ESE Kongress 2014

Vortragsskript:

Die SOLID-Prinzipien
Fünf Grundsätze für bessere Software

Frank Listing, MicroConsult GmbH

Die Qualität der Software ist nicht in allen Projekten ideal, und deshalb werden in vielen Bereichen Anstrengungen unternommen, eine Verbesserung zu erreichen. Der Einsatz von Software Engineering soll den Code in allen seinen Aspekten verbessern.

Allerdings wird man immer wieder mit der Meinung „Warum sollte ich überhaupt guten Code schreiben? – Interessiert doch eh keinen“ konfrontiert. Erfahrungswert: Ein großer Teil der Programmierer schreibt schlechten Quellcode -> "Was willst Du? Es läuft doch!"

Der Preis schlechten Codes:
- Hoher Wartungsaufwand
- Hohe Kosten für die Weiterentwicklung
- Aufwändige Fehlersuche

ABER: Die interne Qualität des Codes trägt langfristig zu einer Reduzierung der Kosten bei.

Warum ist guter Code wichtig?

- Code schreiben ist ein relativ kleiner Teil der Softwareentwicklung. Ca. 80% der Kosten sind Wartungskosten es muss langfristiger gedacht werden.
- Es wird wenig neuer Code geschrieben. Die hauptsächliche Arbeit besteht aus Änderungen. Der größte Anteil an der Arbeit ist nicht das Codieren, sondern das Verstehen (Lesen) von Code.
- Fehlerbehebungen in unverständlichem Code erzeugen schnell neue Fehler.
- Wenn am Anfang des Projektes auf Kosten der Codequalität Zeit gespart wird, wird dies am Ende des Projektes ein Vielfaches der eingesparten Zeit kosten.

Prinzipiell ist jedem (erfahrenen) Entwickler bekannt, dass schlechter Code die Arbeit behindert. Allerdings passiert es immer wieder, dass durch hohen Druck chaotischer Code geschrieben wird, damit Termine eingehalten werden können.

ABER: Das funktioniert nicht. Der schlechte Code führt dazu, dass die Arbeit gleich langsamer wird und Termine nicht eingehalten werden. Es gibt nur einen Weg: Von Anfang an muss sauberer Code geschrieben werden!

Was ist sauberer Code?

- Sauberer Code ist gut zu lesen.
- Er kann auch von anderen Entwicklern verstanden werden.
- Klassen und Methoden sind auf die Erfüllung einer Aufgabe ausgerichtet und werden nicht durch Nebenaufgaben "verunreinigt".
- Die Abhängigkeiten zu anderem Code sind auf ein Minimum begrenzt.
- Sauberer Code ist gut zu testen.
- Es gibt keine Duplizierungen.
- Der Code enthält keine Überraschungen.

Guter Code motiviert

- Alle Beteiligten sind stolz auf ihre Arbeit.
- Das Programmieren macht mehr Spaß.
- Er enthält weniger Fehler.
- Guter Code ist einfacher zu testen.

Für den Mitarbeiter im Projekt heißt das: Guter Code reduziert unangenehme Arbeit.

Um die Erstellung guten Codes zu erleichtern, wurden mehrere Prinzipen für die Softwareentwicklung formuliert. Diese können als eine Art Checkliste gesehen werden, die in der täglichen Arbeit des Entwicklers als Hilfe dienen, die eigene Arbeit zu reflektieren bzw. von vornherein Fehler und kritische Konstrukte zu vermeiden.

Prominente Vertreter solcher Prinzipien sind die SOLID-Prinzipien. SOLID wurde von Robert C. Martin geprägt. Es ist ein Akronym und steht für:

Single-Responsibility-Prinzip
Open-Closed-Prinzip
Liskovsches Substitutionsprinzip
Interface-Segregation-Prinzip
Dependency-Inversion-Prinzip

Die Einhaltung dieser Prinzipien führt zu besserem Code und macht Software besser wartbar.

Single-Responsibility-Prinzip

Das Single-Responsibility-Prinzip besagt, dass eine Klasse nur eine Verantwortlichkeit haben soll. Änderungen an der Funktionalität sollen nur Auswirkungen auf wenige Klassen haben. Je mehr Code geändert werden muss, desto höher ist das Fehlerrisiko.

Hält man sich nicht an dieses Prinzip, führt das zu vielen Abhängigkeiten und hoher Vernetzung. Das ist wie im wirklichen Leben: Ab einer bestimmten Größe wird ein Universalwerkzeug unhandlich.

Wie kann erkannt werden, ob die Klasse mehr als eine Aufgabe erfüllt? – Die Klasse darf nur einen Grund zur Änderung haben. D.h. wenn sich zwei Anforderungen ändern, darf nur eine davon eine Auswirkung auf die Klasse haben. Hat die Klasse mehrere Änderungsgründe, erfüllt sie zu viele Aufgaben.

