Informationen zu S.O.L.I.D

Von Info-Website Lernsituation 06

- **S**: Single-Responsibility-Prinzip
- **O**: Open-Closed-Prinzip
- **L**: Liskovsches Substitutionsprinzip
- **I**: Interface-Segregation-Prinzip
- **D**: Dependency-Inversion-Prinzip

S: Single-Responsibility-Prinzip

In erster Linie beschreibt das Single-Responsibility-Prinzip, dass jede Klasse nur eine fest definierte Aufgabe haben soll. Diese Aufgabe soll, soweit es möglich ist und es Sinn ergibt, abstrahiert sein.

Das Single-Responsibility-Prinzip (SRP) ist ein grundlegendes Konzept im Software-Design und Teil der S.O.L.I.D-Prinzipien. Es besagt, dass eine Klasse (oder eine Funktion) in einem Software-System nur eine einzige Verantwortlichkeit oder Aufgabe haben sollte. Mit anderen Worten, eine Klasse sollte nur für eine spezifische Art von Arbeit verantwortlich sein.

Die Idee hinter dem SRP ist, den Code klarer und wartbarer zu gestalten, indem man sicherstellt, dass eine Klasse nicht zu viele verschiedene Dinge gleichzeitig tut. Wenn eine Klasse mehrere Verantwortlichkeiten hat, wird der Code oft unübersichtlich und schwer zu pflegen, da Änderungen an einer Verantwortlichkeit Auswirkungen auf andere Teile der Klasse haben können.

Durch die Anwendung des SRP wird der Code modularer und leichter verständlich. Jede Klasse kann sich auf eine spezifische Aufgabe konzentrieren und ist nicht von anderen Aufgaben abhängig. Dies fördert auch die Wiederverwendbarkeit, da gut definierte Klassen leichter in anderen Teilen des Systems eingesetzt werden können.

In der Praxis bedeutet das SRP, dass Sie Klassen oder Funktionen so gestalten sollten, dass sie nur für eine bestimmte Art von Aufgabe verantwortlich sind und sich nicht mit anderen Dingen befassen. Wenn Sie feststellen, dass eine Klasse zu viele Verantwortlichkeiten hat, sollten Sie sie in kleinere Klassen aufteilen, um das SRP zu respektieren und den Code sauberer und wartbarer zu gestalten.

O: Open-Closed-Prinzip

- Module sollten sowohl offen (für Erweiterungen) als auch geschlossen (für Modifikationen) sein.“ (Bertrand Meyer [2])
- Das Open-Closed-Prinzip (OCP) beschreibt, dass Änderungen an Klassen oder Funktionen das Verhalten dieser nicht ändern dürfen.

- Das Open-Closed-Prinzip (OCP) ist ein Grundsatz des Software-Designs, der besagt, dass Software-Entitäten (z. B. Klassen, Module, Funktionen) offen für Erweiterungen sein sollten, aber geschlossen für Modifikationen. Mit anderen Worten, bereits existierender Quellcode sollte nicht verändert werden müssen, um neue Funktionen oder Verhalten hinzuzufügen.

- Das OCP fördert die Erstellung von Software, die leicht erweiterbar ist, ohne die bestehende Funktionalität zu gefährden. Anstatt vorhandenen Code zu ändern, um neue Anforderungen zu erfüllen, sollten Entwickler in der Lage sein, neue Funktionalität hinzuzufügen, indem sie Klassen erweitern oder neue Klassen erstellen, die von den vorhandenen Klassen abgeleitet sind.

- Dieses Prinzip trägt dazu bei, dass der Code stabiler und wartungsfreundlicher wird. Es ermöglicht auch die Wiederverwendung von bestehendem Code, da Änderungen in einer Klasse die anderen Teile des Systems nicht beeinflussen. Das OCP ist ein wichtiger Bestandteil der S.O.L.I.D-Prinzipien des Software-Designs und fördert die Entwicklung von flexiblen und erweiterbaren Softwarelösungen.

