Il pattern **Strategy** è un modello di progettazione comportamentale che consente di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Questo pattern permette di selezionare l'algoritmo da utilizzare in modo dinamico, rendendo il codice più flessibile e mantenibile.

Componenti del Pattern Strategy

1. **Strategy (Strategia)**: Un'interfaccia comune a tutte le strategie concrete. Dichiarerà il metodo (o i metodi) che tutte le strategie devono implementare.
2. **ConcreteStrategy (Strategia Concreta)**: Implementa l'interfaccia Strategy con un particolare algoritmo o comportamento.
3. **Context (Contesto)**: Mantiene un riferimento a una Strategy e delega il comportamento a quella strategia.

Come Funziona

- **Context**: Ha un riferimento a una Strategy concreta. Può cambiare la Strategy in modo dinamico.
- **Strategy**: Definisce l'interfaccia per l'algoritmo o il comportamento.
- **ConcreteStrategy**: Implementa l'algoritmo specifico seguendo l'interfaccia Strategy.

Esempio con Pokémon

Immagina di avere un Pokémon che può utilizzare diverse strategie di attacco. Puoi utilizzare il pattern Strategy per implementare diverse modalità di attacco che possono essere intercambiate dinamicamente.

Interfaccia Strategy

java Copia codice

```java
public interface StrategiaAttacco {
    void attacca(Pokemon attaccante, Pokemon difensore);
}
```

ConcreteStrategy

Attacco di Fuoco

java Copia codice

```java
public class AttaccoFuoco implements StrategiaAttacco {
    @Override
    public void attacca(Pokemon attaccante, Pokemon difensore) {
        System.out.println(attaccante.getNome() + " usa Attacco di Fuoco su " + difensore.getNome());
        // Logica di attacco di fuoco
    }
}
```

Attacco d'Acqua

java Copia codice

```java
public class AttaccoAcqua implements StrategiaAttacco {
    @Override
    public void attacca(Pokemon attaccante, Pokemon difensore) {
        System.out.println(attaccante.getNome() + " usa Attacco d'Acqua su " + difensore.getNome());
        // Logica di attacco d'acqua
    }
}
```

Context

java Copia codice

```java
public class Pokemon {
    private String nome;
    private StrategiaAttacco strategiaAttacco;

    public Pokemon(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    public void setStrategiaAttacco(StrategiaAttacco strategiaAttacco) {
        this.strategiaAttacco = strategiaAttacco;
    }

    public void attacca(Pokemon difensore) {
        strategiaAttacco.attacca(this, difensore);
    }
}
```

Applicazione Principale

java Copia codice

```java
public class MainApp {
    public static void main(String[] args) {
        Pokemon pikachu = new Pokemon("Pikachu");
        Pokemon charmander = new Pokemon("Charmander");

        // Imposta la strategia di attacco di fuoco per Pikachu
        pikachu.setStrategiaAttacco(new AttaccoFuoco());
        pikachu.attacca(charmander); // Output: Pikachu usa Attacco di Fuoco su Charmander

        // Cambia la strategia di attacco ad acqua per Pikachu
        pikachu.setStrategiaAttacco(new AttaccoAcqua());
        pikachu.attacca(charmander); // Output: Pikachu usa Attacco d'Acqua su Charmander
    }
}
```

Come Funziona Questo Esempio

1. **StrategiaAttacco**: È l'interfaccia che definisce il metodo **`attacca`** che tutte le strategie concrete devono implementare.
2. **AttaccoFuoco** e **AttaccoAcqua**: Sono implementazioni concrete dell'interfaccia **`StrategiaAttacco`**. Implementano il metodo **`attacca`** con la logica specifica dell'attacco di fuoco e dell'attacco d'acqua rispettivamente.
3. **Pokemon (Context)**: Contiene un riferimento a una **`StrategiaAttacco`** e usa questo riferimento per eseguire il metodo **`attacca`**. La strategia di attacco può essere cambiata dinamicamente utilizzando il metodo **`setStrategiaAttacco`**.

Benefici del Pattern Strategy

- **Modularità**: Ogni algoritmo è incapsulato in una classe separata, migliorando la modularità del codice.
- **Manutenibilità**: È più facile aggiungere nuovi algoritmi o modificare quelli esistenti senza modificare il contesto.
- **Flessibilità**: Gli algoritmi possono essere cambiati a runtime, permettendo una grande flessibilità.

Questo pattern è utile in scenari in cui il comportamento di un oggetto deve essere selezionabile a runtime, come nel caso degli attacchi dei Pokémon che possono cambiare strategia in base alla situazione.