Es ist also besser, viele kleine Klassen zu haben, als wenige große. Der Code wird dadurch nicht umfangreicher. Er wird nur anders organisiert. Analogie aus dem Bastelkeller: Wenn alle Schrauben in einer Kiste sind, ist es schwer, die Richtige zu finden. Sind sie in mehrere Schachteln sortiert, geht die Suche viel schneller. Genauso verhält es sich mit den Klassen.

Open-Closed-Prinzip

Nach dem Open-Closed-Prinzip soll eine Klasse offen für Erweiterungen, aber geschlossen gegenüber Modifikationen sein. Das Verhalten einer Klasse darf erweitert aber nicht verändert werden. Dieses Prinzip hilft, Fehler in schon fertigen Codeteilen zu vermeiden. Wenn eine Erweiterung nur durch Änderungen innerhalb einer Klasse erreicht werden kann, ist die Gefahr, dass durch die Änderung schon fertig implementierte Funktionen neue Fehler bekommen, sehr groß.

Das Open-Closed-Prinzip kann normalerweise über zwei Wege erreicht werden:

- Vererbung
- Einsatz von Interfaces

Durch die Einhaltung dieses Prinzips können einer Applikation neue Funktionen hinzugefügt werden, ohne bestehende Klassen zu verändern.

Schon in der (nicht objektorientierten) C-Bibliothek finden sich Beispiele für dieses Prinzip, z.B. die Implementierung der Quick-Sort-Funktion:

```
void qsort(void *base, size_t num, size_t size,
      int (*comparer)(void *element1, void *element2));
```

Um einen eigenen Datentypen sortieren zu können, muss nicht die qsort-Funktion umgeschrieben werden. Der Algorithmus bleibt immer gleich. Der Anwender muss dem Algorithmus lediglich seine eigene Vergleichsfunktion übergeben. Damit wird eine Erweiterbarkeit erreicht, ohne den Algorithmus verändern zu müssen.

Liskovsches Substitutionsprinzip

Das Liskovsche Substitutionsprinzip fordert, dass abgeleitete Klassen immer anstelle ihrer Basisklasse einsetzbar sein müssen. Subtypen müssen sich so verhalten wie ihr Basistyp. Das klingt selbstverständlich, ist es das aber auch? Der Compiler weiß, dass eine abgeleitete Klasse auch vom Typ der Basisklasse ist – also in diese immer konvertiert werden kann. Ist das ausreichend? Das Liskovsche Substitutionsprinzip geht weiter als der Compiler.

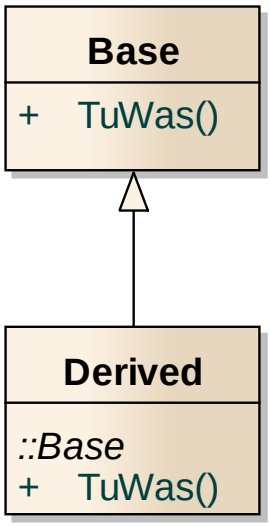

Bild 1: Liskovsches Substitutionsprinzip

Wenn eine Methode einen Parameter vom Typ Base (aus Bild 1) erwartet, z.B. `public void TuWasTolles(Base b)`, darf es keinen Unterschied machen, ob ein Objekt vom Typ `Base` oder vom abgeleiteten Typ `Derived` übergeben wird. In der Klasse `Derived` muss sichergestellt werden, dass das Verhalten aus Sicht der Methode `TuWasTolles()` genauso ist wie bei der Klasse `Base`. Wenn z.B. die Methode der Basisklasse keine Exceptions wirft, darf auch die Methode der abgeleiteten Klasse keine Exceptions werfen. Eine abgeleitete Klasse darf ihre Basisklasse erweitern, aber nicht einschränken oder verändern. Leider wird dieses Prinzip oft missachtet. Unit-Tests können sicherstellen, dass nicht versehentlich ein Verstoß gegen das Liskovsche Substitutionsprinzip in die Software eingebaut wird.

Interface-Segregation-Prinzip

Das Interface-Segregation-Prinzip besagt, dass ein Client nicht von den Funktionen eines Servers abhängig sein darf, die er gar nicht benötigt. D.h. dass ein Interface nur die Funktionen enthalten darf, die auch wirklich eng zusammengehören. Die Problematik ist, dass durch "fette" Interfaces Kopplungen zwischen den ansonsten unabhängigen Clients entstehen.

