„3. Liskov Substitution Principle (LSP)„

Das **Liskov Substitution Principle (LSP)** oder auf Deutsch das **Liskovsche Substitutionsprinzip** ist eines der SOLID-Prinzipien der objektorientierten Programmierung. Es wurde von Barbara Liskov formuliert und besagt, dass **Unterklassen ohne Probleme anstelle ihrer Basisklassen verwendet werden können sollten**, ohne dass das Verhalten des Programms verändert wird.

Das bedeutet, dass Objekte von Unterklassen die Funktionen ihrer Basisklasse vollständig erfüllen müssen. Wenn eine Unterklasse das Verhalten so stark ändert, dass es das ursprüngliche Verhalten der Basisklasse verletzt, verstößt dies gegen das LSP.

Kern des LSP:

- **Wenn S eine Unterklasse von T ist**, dann sollten Objekte von Typ S sich so verhalten, dass sie überall dort eingesetzt werden können, wo Objekte vom Typ T erwartet werden.
- Anders ausgedrückt: **Funktionen, die Objekte einer Basisklasse verwenden, sollten in der Lage sein, auch Objekte von abgeleiteten Klassen zu verwenden, ohne dass das erwartete Verhalten geändert wird.**

Beispiel: Rechtecke und Quadrate:

Nehmen wir an, wir haben eine Basisklasse `Rechteck` und eine Unterklasse `Quadrat`. Da ein Quadrat eine spezielle Art von Rechteck ist, könnte man annehmen, dass eine Vererbung von `Quadrat` von `Rechteck` sinnvoll wäre.

```java
class Rechteck {
    protected int breite;
    protected int höhe;

    public void setBreite(int breite) {
        this.breite = breite;
    }

    public void setHöhe(int höhe) {
        this.höhe = höhe;
    }

    public int berechneFläche() {
        return breite * höhe;
    }
}
```

Nun erstellen wir eine Unterklasse `Quadrat`, weil ein Quadrat ein Rechteck ist, bei dem **Breite und Höhe gleich sind**.

java Code kopieren

```java
class Quadrat extends Rechteck {
    @Override
    public void setBreite(int breite) {
        this.breite = breite;
        this.höhe = breite;  // Beim Quadrat müssen Höhe und Breite gleich sein
    }

    @Override
    public void setHöhe(int höhe) {
        this.breite = höhe;
        this.höhe = höhe;  // Beim Quadrat müssen Höhe und Breite gleich sein
    }
}
```

Auf den ersten Blick sieht das logisch aus. Aber was passiert, wenn wir ein Quadrat als ein Rechteck verwenden?

```java
Rechteck rechteck = new Quadrat();
rechteck.setBreite(5);
rechteck.setHöhe(10);
System.out.println("Fläche: " + rechteck.berechneFläche());
```

Problem:

Hier erwarten wir vielleicht eine Fläche von 50 (5 * 10). Doch da ein Quadrat sicherstellt, dass **Breite und Höhe immer gleich sind**, wird die Breite durch die `setHöhe()` Methode auf 10 geändert. Die berechnete Fläche ist also 100 (10 * 10) – **ein unerwartetes Ergebnis**! Das verstößt gegen das LSP, weil das Quadrat nicht das Verhalten der Basisklasse Rechteck korrekt unterstützt.

Warum verletzt dieses Beispiel das LSP?

- Ein Quadrat kann nicht sinnvoll alle Eigenschaften eines Rechtecks übernehmen, ohne dabei den Code zu verletzen.
- In diesem Fall funktioniert die Beziehung "Ein Quadrat ist ein Rechteck" nicht, weil das Quadrat eine **Einschränkung** hat, die das Rechteck nicht hat: Die Seiten müssen immer gleich lang sein. Daher führt das zu **unerwartetem Verhalten**, wenn das Quadrat als Rechteck verwendet wird.

Lösung für dieses Problem:

Anstatt `Quadrat` von `Rechteck` abzuleiten, sollten wir die beiden Konzepte getrennt behandeln, weil ihre Eigenschaften zwar ähnlich, aber nicht identisch sind. Hier könnten wir stattdessen eine gemeinsame Oberklasse verwenden, z. B. `Form`, und sowohl `Rechteck` als auch `Quadrat` davon ableiten:

```
class Form {
    public int berechneFläche() {
        // Allgemeine Logik für die Fläche, wenn möglich
        return 0;
    }
}

class Rechteck extends Form {
    protected int breite;
    protected int höhe;

    public Rechteck(int breite, int höhe) {
        this.breite = breite;
        this.höhe = höhe;
    }

    @Override
    public int berechneFläche() {
        return breite * höhe;
    }
}

class Quadrat extends Form {
    protected int seitenLänge;

    public Quadrat(int seitenLänge) {
        this.seitenLänge = seitenLänge;
    }

    @Override
    public int berechneFläche() {
        return seitenLänge * seitenLänge;
    }
}
```

In diesem Fall können wir `Rechteck` und `Quadrat` separat behandeln und haben keine ungewollten Nebeneffekte mehr, wenn wir die beiden Typen austauschen.

Wie erkennt man Verstöße gegen das LSP?

Um festzustellen, ob das LSP verletzt wird, kannst du dir folgende Fragen stellen:

1. **Können Unterklassen problemlos anstelle ihrer Basisklasse verwendet werden, ohne dass sich das Verhalten des Programms ändert?**
2. **Erwartet der Code ein bestimmtes Verhalten, das durch die Unterklasse gebrochen wird?**
3. **Wird das Verhalten durch die Unterklasse eingeschränkt oder verändert, sodass es zu unerwarteten Ergebnissen kommt?**

Zusammenfassung:

- **Liskov Substitution Principle (LSP)**: Unterklassen sollten ihre Basisklassen ersetzen können, ohne dass das Programmverhalten verändert wird.
- **Warum wichtig?**: Verletzungen des LSP führen zu **unerwartetem Verhalten**, schwer zu wartendem Code und potenziellen Bugs.
- **Typische Probleme**: Wenn Unterklassen das Verhalten der Basisklasse einschränken oder Methoden unerwartete Ergebnisse liefern (z. B. ein flugunfähiger Vogel, der fliegen soll), wird das LSP verletzt.
- **Lösung**: Durch korrekte Vererbung, Vermeidung von übermäßiger Spezialisierung oder Nutzung von Schnittstellen kann man das LSP einhalten und sauberen, wartbaren Code erstellen.