Stabile und evolvierbare Software durch Einhaltung der SOLID-Prinzipien

Markus Just*, Manfred Dausmann, Joachim Goll

Fakultät Informationstechnik der Hochschule Esslingen - University of Applied Sciences

Wintersemester 2015/2016

Softwaresysteme, die heutzutage entwickelt werden, sind sehr komplex. Bei ihrer Entwicklung muss alles getan werden, um Fehler zu vermeiden. Dies betrifft nicht nur die Programmierung, bei der man „defensiv" programmieren sollte, um das Auftreten von Fehlern möglichst zu verhindern. Auch der Entwurf eines Systems wird von Prinzipien geprägt, die eingehalten werden sollten, um ein stabiles Design[1] zu erreichen.
Entwurfsprinzipien verkörpern Erfahrung und gelten allgemein. Sie werden außer in der selbst zu erstellenden Software auch in Entwurfsmustern eingesetzt. Entwurfsprinzipien gewährleisten, dass eine Software gewisse Qualitätsmerkmale besitzt. So ist z. B. „Programmieren gegen eine Schnittstelle" ein Entwurfsprinzip, welches die Erweiterbarkeit und die Testbarkeit unterstützt. Aus diesem Grund wird dieses Prinzip auch in wiederholtem Maße bei Entwurfsmustern eingesetzt. Erweiterbarkeit durch Spezialisierung wird durch Einhalten des Liskovschen Substitutionsprinzips sichergestellt.
Aufgrund verschiedener Rahmenbedingungen wie Zeitdruck, Unerfahrenheit oder Nachlässigkeit werden Entwurfsprinzipien häufig verletzt. Um dies zu verhindern, ist es unerlässlich, den erstellten Quellcode manuell durch regelmäßige Reviews und durch die Nutzung diverser Tools zu prüfen.

Entwurfsziele für Entwurfsprinzipien

Durch den Einsatz von Entwurfsprinzipien sollen verschiedene Entwurfsmerkmale erreicht werden, die eine hohe Softwarequalität sicherstellen. Ziel ist es beispielsweise, **Verantwortlichkeiten der Softwaremodule zu trennen**, um Abhängigkeiten zu reduzieren. Außerdem soll die **Testbarkeit**, z. B. durch das Ermöglichen von Mocks, sowie **Korrektheit** gewährleistet sein. Auch **Evolvierbarkeit** spielt eine große Rolle. Evolvierbarkeit bedeutet nicht nur, dass das System **erweiterbar** ist, sondern dass die verschiedenen Komponenten auch im Rahmen anderer Architekturen **wiederverwendbar** sind.

Abhängigkeiten

Das Hauptproblem von Software ist in der Regel das Vorhandensein zu vieler Abhängigkeiten innerhalb eines Systems. Daher gilt es, diese Abhängigkeiten, soweit es möglich ist, zu eliminieren, was das Entstehen verworrener Strukturen und einer starken Kopplung der Komponenten verhindert. Da es nicht möglich ist, Abhängigkeiten komplett zu entfernen, muss es das Ziel beim Entwurf eines Systems sein, diese so gering wie möglich zu halten. Abhängigkeiten können innerhalb eines selbst erstellten Softwaresystems auftreten oder auch in Bezug auf die Umgebung, in der das System läuft. Die Abhängigkeit zwischen verschiedenen Komponenten wird auch als Kopplung dieser Komponenten bezeichnet. Wenn diese in einem System mit hoher Kopplung ausgetauscht werden, ist die Software möglicherweise nicht mehr lauffähig.

Die SOLID-Prinzipien

Robert C. Martin fasst eine wichtige Gruppe von Prinzipien zur Erzeugung wartbarer und erweiterbarer Software unter dem Begriff SOLID[2]-Prinzipien zusammen [1]. Dieser Begriff soll andeuten, dass die Verwendung dieser Prinzipien für das Schreiben hochwertiger Software unabdingbar ist.
SOLID steht für **S**ingle-Responsibility-Prinzip (SRP), **O**pen-Closed-Prinzip (OCP), **L**iskovsches Substitutionsprinzip (LSP), **I**nterface-Segregation-Prinzip (ISP) und **D**ependency-Inversion-Prinzip (DIP). Robert C. Martin erklärte diese Prinzipien zu den wichtigsten Entwurfsprinzipien. Die SOLID-Prinzipien selbst stammen aber nicht alle von Robert C. Martin, sondern im Falle des OCP von Bertrand Meyer und im Falle des LSP von Barbara Liskov. Im Folgenden werden die Bedeutungen und die zu erreichenden Entwurfsmerkmale der Prinzipien kurz vorgestellt.

Single Responsibility Prinzip

Das Single Responsibility Prinzip fordert, wie dem Namen zu entnehmen ist, dass Software-

*Diese Arbeit wurde durchgeführt bei der Firma IT-Designers GmbH, Esslingen
[1] Bei einem stabilen Design haben Änderungen am System keine unerwarteten Auswirkungen.
[2] Das Akronym SOLID wurde von Michael Feathers eingeführt.

