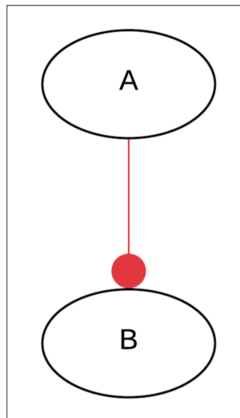
LIESERS CLEAN CODE

Vom DIP zum IOSP

Das Dependency Inversion Principle hat die Softwareentwicklung beim Umgang mit Abhängigkeiten entscheidend vorangebracht. Zeit, den nächsten Schritt zu gehen.

Abhängigkeiten zwischen Funktionseinheiten sind eine der größten Herausforderungen bei der Entwicklung von Softwaresystemen. Das gilt auf jeder Ebene der Modulhierarchie, also für Methoden, Klassen, Bibliotheken, Pakete, Komponenten und Microservices. Sobald Abhängigkeiten ungeplant und unreflektiert wuchern können, steht es schlecht um das System. Dies hat zwei Gründe. Zum einen erschweren Abhängigkeiten das automatisierte Testen. Wenn immer nur „alles“ getestet werden kann, steigt der Aufwand für das Sicherstellen der Korrektheit immens an. Insofern ist es notwendig, Abhängigkeiten so zu strukturieren, dass sich die entstehenden Einheiten leicht automatisiert testen lassen.



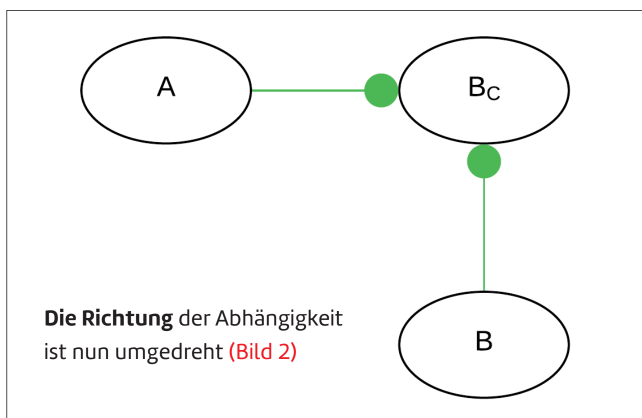
Nachteilige Richtung der Abhängigkeit (Bild 1)

Der zweite Grund ist die Wandelbarkeit. Wenn ein Entwickler die Zusammenhänge zwischen den Bestandteilen des Systems nicht leicht verstehen kann, ist es nicht weit her mit der Wandelbarkeit. Dann wird jede Änderung das Problem weiter verstärken.

Aus diesem Grund lautet die Empfehlung, die Abhängigkeitsrichtung umzudrehen. Als Prinzip ist dies bekannt als Dependency Inversion Principle (DIP).

Die Richtung der Abhängigkeiten umzudrehen mag zunächst sehr abstrakt klingen, jedoch steckt darin tatsächlich eine Lösung für das Problem der Abhängigkeiten.

In Bild 1 ist eine Abhängigkeit zwischen den beiden Funktionseinheiten *A* und *B* zu sehen. Die Richtung dieser Abhängigkeiten ist „falsch herum“, angedeutet durch die rote Einfärbung des Pfeils. Zunächst spielt es keine Rolle, welche Richtung wir als nachteilig betrachten. Schauen wir uns die Antwort des Dependency Inversion Principle an. In Bild 2 ist die Abhängigkeitsstruktur gemäß DIP geändert. Zwischen *A* und *B* liegt nun ein Kontrakt (engl. Contract) für *B*. Somit ist *A* nun nicht mehr direkt von *B* abhängig, sondern von B_C, dem Kontrakt für *B*.



Die Richtung der Abhängigkeit ist nun umgedreht (Bild 2)

Nun könnte man sagen, dass es sich hierbei um einen Trick handelt. In Bild 3 sind die beiden Funktionseinheiten sowie der Kontrakt wieder vertikal angeordnet. Somit liegt erneut eine Abhängigkeit in der Richtung von oben nach unten vor. Daraus könnte man schlussfolgern, dass die Abhängigkeiten doch nicht vollständig umgekehrt wurden.

Im Unterschied zur Ausgangssituation ist *A* aber jetzt nur noch vom Kontrakt B_C abhängig. Insofern handelt es sich um eine deutliche Verbesserung der Situation, da nun der Kontrakt auch anderweitig implementiert werden kann. *A* ist eben nicht mehr unmittelbar von *B* abhängig.