L: Liskovsches Substitutionsprinzip

- „Sei q(x) eine Eigenschaft des Objektes x vom Typ T, dann sollte q(y) für alle Objekte y des Typs S gelten, wobei S ein Subtyp von T ist.“ (Barbara Liskov [3])
- Um es vereinfacht auszudrücken: Objekte einer abgeleiteten Instanz müssen sich genauso verhalten wie eben diese, damit Verwender glauben, die Basisklasse zu benutzen.

- Das Liskovsche Substitutionsprinzip (LSP) ist ein Prinzip des objektorientierten Software-Designs, das von Barbara Liskov entwickelt wurde. Es besagt, dass Objekte einer abgeleiteten Klasse in der Lage sein sollten, anstelle von Objekten ihrer Basisklasse verwendet zu werden, ohne die Korrektheit des Programms zu beeinträchtigen.

- In einfachen Worten bedeutet dies, dass eine abgeleitete Klasse eine erweiterte Version ihrer Basisklasse sein sollte und alle Verträge, die von der Basisklasse definiert sind, erfüllen muss. Ein Client, der eine Instanz der

Basisklasse verwendet, sollte in der Lage sein, stattdessen eine Instanz der abgeleiteten Klasse zu verwenden, ohne unerwartetes Verhalten oder Fehler im Programm zu verursachen.

- Das LSP trägt zur Erreichung von Robustheit und Konsistenz in objektorientierten Systemen bei, indem es sicherstellt, dass Subklassen die erwarteten Verhalten und Invarianten der Basisklasse respektieren. Wenn das LSP verletzt wird, kann dies zu Fehlern und inkonsistentem Verhalten führen.

- In der Praxis bedeutet das Liskovsche Substitutionsprinzip, dass Entwickler sicherstellen müssen, dass abgeleitete Klassen die Verträge und Erwartungen der Basisklassen nicht verletzen und gleichzeitig zusätzliche Funktionalität hinzufügen können. Dies ermöglicht es, die abgeleiteten Klassen anstelle der Basisklassen zu verwenden, ohne die Integrität des Systems zu gefährden.

I: Interface-Segregation-Prinzip

- „Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden.“ (Robert C. Martin [4])
- Wie der Name des Prinzips vermuten lässt, besagt es, dass ein Client nicht gezwungen werden soll, ob via Interface oder Vererbung Abhängigkeiten zu nutzen, die er gar nicht benötigt. Kurz gesagt soll man versuchen, seine Interfaces so schlank und spezifisch und gleichzeitig so genau wie möglich zu halten. Eine Regel für die Validierung braucht beispielsweise nur diese eine Funktion (validate) fest vorgeschrieben. Denn diese Regel hat lediglich die Aufgabe zu validieren. Es könnte sein, das einzelne Regeln noch andere Funktionen benötigen, wie beispielsweise ein Repository abzufragen. Jedoch gilt das nicht universell für alle Regeln. Es ergibt also keinen Sinn, nur weil eine Regel eine Funktion wie getUserFromDatabase benötigt, dieses Prinzip auf alle Regeln anzuwenden. Sollte man jedoch bemerken, dass in unserem Beispiel viele Regeln die getUserFromDatabase-Funktion benötigen, so ist es durchaus sinnvoll, ein weiteres Interface zu schreiben und nicht das Bestehende zu erweitern.
- Das Interface-Segregation-Prinzip (ISP) ist ein Prinzip des objektorientierten Software-Designs, das besagt, dass Clients nicht gezwungen sein sollten, von Schnittstellen abhängige Methoden zu implementieren, die sie nicht verwenden. Kurz gesagt, es fordert, dass Schnittstellen in kleinere, spezifischere Teile aufgeteilt werden, um eine übermäßige Abhängigkeit zu vermeiden.

