
        this.metodoDiPagamento.paga(ammontare)
    }


}
```



In questa seconda versione abbiamo aggiunto nel costruttore un parametro *MetodoDiPagamento*, che abbiamo memorizzato nella classe, per usarlo nel metodo paga. In sostanza ora la classe *SistemaSoftware* può pagare usando diversi metodi di pagamento. La classe *SistemaSoftware* non sa neanche quale sarà l'implementazione effettiva, ma sa come pagare quando riceve in input un ammontare che rappresenta il prezzo da pagare.
Questa versione, rispetto alla precedente, ha diversi vantaggi:

- quando dobbiamo cambiare metodo di pagamento, sarà sufficiente fornire una nuova implementazione in input al *SistemaSoftwareV2*. Il codice è ora più **manutenibile**: delle classi *SistemaSoftware, MetodoDiPagamento, MetodoPagamentoPaypal, MetodoPagamentoConCarta*, quale dovremmo modificare per aggiungere un nuovo metodo di pagamento? **Nessuna!**



- la classe *SistemaSoftware* risulta molto più **semplice** da leggere perché tutte le righe di codice che si occupano del pagamento sono state spostate in un componente a se stante. Lo sviluppatore terzo che dovrà studiare il codice per modificarlo non farà fatica a leggere e comprendere le parti che costituiscono il sistema poiché la classe principale è costituita da poche righe, che rimandano le funzioni specifiche a componenti specifici.

L'operazione che abbiamo fatto, che ha migliorato il design del sistema, tecnicamente si chiama operazione di **refactoring**. Un'operazione di refactoring è quindi una modifica al codice che migliora il design.

Siamo ora pronti a parlare di *Dependency Injection*.

Nell'esempio appena visto, *SistemaSoftware* non dipende direttamente dal *MetodoPagamentoPaypal*. *SistemaSoftware* dipende da un generico metodo di pagamento, ma non sappiamo quale; sappiamo cosa dovrà fare perché è scritto nell'interfaccia.
Inizialmente potremmo adottare questa soluzione:

```
class SistemaSoftwareV2Beta {

    paga(ammontare: number) {
                                                    new
MetodoPagamentoPaypal().paga(ammontare)
    }
}
```

Ma per quanto questa versione permette di diminuire il codice nella classe *SistemaSoftware*, rispetto alla versione precedente ha un problema: per aggiornare il metodo di pagamento dobbiamo ancora modificare una classe esistente, mentre con la soluzione precedente, in cui il metodo di pagamento viene specificato nel costruttore, non serve modificare la classe esistente. La versione precedente fornisce il metodo di pagamento concreto senza che *SistemaSoftware* dipenda direttamente da un'istanza specifica.

Ogni volta che la dipendenza concreta viene fornita nel costruttore specificando come parametro l'interfaccia e non la classe concreta si parla di Dependency Injection.

Con la *Dependency Injection*, in generale, permettiamo ad una classe di non dipendere direttamente da una eventuale implementazione. Sfruttando le interfacce e il polimorfismo possiamo far interagire le due classi in maniera che il collegamento sia fatto tramite un'interfaccia, facendo quindi sì che il design sia semplice e facilmente manutenibile per l'evoluzione del sistema. Tecnicamente si dice che la classe e la dipendenza sono *debolmente accoppiate*, mentre nella classe *SistemaSoftwareV2Beta* sopra, tale classe è *fortemente accoppiata* con

MetodoPagamentoPaypal perché lo istanzia direttamente nel metodo, cosa che in generale vogliamo evitare per tutte le ragioni discusse.

La Dependency Injection è un pattern nel quale oggetti complessi non istanziano altri oggetti all'interno dei loro metodi o nel proprio costruttore: le dipendenze saranno fornite dall'esterno tramite il costruttore.

Vediamo ora un altro esempio per comprendere meglio tale concetto.

Partiamo da questo requisito:

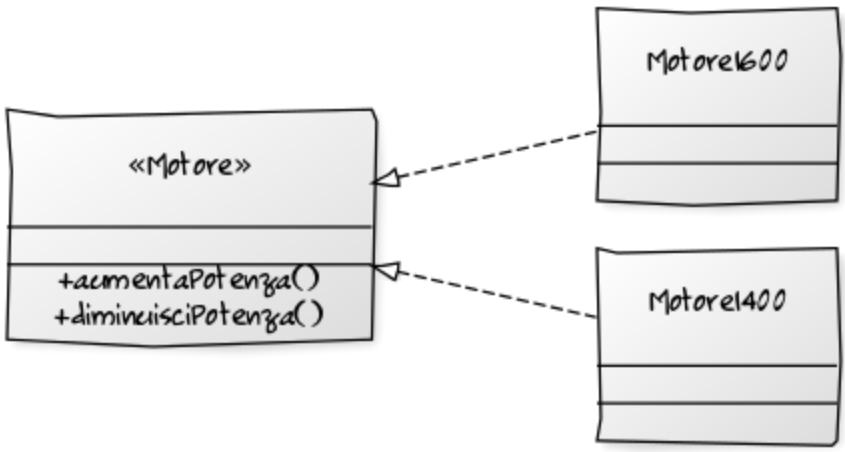
Progettare un'officina che permetta di costruire automobili con diversi motori.

Per implementare questo requisito supponiamo di avere una classe *Automobile* che ha bisogno di un motore per funzionare.
Il motore può avere diverse implementazioni: *Motore1400*, *Motore1600*, ecc.
Come potremmo implementare il tutto usando il design pattern Dependency Injection appena visto?

Iniziamo dall'interfaccia *Motore* e dai motori concreti:

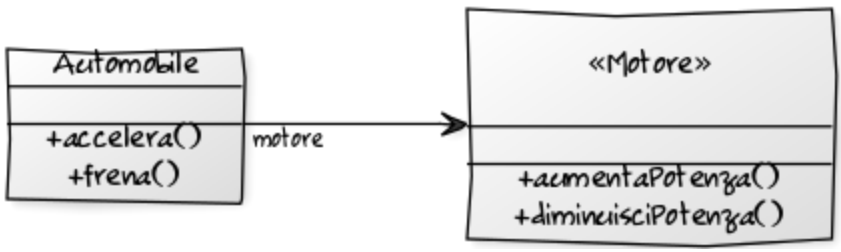
```
interface Motore {
    aumentaPotenza():number
    diminuisciPotenza():number
}