Wie bereits erwähnt stellt das Dependency Inversion Principle einen wichtigen Entwicklungsschritt dar, weil unter Beachtung des Prinzips plötzlich das Testen wieder möglich wird, ohne dass gleich „alles“ getestet werden muss. In Kombination mit Dependency Injection und Attrappen lassen sich nun einzelne Einheiten isoliert testen.

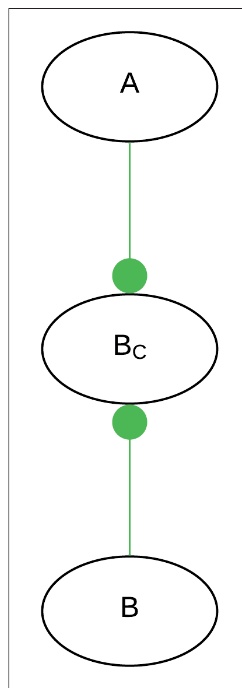
Doch treten wir einen Schritt zurück und fragen uns, was das Grundproblem ist und ob es durch das DIP bereits abschließend gelöst ist.

Das Problem ist hier nicht, wie man vermuten würde, die Abhängigkeit an sich. Stattdessen entsteht das Problem dadurch, dass in der Funktionseinheit *A* zwei Aspekte vermischt sind. *A* enthält Logik und verwendet *B*. Würde *A* keine Logik enthalten, sondern nur *B* verwenden, ergäbe die Abhängigkeitsstruktur keinen Sinn. Dann wäre *A* eine leere Hülle um *B*. Weil *A* also Logik enthält, möchten wir Tests schreiben für *A*. Dabei entsteht dann schnell der Wunsch, *A* isoliert zu testen. Dazu ist es dann erforderlich, B_C zwischen die Abhängigkeit zu stellen, um mithilfe von Dependency Injection und Attrappen *A* isoliert testen zu können, ohne dass also das reale *B* mitgetestet wird.

An dieser Stelle kommt der neue Gedanke: Was würde passieren, wenn wir die Aspekte Logik und Integration nicht mehr vermischen? In Bild 4 ist eine Struktur zu sehen, die genau das zeigt.

Nun ist die Funktionseinheit *I* dafür zuständig, die beiden Funktionseinheiten *A* und *B* zu integrieren. *I* enthält selbst keine Logik. In diesem Fall ist das sinnvoll, weil *I* die Aufgabe hat, *A* und *B* in geeigneter Weise aufzurufen, zu integrieren. In *A* und *B* sind jetzt keine Aspekte mehr vermischt, weil beide ausschließlich Logik enthalten. Eine Funktionseinheit, die nur Logik enthält und keine andere Funktionseinheit aufruft, nennen wir Operation. Daraus resultiert das IOSP: Integration Operation Segregation Principle.

Doch ist damit nun tatsächlich das DIP überflüssig geworden? Stellt das IOSP tatsächlich eine Weiterentwicklung dar? Dazu müssen wir uns wiederum anschauen, wie es um die beiden Werte Korrektheit und Wandelbarkeit bestellt ist. Schließlich geht es nicht darum, dogmatisch irgendwelche Prinzipien einzuhalten, sondern darum, die Werte zu erreichen.

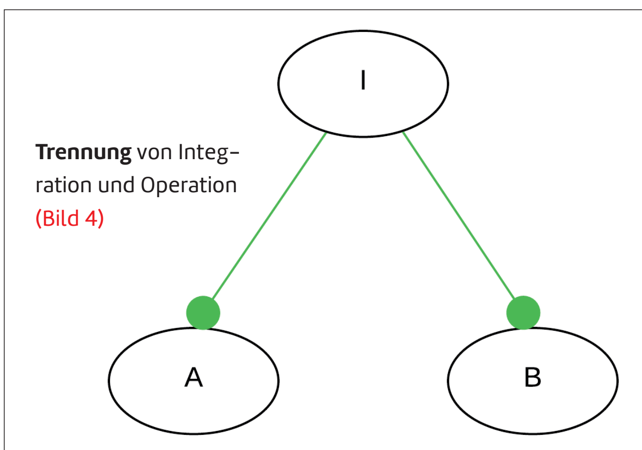


Vertikale Anordnung (Bild 3)

Wir unterscheiden bei automatisierten Tests zwischen Integrationstests und Unit-Tests. Ein Integrationstest testet mehrere Funktionseinheiten inklusive ihrer realen Abhängigkeiten. Für die Strukturen in Bild 1 und Bild 2 lässt sich leicht ein Integrationstest schreiben: Im Integrationstest wird *A* aufgerufen. Ob *A* dann unmittelbar oder mittels Kontrakt von *B* abhängt, spielt beim Integrationstest keine Rolle.

