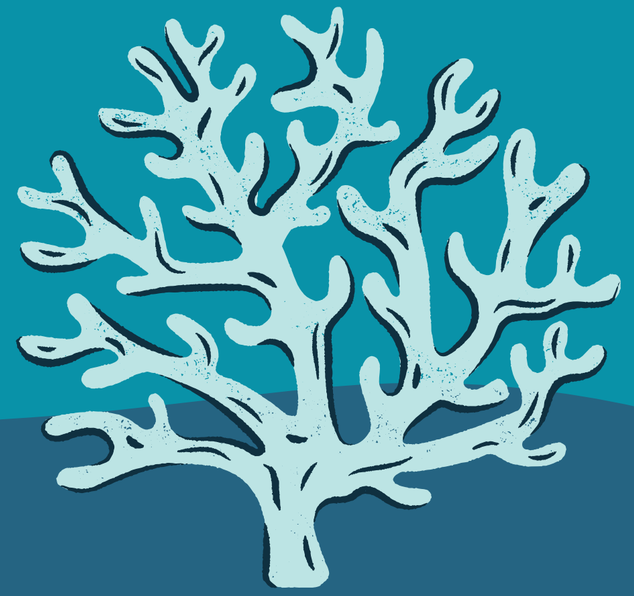
PROGETTO TYPESCRIPT

SUNNEE

brand di beachwear sostenibile



Autrice: Chiara Barletta

CodePen:https://codepen.io/Chiara-Barletta/pen/dPYoXBm

GitHub:https://github.com/junior5969/typescript-project

OBIETTIVI DEL PROGETTO

- Simulare un sistema di gestione per prodotti moda mare
- Gestione clienti, ordini e processi di produzione sostenibili
- Validazione dei dati e gestione dello stato dei prodotti

PROGETTAZIONE

Il progetto risulta rappresentato da 3 elementi principali rappresentati da interfacce e implementati in classi:
Cliente, Prodotto, Processo di produzione

Ogni elemento presenta proprietà e metodi, in particolare alcune proprietà sono multi valore ...

. . . ad esempio

PRODOTTO

Proprietà: tipo

Valori: costume da bagno, pareo, cappello

Proprietà: stato

Valori: disponibile, esaurito

Per questo motivo, per la definizione delle proprietà all'interno delle interfacce IProdotto, ICliente, e IProcessoProduzione, ho scelto di utilizzare una combinazione di:

Literal Types
per definire set finiti e controllati di valori validi

+

Type Guards
per verificare dinamicamente l’appartenenza un tipo definito

```ts
export type TipiProdotto = "costume da bagno" | "pareo" | "cappello";
```

```ts
const tipiValidi: TipiProdotto[] = ["costume da bagno", "pareo", "cappello"];

function isTipo(x: any): x is TipiProdotto {
  return typeof x === "string" && tipiValidi.includes(x as TipiProdotto);
}
```

In particolare, questi Type Guards sono stati utilizzati nei costruttori e nei metodi per bloccare in fase di esecuzione valori non ammessi e per lanciare errori informativi in caso di input errato, ad esempio:

```ts
export type TipiPagamento = "mastercard" | "postepay" | "bonifico";

const pagamentiValidi: TipiPagamento[] = ["mastercard", "postepay", "bonifico"];

function isPagamento(x: any): x is TipiPagamento {
  return (typeof x === 'string') && pagamentiValidi.includes(x as TipiPagamento);
}
```

```
class Cliente implements ICliente {
  nome: string;
  cognome: string;
  email: string;
  metodoPagamento: TipiPagamento;

  constructor(
    nome: string,
    cognome: string,
    email: string,
    metodoPagamento: TipiPagamento
  ) {
    if (!isPagamento(metodoPagamento)) {
      throw new Error(`Metodo di pagamento "${metodoPagamento}" non valido`);
    }

    this.nome = nome;
    this.cognome = cognome;
    this.email = email;
    this.metodoPagamento = metodoPagamento;
  }
```

RELAZIONI TRA LE ENTITA'

- Cliente → sceglie→ Prodotto

  se disponibile

- Prodotto → assegnato → Cliente

  stato:ordinato

- Prodotto → entra in → Processo di produzione

FLUSSO OPERATIVO

Il flusso dell'applicazione parte dal metodo ordinaProdotto() chiamato da Cliente. Questo metodo controlla se il Prodotto è disponibile:

- se non disponibile, stampa un messaggio di avviso;
- se disponibile, chiama prodotto.assegnaCliente(cliente).

Il metodo assegnaCliente() cambia lo stato del Prodotto in "ordinato" e stampa i dettagli del Cliente.

Infine, il Prodotto viene passato al metodo aggiungiProdotto(prodotto) per essere inserito nel Processo di produzione.

MODULARITÀ DEL CODICE

Il progetto è organizzo in moduli separati per ogni entità:

- Prodotto
- Cliente
- ProcessoProduzione
- Tipi e validazioni (types.ts)

per dividere la programmazione e rendere il codice più leggibile e riutilizzabile