class Motore1400 implements Motore {
    aumentaPotenza(): number {
        return 10;
    }
```

```
    diminuisciPotenza(): number {
        return -10;
    }
}


class Motore1600 implements Motore {
    aumentaPotenza(): number {
        return 20;
    }

    diminuisciPotenza(): number {
        return -20;
    }
}
```



La classe *Automobile*, che userà il motore, non sarà accoppiata ad un motore specifico, ma all'interfaccia *Motore* generica. In questo modo possiamo creare automobili con differenti motori.

```
class Automobile {
   private velocita: number;

   constructor(private motore:Motore){}

   accelera(){

    this.velocita =
          Math.min(this.motore.aumentaPotenza(),
360)
   }
   frena(){
    this.velocita =
                                          Math.max(0,
this.motore.diminuisciPotenza())
   }
}
```



Alcune note relative al codice della classe *Automobile*:

1. In Typescript se aggiungo il qualificatore private nel parametro del costruttore sto automaticamente definendo il relativo campo di classe che assume anche il valore del parametro (in JAVA invece devo sempre definire parametro e

campo separatamente e aggiungere nel costruttore l'istruzione di assegnazione);

2. Uso le API *Math.min* e *Math.max* per limitare la velocità dell'automobile tra 0 e 360. Riusciamo a immaginare come potremmo rifattorizzare il sistema per spostare questa gestione al di fuori della classe *Automobile*?

Resta da capire un'ultima cosa: quale classe si occupa di configurare le istanze della classe *Automobile*? Potrebbe essere il nostro *main*, o anche un'altra classe:

```
class Officina {

    macchina1400(){
        return new Automobile(new Motore1400())
    }

    macchina1600(){
        return new Automobile(new Motore1600())
    }

}
```

La classe che si occupa di specificare l'implementazione concreta si chiama ***injector***. L'injector **configura** quindi le classi per prepararle al loro corretto funzionamento.
L'unica cosa che cambia in un buon design quando evolviamo un sistema è la **configurazione** delle classi.

Cosa imparerai con l'esperienza

Mi capita a volte di trovare nel codice dei miei studenti un errore molto comune. Supponiamo di avere una classe con una dipendenza.

```
eseguiOperazione() {
    this.dipendenza.applicaOperazione()
}
```

La dipendenza non è configurata nel costruttore, ma con un metodo *setDipendenza(dipendenza)* :

```
setDipendenza(dipendenza) {
    this.dipendenza = dipendenza
}
```

Anche se sintatticamente è possibile specificare la dipendenza tramite un metodo e non nel costruttore, questo approccio ha uno svantaggio importante: prima di eseguire il metodo *eseguiOperazione*, devo necessariamente aver eseguito almeno una volta il metodo *setDipendenza*, altrimenti incorrerei in una eccezione perché il campo *dipendenza* non è ancora stato istanziato. Spostando la injection della dipendenza nel costruttore non avrò mai questo problema poiché per usare la classe devo necessariamente passare il parametro al costruttore. Il mio consiglio è quindi quello di **configurare le dipendenze sempre tramite costruttore**.

Diverso il discorso di passare la dipendenza come ulteriore parametro di un metodo ed utilizzarla solo in quello specifico metodo. Una soluzione di questo tipo:

```
eseguiOperazione(dipendenza, parametriDiInput)
```

potrebbe avere senso se la dipendenza è usata solo ed esclusivamente in quel metodo di classe e in nessun altro. Se fosse usata in altri metodi della stessa classe probabilmente è opportuno esplicitare la dipendenza nel costruttore.

Se vuoi approfondire lo sviluppo del codice e i design pattern puoi fare riferimento al testo su Amazon:

https://www.amazon.it/Design-pattern-Typescript-ragionare-manutenibile/dp/1719800375