Das Gleiche wird mit der Struktur in Bild 4 erreicht. Hier wird *I* im Integrationstest aufgerufen. *I* ruft seinerseits *A* und *B* auf, somit wird auch hier die Integration der Funktionseinheiten getestet. Halten wir also fest: Integrationstests sind ohne DIP, mit DIP und mit IOSP möglich.

Kommen wir zu den Unit-Tests. Beginnen wir mit *B*: Ein Unit-Test ist leicht möglich, da *B* keine Abhängigkeiten hat. Spannend wird es bei *A*. Ist *A* direkt von *B* abhängig, siehe Bild 1, lässt sich kein Unit-Test für *A* schreiben. Daraus wurde vor vielen Jahren geschlossen, dass ein Interface notwendig ist. Tatsächlich hilft das DIP, weil bei der Struktur in Bild 2 ein Unit-Test für *A* möglich ist. Es muss dazu ein Interface erstellt, Dependency Injection eingeführt und im Test mit Attrappen gearbeitet werden.



Trennung von Integration und Operation (Bild 4)

Und nun zum IOSP. Bei der Struktur in Bild 4 lassen sich für *A* und *B* leicht Unit-Tests schreiben. Beide haben keine Abhängigkeiten. Können wir hier tatsächlich auf Interfaces zwischen *I* und *A* beziehungsweise *B* verzichten? Ja! Es gibt keine Logik in *I*. Somit gibt es auch keinen Grund, *I* isoliert mit einem Unit-Test zu testen. Ob die Integrationsaufgabe von *I* korrekt funktioniert, prüfen wir mit einem Integrationstest auf *I*.

Fazit

Ich kann es gar nicht deutlich genug betonen: Wenn der Aspekt der Integration herausgezogen wird, hier zur Funktionseinheit *I*, wird das Schreiben von Unit-Tests möglich, ohne dass das DIP benötigt wird. Insofern ist das IOSP eine Weiterentwicklung des DIP. Die Herausforderungen, die mit Abhängigkeiten verbunden sind, löst das IOSP dadurch, dass der Aspekt „Abhängigkeit" in nur dafür verantwortliche Methoden herausgezogen wird. Bei Anwendung des DIP bleiben die beiden Aspekte Logik und Integration vermischt. Die Probleme, die damit einhergehen, werden durch Interfaces abgemildert, letztlich aber nicht konsequent angegangen.

Vielleicht fragen Sie sich nun, ob damit IoC-Container, Interfaces, Mock-Frameworks et cetera überflüssig werden. Wir reden hierbei über Techniken oder Frameworks, die im Kontext von DIP hilfreich bis notwendig sind. Setzt man konsequent auf das IOSP, kann der Einsatz von Interfaces, Dependency Injection und Attrappen auf ein sehr geringes Maß reduziert werden. Am Ende bleibt eine geringe Zahl von Testfällen übrig, bei denen man trotz IOSP mit Attrappen testen möchte, etwa weil es eine Abhängigkeit von teuren Ressourcen gibt, die nicht ständig zur Verfügung stehen. Meiner Erfahrung nach lässt sich der Einsatz von Dependency Injection und Attrappen auf sehr wenige Testfälle reduzieren.

Wenn bestimmte Frameworks wie etwa ASP.NET Core auf Dependency Injection setzen und einen IoC-Container an Bord haben, spricht nichts grundsätzlich dagegen, diese Mechanismen einzusetzen. Allerdings sollte sich der Einsatz hier auf ein minimales Maß reduzieren lassen, an der Schnittstelle zwischen spezifischem Framework oder Infrastruktur und „dem Rest" der Anwendung. Im Kern eines Softwaresystems wird das DIP nicht benötigt, sondern kann vollständig durch das IOSP ersetzt werden. ■

Stefan Lieser
sucht ständig nach Verbesserung und neuen Wegen, um die innere Qualität von Software zu optimieren. Gemeinsam mit Ralf Westphal hat er die Clean Code Developer Initiative (https://clean-code-developer.de) ins Leben gerufen.
https://ccd-akademie.de

dnpCode A2201CleanCode