- Das ISP zielt darauf ab, den sogenannten "Interface Pollution" zu verhindern, bei dem eine Schnittstelle viele Methoden enthält, von denen einige nicht für

alle Implementierer relevant sind. Dies kann dazu führen, dass Klassen gezwungen sind, leere oder nicht implementierte Methoden bereitzustellen, was zu unnötigem Code und einer erhöhten Komplexität führt.

- Durch die Aufteilung von großen Schnittstellen in kleinere, spezialisierte Schnittstellen können Klassen nur die Methoden implementieren, die für sie relevant sind. Dies führt zu saubererem Code, geringerer Kopplung zwischen Klassen und einer besseren Wartbarkeit.

- Das Interface-Segregation-Prinzip ist ein wichtiger Bestandteil der S.O.L.I.D-Prinzipien des Software-Designs und fördert die Erstellung von flexiblen und modularen Softwarelösungen, bei denen Klassen nur auf die Funktionen zugreifen, die sie benötigen, und nicht mit unnötigen Abhängigkeiten belastet werden.

D: Dependency-Inversion-Prinzip

- „Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.“ (Robert C. Martin)
- Das letzte Prinzip von SOLID beschäftigt sich mit den Abhängigkeiten von abgeschlossenen Klassen untereinander. Klassen, die eine höhere Hierarchie innerhalb der Software aufweisen, sollen grundsätzlich abhängig von Klassen niedrigerer Hierarchien sein. Denn grundsätzlich gilt die Regel: Je niedriger die Klasse innerhalb der Hierarchie, desto spezifischere Probleme löst sie. Wenn eine hierarchisch niedrigere Klasse von einer höheren Klasse abhängig ist, entsteht ein Problem.
- Haben wir bspw. eine Klasse, die einen Sortieralgorithmus beinhaltet, zum Beispiel Bubblesort, so funktioniert dieser ohne jede Abhängigkeit zu anderen Klassen. Es sollte also nicht vorkommen, dass in dieser Klasse eine Abhängigkeit zu beispielsweise einem Repository besteht. Das würde das erste und das fünfte Prinzip verletzen.
- Das Dependency-Inversion-Prinzip (DIP) ist ein Grundsatz des objektorientierten Software-Designs, der als Teil der S.O.L.I.D-Prinzipien entwickelt wurde. Das Prinzip besagt, dass Abhängigkeiten in einer Software so organisiert sein sollten, dass sie von höheren Ebenen zu niedrigeren Ebenen zeigen. Kurz gesagt, die Abhängigkeiten sollten umgekehrt sein, wobei hochrangige Module nicht von niedrigrangigen Modulen abhängen, sondern beide von abstrakten Schnittstellen oder abstrakten Klassen abhängen.

- Das DIP fördert die Trennung von Schnittstellen von konkreten Implementierungen und ermöglicht es, Module unabhängig voneinander zu entwickeln und zu testen. Dies trägt zur Flexibilität und Wartbarkeit des Codes bei, da Änderungen an niedrigeren Ebenen keine Auswirkungen auf höhere Ebenen haben.

- In der Praxis bedeutet das DIP, dass Sie Abstraktionen (z. B. Schnittstellen oder abstrakte Klassen) verwenden sollten, um Abhängigkeiten zwischen Modulen zu definieren. Statt sich direkt auf konkrete Implementierungen zu verlassen, sollten Module von diesen abstrakten Schnittstellen oder Klassen abhängen. Dies ermöglicht es, verschiedene Implementierungen auszutauschen, ohne den gesamten Code zu ändern, solange sie die vereinbarten Schnittstellen erfüllen.

- Das Dependency-Inversion-Prinzip trägt zur Erstellung flexibler, erweiterbarer und wartbarer Softwarelösungen bei, indem es die Abhängigkeiten zwischen Modulen auf eine Weise organisiert, die die Anpassung an neue Anforderungen und die Wiederverwendung von Code erleichtert.