oracolo

Una classe Oracolo espone un metodo per restituire un numero casuale (`stampaNumero`).

È richiesto di estendere il comportamento in modo da:

- Stampare un messaggio di benvenuto prima di tornare il numero;
- Stampare un messaggio di saluto alla fine;
- Dare la possibilità di invertire i messaggi a runtime.

Descrivere il design pattern utilizzato (nome e scopo generale) e la soluzione, includendo una breve descrizione del ragionamento e una eventuale rappresentazione del risultato finale.

Soluzione

Si può risolvere questo problema con il pattern Decorator, che permette di estendere i comportamenti di un oggetto esistente creandone uno nuovo. In questo modo è possibile estendere le proprietà ed i comportamenti già implementati senza modificare l'oggetto originale.

Il Decorator è strutturato in questo modo:

- Component: l'interfaccia che definisce i metodi comuni a tutti gli oggetti;
- Concrete Component: l'implementazione del componente di base;
- Base Decorator: il decoratore di base ha l'oggetto da estendere tra le sue proprietà e implementa o estende il componente di base;
- Concrete Decorator: aggiunge comportamenti estendendo il decoratore di base;
- Client: si occupa di creare il componente di base e passarlo al decoratore.

Esempio di implementazione

`oracolo.ts`

```
interface IOracolo {
  stampaNumero(): void
}

class Oracolo implements IOracolo {
  stampaNumero(): void {
    console.log(Math.random())
  }
}

abstract class OracoloDecorator implements IOracolo {
```

```
  protected oracolo: IOracolo

  constructor(oracolo: IOracolo) {
    this.oracolo = oracolo
  }

  stampaNumero(): void {
    this.oracolo.stampaNumero()
  }
}

class OracoloBenvenuto extends OracoloDecorator {
  stampaNumero(): void {
    console.log("Benvenuto!")
    super.stampaNumero()
  }
}

class OracoloArrivederci extends OracoloDecorator {
  stampaNumero(): void {
    super.stampaNumero()
    console.log("Arrivederci!")
  }
}

const oracolo = new Oracolo()
const oracoloBenvenuto = new OracoloBenvenuto(oracolo)
const oracoloArrivederci = new OracoloArrivederci(oracoloBenvenuto)
oracoloArrivederci.stampaNumero()
```

Output di esempio

```
Benvenuto!
0.07690864486562954
Arrivederci!
```

pizza

Stai sviluppando un software per il menù di una pizzeria. Ogni pizza ha un prezzo base e una serie di aggiunte disponibili che l'utente può scegliere. Ogni aggiunta ha un prezzo opzionale diverso e le scelte sono molte e potenzialmente variabili nel tempo.

Descrivi quale design pattern useresti per risolvere il problema motivando la risposta. Fornisci anche una rappresentazione in codice, pseudo-codice o diagramma delle classi della soluzione proposta.

Soluzione

Come per l'oracolo, si può risolvere questo problema usando il Decorator. In questo modo è possibile aggiungere un numero illimitato di ingredienti senza modificare la classe base `Pizza`.

Esempio di implementazione

`pizza.ts`

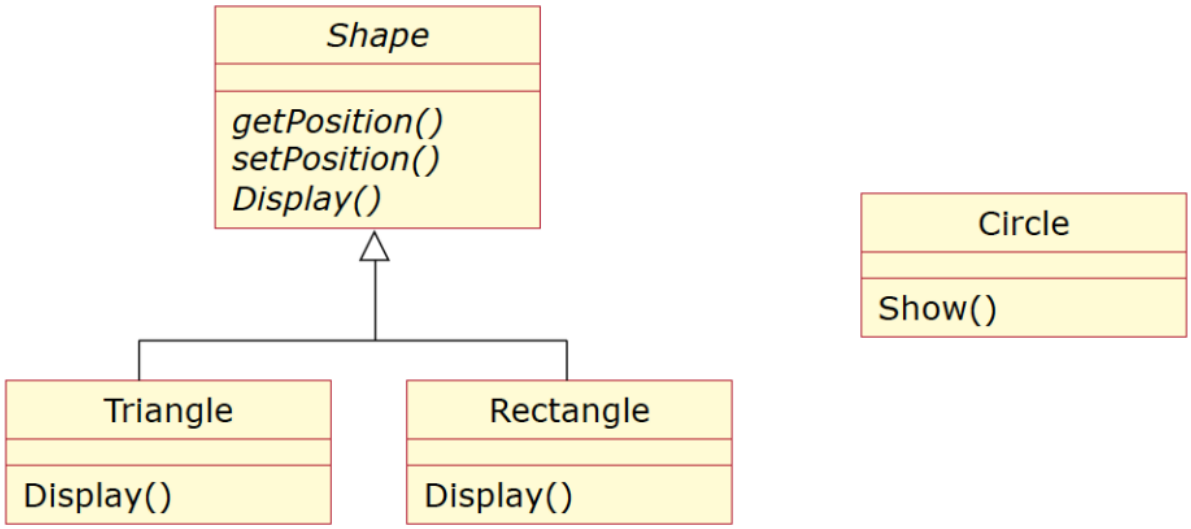
```ts
interface IPizza {
  name: string
  price: number
}

class Pizza implements IPizza {
  name: string
  price: number

  constructor(name: string, price: number) {
    this.name = name
    this.price = price
  }
}

abstract class PizzaDecorator implements IPizza {
  protected pizza: IPizza

  constructor(pizza: IPizza) {
    this.pizza = pizza
  }

  get name(): string {
    return this.pizza.name
```

```
  }

  get price(): number {
    return this.pizza.price
  }
}

class PizzaWithMushrooms extends PizzaDecorator {
  get name(): string {
    return `${this.pizza.name} + mushrooms`
  }

  get price(): number {
    return this.pizza.price + 1.5
  }
}

class PizzaWithWholeWheat extends PizzaDecorator {
  get name(): string {
    return `${this.pizza.name} + whole wheat`
  }

  get price(): number {
    return this.pizza.price + 2
  }
}

const pizza = new Pizza("Margherita", 5)
console.log("Base pizza: " + pizza.name + " - €" + pizza.price)

const pizza2 = new PizzaWithMushrooms(pizza)
console.log("Pizza with additions: " + pizza2.name + " - €" + pizza2.price)

const pizza3 = new PizzaWithWholeWheat(pizza2)
console.log("Pizza with additions: " + pizza3.name + " - €" + pizza3.price)
```