Module nur eine einzige Verantwortlichkeit realisieren, also nur eine einzige Aufgabe erfüllen sollen. Verantwortlichkeit steht in diesem Kontext für den Grund einer Änderung. Kann es also in einem Software-Modul mehrere Gründe für eine Änderung geben, dann wird das SRP verletzt. Es soll eine **Trennung der Verantwortlichkeiten** und damit eine Reduktion der Abhängigkeiten erreicht werden. Das Single Responsibility Prinzip ist zwar ein einfach zu verstehendes Prinzip, wird in der Realität aber mit am häufigsten verletzt.

Open Closed Prinzip
Das Open Closed Prinzip fordert, dass Module offen für Erweiterungen und gleichzeitig geschlossen für Veränderungen sind. Offen für Erweiterungen bedeutet, dass ein SoftwareModul durch das Hinzufügen von Code erweitert werden kann, was durch die Nutzung von Abstraktionen erreicht werden kann. Geschlossenheit eines Moduls gegenüber Veränderungen heißt, dass ein Modul wiederverwendet werden kann, ohne den Code und die vorhandenen Schnittstellen anpassen zu müssen. Das Einhalten dieses Prinzips trägt zur **Wiederverwendbarkeit** und zur **Erweiterbarkeit** bei.

Liskovsches Substitutionsprinzip
Das Liskovsche Substituionsprinzip fordert von abgeleiteten Klassen im objektorientierten Entwurf, dass sie dasselbe Verhalten aufweisen wie ihre Basisklassen. Ein Client soll es also nicht merken, wenn an die Stelle des Objekts einer Basisklasse plötzlich ein Objekt einer abgeleiteten Klasse tritt. Dies kann auf verschieden Weisen realisiert werden. Bei einer reinen Erweiterung werden die Methoden einer Basisklasse nicht durch die abgeleiteten Klassen überschrieben. Gegenüber einem Client, der die Methoden der Basisklasse über ein Objekt der abgeleiteten Klasse nutzt, verhält sich ein Objekt der abgeleiteten Klasse genauso wie ein Objekt der Basisklasse. Sollte eine abgeleitete Klasse die Methoden ihrer Basisklasse überschreiben, müssen die Verträge der Basisklasse, also Vor- und Nachbedingungen der Methoden und Invarianten der Klasse, eingehalten werden, damit keine unerwünschten Nebeneffekte auftreten. Das LSP führt zu **Erweiterbarkeit** und **Korrektheit**.

Interface Segregation Prinzip
Das Interface Segregation Prinzip gibt vor, dass Interfaces ihren Clients nur Methodenschnittstellen anbieten sollen, die von diesen wirklich benötigt werden. "Fat"bzw. "wideÏnterfaces, das sind Interfaces, die eine hohe Anzahl an Methodenschnittstellen aufweisen, sollen also vermieden werden. Sollte ein Interface Methodenschnittstellen aufweisen, die nicht von allen Clients benötigt werden, dann muss analysiert werden, wie man die Interfaces sinnvoll aufteilen kann, so dass jeder Client nur die Schnittstellen bekommt, die er auch benötigt. Somit wirken sich Änderungen an diesen Schnittstellen nur auf die Clients aus, die auch davon Gebrauch machen, also werden unnötige **Abhängigkeiten** vermieden.

Dependency Inversion Prinzip
Das Dependency Inversion Prinzip fordert, dass Klassen einer höheren Ebene nicht direkt von Klassen niedriger Ebenen abhängen sollen, sondern beide sollen von Abstraktionen abhängen. Oftmals werden Klassen in Ebenen eingeteilt. Dabei beschreibt in einem hierarchischen Modell nach Grady Booch die oberste Ebene die Politik der Geschäftsprozesse, die Ebene darunter beschreibt die Mechanismen und die unterste beschreibt Hilfsdienste. Anstatt des direkten Zugriffs und damit der direkten Abhängigkeit einer Klasse der höheren Ebene von einer Klasse der niedrigeren Ebene soll eine Abstraktionsschicht eingeführt werden, damit die Klasse der höheren Ebene nur noch von einer Abstraktion der unteren Ebene abhängt. Dieses Prinzip führt zu einer **Reduktion von Abhängigkeiten** und zu einer **Erhöhung der Testbarkeit**, indem es die Möglichkeit bereitstellt, Mocks zu nutzen.

Bewertung
SOLID legt die Basis für das Design eines robusten, wartbaren, korrekten und langfristig erweiterbaren Softwaresystems. Damit kann die Lebensdauer und Wiederverwendbarkeit von Software erhöht werden. Software, die nach diesen Prinzipien entwickelt wird, lebt nicht nur länger, sondern ist auch infolge der Reduktion von Abhängigkeiten leicht testbar.
Es wird in mehrfachem Sinne durch SOLID Qualität erzeugt: Das Design verhindert das Altern der Software durch die Reduktion der Abhängigkeiten (Single Responsibiliy Principle, Interface Segregation Principle, Dependency Inversion Principle), erhöht die Testbarkeit durch Mocks (Dependency Inversion Principle), sorgt für Korrektheit (Liskovsches Substitutionsprinzip) und sorgt durch das Open-Closed-Prinzip für Stabilität und Erweiterbarkeit.

[1] Robert C. Martin: Agile Software Developement, Pearson Ptr, 2011