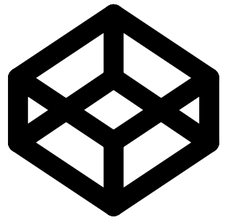
Progetto typescript di Andrea Altomare

[codepen]



[GitHub]

1. Uso delle Interfacce

Allora, ho usato le interfacce per definire la "struttura" delle cose nel sistema, come i partecipanti, i corsi e le aziende. In pratica, le interfacce mi permettono di dire: "Ehi, qualsiasi cosa che voglia essere un partecipante DEVE avere un nome, un cognome, un livello di istruzione e così via". Questo aiuta a tenere tutto più chiaro e ordinato, senza che qualcuno possa creare un oggetto che non rispetta il formato che mi aspetto.
Ad esempio, per la classe Partecipante ho scritto questa interfaccia:

```
interface IPartecipante {
nome: string;
cognome: string;
paeseDiOrigine: string;
livelloDiIstruzione: string;
competenzeLinguistiche: string[];
ambitoDiFormazioneInteresse: string;
iscrivitiCorso(corso: ICorso): void}
```

Qui definisco che un partecipante ha delle proprietà specifiche e un comportamento: iscriversi a un corso. Se qualcun altro prova a scrivere un partecipante senza una di queste proprietà, TypeScript ti dà un errore e ti fa capire che qualcosa non va. Quindi, in pratica, mi aiuta a non fare cavolate e a mantenere tutto allineato.

2. Le Classi per la Logica

Per quanto riguarda le classi, ho deciso di usarle per definire il comportamento delle entità (come partecipanti, corsi e aziende). Le classi sono utili perché posso incapsulare sia i dati che il comportamento in un singolo oggetto. Se voglio un partecipante, non solo ho i dati (nome, cognome, ecc.), ma ho anche il metodo che permette al partecipante di iscriversi a un corso.

Ad esempio, nella classe Partecipante ho scritto il metodo iscrivitiCorso per fare in modo che quando un partecipante si iscrive a un corso, il corso stesso aggiunga il partecipante alla lista degli iscritti:

typescript

```
class Partecipante implements IPartecipante {
constructor(
public nome: string,
public cognome: string,
public paeseDiOrigine: string,
public livelloDiIstruzione: string,
public competenzeLinguistiche: string[],
public ambitoDiFormazioneInteresse: string ) {}
iscrivitiCorso(corso: ICorso): void {
corso.aggiungiPartecipante(this);
console.log(`${this.nome} ${this.cognome} si è iscritto al corso "${corso.titolo}"`);  }}
```

Così, ogni partecipante sa come iscriversi ai corsi in modo autonomo. Non è il sistema a fare tutto per lui, ma è proprio il partecipante che, attraverso il metodo iscrivitiCorso, interagisce con il corso.

3. Corsi e Aziende con i Metodi Specifici

L'idea di mettere i metodi specifici dentro le classi Corso e Azienda è che ogni oggetto deve sapere cosa fare quando succedono certe cose. Ad esempio, quando un partecipante si iscrive a un corso, è il corso stesso che gestisce l'aggiunta di quel partecipante alla lista di iscritti. Non devo delegare questo compito a un'altra parte del codice, ma lo faccio direttamente nel corso. Stessa cosa per le aziende: quando una posizione è offerta a un partecipante, è l'azienda che fa questa offerta. La logica è localizzata e ben separata.

In pratica, nella classe Corso ho messo il metodo aggiungiPartecipante:

typescript

```
class Corso implements ICorso {
public iscritti: IPartecipante[] = [];
constructor(
public titolo: string,
public descrizione: string,
public settoreProfessionale: string,
public durata: number) {}
aggiungiPartecipante(partecipante: IPartecipante): void {
this.iscritti.push(partecipante);
console.log(`${partecipante.nome} si è iscritto al corso "${this.titolo}"`);}
```

In questo modo, il corso sa come aggiungere i partecipanti e non è necessario che qualcun altro lo faccia per lui. È una gestione più chiara e semplice.

4. Uso della Classe IncluDO per Gestire Tutto

La classe IncluDO è come un gestore centrale. È il "controllore" che coordina corsi, aziende e partecipanti. Ho deciso di fare una classe che gestisce l'intero flusso del programma perché altrimenti rischiamo di perdere il controllo su tutto e far fare troppe cose a troppe classi diverse. Quindi, IncluDO si occupa di:

Aggiungere corsi e aziende.

Iscrivere i partecipanti ai corsi.

Offrire posizioni alle aziende.

Questo approccio centralizza la logica in un unico posto, evitando che ogni singolo oggetto (come un corso o un'azienda) debba gestire l'intero flusso. Il codice rimane più chiaro, perché il flusso principale del programma è nel IncluDO.

Ecco un esempio:

typescript

```
class IncluDO {
private corsi: ICorso[] = [];
private aziende: IAzienda[] = [];
aggiungiCorso(corso: ICorso): void {
this.corsi.push(corso);
console.log(`Corso "${corso.titolo}" aggiunto con successo!`)}
aggiungiAzienda(azienda: IAzienda): void {
this.aziende.push(azienda);
console.log(`Azienda "${azienda.nomeAzienda}" aggiunta con successo!`);  }
iscriviPartecipanteACorso(partecipante: IPartecipante, corso: ICorso): void {
corso.aggiungiPartecipante(partecipante);
console.log(`${partecipante.nome} ${partecipante.cognome} si è iscritto al corso "${corso.titolo}"`); }
collegaPartecipanteAzienda(partecipante: IPartecipante, azienda: IAzienda, posizione: string): void {
if (azienda.posizioniAperte.includes(posizione)) {
azienda.offriPosizione(partecipante, posizione);
console.log(`Posizione "${posizione}" offerta a ${partecipante.nome} ${partecipante.cognome} presso "${azienda.nomeAzienda}"`);
} else {
console.log(`La posizione "${posizione}" non è disponibile presso "${azienda.nomeAzienda}"`);
}
}
}
```

Tutto il progetto è stato strutturato con l'idea di modularità e mantenibilità. Usare interfacce e classi mi permette di avere un codice pulito e facilmente estendibile. Le interfacce stabiliscono delle regole che tutti devono seguire, le classi definiscono il comportamento degli oggetti, e IncluDO centralizza la logica principale, facendo da ponte tra tutte le entità.
Se hai bisogno di più chiarimenti o vuoi approfondire qualche parte, fammi sapere!

fine