Output di esempio

```
Base pizza: Margherita - €5
Pizza with additions: Margherita + mushrooms - €6.5
Pizza with additions: Margherita + mushrooms + whole wheat - €8.5
```

shapes



- Una classe astratta `Shape` con i metodi indicati;
- Due implementazioni specifiche (`Triangle` e `Rectangle`) che implementano il metodo `display()` per stampare la forma;
- Una classe `Circle` derivante da una libreria che ha solo il metodo `show()` per stampare la forma;
- Un client che già lavora con `Shape` di tipo triangolo e rettangolo, ci viene chiesto di aggiungere anche la possibilità di gestire cerchi, come procedete?

```
class Position {
  readonly x: number
  readonly y: number
  constructor(x: number, y: number) {
    this.x = x
    this.y = y
  }
}

abstract class Shape {
  protected position: Position = new Position(0, 0)
  getPosition(): Position {
    return this.position
  }
```

```
  setPosition(position: Position) {
    this.position = position
  }

  abstract display()
}

class Triangle extends Shape {
  display() {
    // logica di stampa del triangolo
  }
}

class Rectangle extends Shape {
  display() {
    // logica di stampa del rettangolo
  }
}

class Circle {
  show(x: number, y: number) {
    //logica per stampa del cerchio
  }
}

// logica del client
const shapes: Shape[] = []

shapes.push(new Triangle())
shapes.push(new Rectangle())
// fare in modo di poter aggiungere anche cerchi

shapes.forEach((shape) => shape.display())
```

Descrivere il design pattern utilizzato (nome e scopo generale) e la soluzione, includendo una breve descrizione del ragionamento e una eventuale rappresentazione del risultato finale.

Soluzione

Per risolvere il problema si può utilizzare il design pattern **Adapter**, che permette a oggetti incompatibili tra loro di comunicare. È strutturato in questo modo:

- **Client**: la logica esistente;
- **Client Interface**: interfaccia che le classi devono usare per comunicare con il client;
- **Service**: la classe che non può essere utilizzata allo stato attuale e necessita un adattamento;
- **Adapter**: la classe che consente l'utilizzo del Service con la logica esistente. Riceve istruzioni dal client e le traduce in qualcosa che la classe incompatibile può gestire.

Si può creare un nuovo oggetto che estende `Shape` e richiama il metodo `show()` di `Circle`, in modo da poterlo utilizzare allo stesso modo del metodo `display()` di `Triangle` e `Rectangle`.

Esempio di implementazione

```
class CircleAdapter extends Shape {
  private circle: Circle

  constructor(circle: Circle) {
    super()
    this.circle = circle
  }

  display() {
    circle.show(this.getPosition().x, this.getPosition().y)
  }
}
```

fish

Abbiamo due classi di pesci: `BigFish` e `LittleFish`, ognuna con la sua implementazione. Entrambi i pesci si muovono in modo casuale tramite il metodo `move()`.
`BigFish` si può muovere in una posizione occupata da un `LittleFish` (e mangiarlo).
`LittleFish` invece non si può muovere dove si trova un `BigFish`.

Descrivere:

- Se e come mai è utile un design pattern per implementare le classi di questo problema
- Quale struttura avranno le classi alla fine (usare pseudo-codice, non ci interessa l'implementazione esatta del movimento e dei controlli)
- Il ragionamento usato per la divisione in classi

Soluzione

Per risolvere questo problema si può utilizzare il **Template Method**, un design pattern che permette di definire una classe con un comportamento di base che potrà essere esteso o modificato in parte attraverso delle sottoclassi.

Esempio di implementazione (pseudocodice)

```
class Position {
  posX: number
  posY: number
  fish: Fish
}

class Ocean {
  fishList: List<Fish>
}

abstract class Fish {
  pos: Position

  constructor(pos: Position) {
    this.pos = pos
  }

  tryMove(newPos: Position): void {
    if (this.canMoveTo(newPos)) {
      this.move(newPos)
    }
  }
```

```
  abstract canMoveTo(pos: Position): boolean

  move(newPos: Position): void {
    this.pos = newPos
  }
}

class BigFish extends Fish {
  canMoveTo(pos: Position): boolean {
    return !(pos.fish instanceof BigFish)
  }

  move(pos: Position, ocean: Ocean): void {
    if (pos.fish instanceof LittleFish) {
      ocean.fishList.remove(pos.fish)
    }

    this.pos = pos
  }
}

class LittleFish extends Fish {
  canMoveTo(pos: Position, ocean: Ocean): boolean {
    return !pos.fish || pos.fish instanceof LittleFish
  }
}
```