# Zusammenfassung: Strategy Pattern

**1. Entwurfsmuster – Einordnung**

Design Patterns sind bewährte Lösungsansätze für wiederkehrende Probleme.

Drei Kategorien:

- Erzeugungsmuster (z. B. Factory, `Singleton`)
- Strukturmuster (z. B. Adapter, `Decorator`)
- Verhaltensmuster (z. B. Observer, `Strategy`)

Ziel: lesbarer, wartbarer Code, nicht möglichst viele Patterns verwenden.

**2. Ausgangsszenario: SimUDuck**

Simulation eines Ententeichs mit vielen verschiedenen Enten.

Enten können:

- `schwimmen`
- `quaken`
- `gezeichnet werden`

Anfangsdesign: Basisklasse Ente, konkrete Unterklassen wie Stockente, Reiherente.

**3. Problem: Erweiterung um neue Verhaltensweisen**

Neue Anforderung: Enten sollen fliegen können.

Problemfälle:

- Gummienten können nicht fliegen.
- Gummienten quaken nicht, sondern quietschen.

Erste naive Lösung:

Methoden `fliegen()` und `quaken()` in Unterklassen überschreiben.

Nachteile:

- Viel Code-Duplikation
- Schwer wartbar
- Änderungen an Verhalten betreffen viele Klassen

**4. Fehlversuch mit Interfaces**

Idee: Interfaces wie Flugfaehig und Quakfaehig.

Problem:

- Verhalten ist fest an Klassen gebunden
- Keine Wiederverwendung von Verhalten
- Kein einfaches Ändern des Verhaltens zur Laufzeit

**5. Lösung: Strategy Pattern**

Grundidee:

Kapsle veränderliches Verhalten in eigene `Klassen` und mache es austauschbar.

a) Strategien definieren

```
- Flugverhalten
- FliegenMitFluegel
- KeinFliegen
- Quakverhalten
- Quaken
- Quietschen
- StummesQuaken
```

Alle implementieren jeweils ein gemeinsames `Interface`.

**6. Neue Struktur mit Strategy Pattern**

Die `abstrakte Klasse` Ente:

besitzt Referenzen auf Flugverhalten und Quakverhalten

delegiert `fliegen()` und `quaken()` an diese Objekte

erlaubt das Setzen des Verhaltens zur Laufzeit

Beispiel:

```
- Gummiente
- NichtFliegen
- Quietschen
```

**7. Vorteile des Strategy Patterns**

- [x] Vermeidung von Code-Duplikation
- [x] Verhalten kann zur Laufzeit geändert werden
- [x] Klassen bleiben offen für Erweiterung, aber geschlossen für Änderung
- [x] Bessere Wartbarkeit und Flexibilität

**Merksatz (prüfungsrelevant 😉):**

„Programmiere gegen Interfaces, nicht gegen Implementierungen.“