Wird ein Aspekt des Interfaces verändert, hat das Auswirkung auf alle Clients – selbst wenn sie diesen Aspekt nicht nutzen. Ein anschauliches Beispiel liefert hier die AWT-Bibliothek von Java: Soll lediglich auf das Ereignis zum Schließen des Fensters reagiert werden, müssen alle Methoden des Interfaces `WindowListener` implementiert werden (Bild 2).

```
public class Fenster extends Frame implements WindowListener
{
    ...

    // Methoden von WindowListener
    public void windowClosing(WindowEvent event)
    {
        System.exit(0);
    }
    public void windowClosed(WindowEvent event){}
    public void windowDeiconified(WindowEvent event){}
    public void windowIconified(WindowEvent event){}
    public void windowActivated(WindowEvent event){}
    public void windowDeactivated(WindowEvent event){}
    public void windowOpened(WindowEvent event){}
}
```

Bild 2: WindowListener

Was kann getan werden, wenn so ein Interface vorliegt und nicht geändert werden kann? Es kann ein Adapter eingesetzt werden (Adapter-Entwurfsmuster). Dieser Adapter implementiert alle Methoden des Interfaces mit einer Dummy-Implementierung und stellt diese virtuell zur Verfügung. Auch hier dient die AWT-Bibliothek von Java als Beispiel. Diese stellt solch einen Adapter für das vorher gezeigte Beispiel zur Verfügung.

```
class MyWindowListener extends WindowAdapter
{
    public void windowClosing(WindowEvent event)
    {
        System.exit(0);
    }
}
```

Bild 3: Adapter

Dependency-Inversion-Prinzip

Das Dependency-Inversion-Prinzip besagt, dass Klassen auf einem höheren Abstraktionslevel nicht von Klassen auf einem niedrigen Abstraktionslevel abhängig sein sollen. Dabei geht es aber nicht darum, die Abhängigkeiten einfach umzudrehen. Abhängigkeiten zwischen Klassen soll es nicht mehr geben; es sollen nur noch Abhängigkeiten zu Interfaces bestehen (beidseitig). Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces. Beispiel: Die Klassen in Bild 4 sind zu stark miteinander verkoppelt.

Die Abhängigkeiten sind so stark, dass ohne Codeänderungen der separate Test einer Klasse nicht möglich ist. Auch Änderungen in den Anforderungen sind durch diese starke Kopplung schwerer umzusetzen.

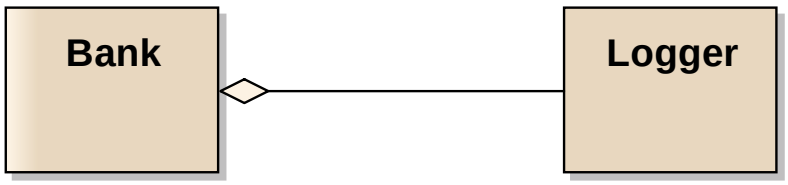

Bild 4: Starke Kopplung

Lösungsvorschlag 1 – Konstruktorparameter

Aggregation wird durch Assoziation ersetzt und die Abhängigkeit von einer speziellen Klasse in die Abhängigkeit zu einem Interface geändert (Bild 5). Das konkrete Objekt (der Klasse `Logger`) wird als Parameter an den Konstruktor der Klasse `Bank` übergeben. Das ist eine sehr einfache, aber nur bedingt flexible Lösung.

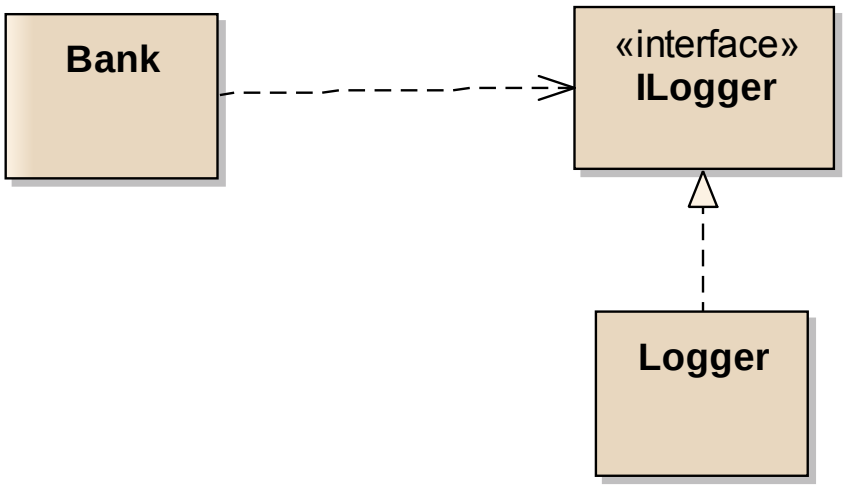

Bild 5: Schwache Kopplung durch Assoziation

Lösungsvorschlag 2 – IoC-Container (IoC: Inversion of Control)

Aggregation wird durch Assoziation ersetzt und die Abhängigkeit von einer speziellen Klasse in die Abhängigkeit zu einem Interface geändert. Die konkrete Implementierung registriert sich beim IoC-Container (Bild 6). Der Nutzer (die Klasse `Bank`) fragt im IoC-Container nach einer Implementierung des benutzten Interfaces (hier `ILogger`).

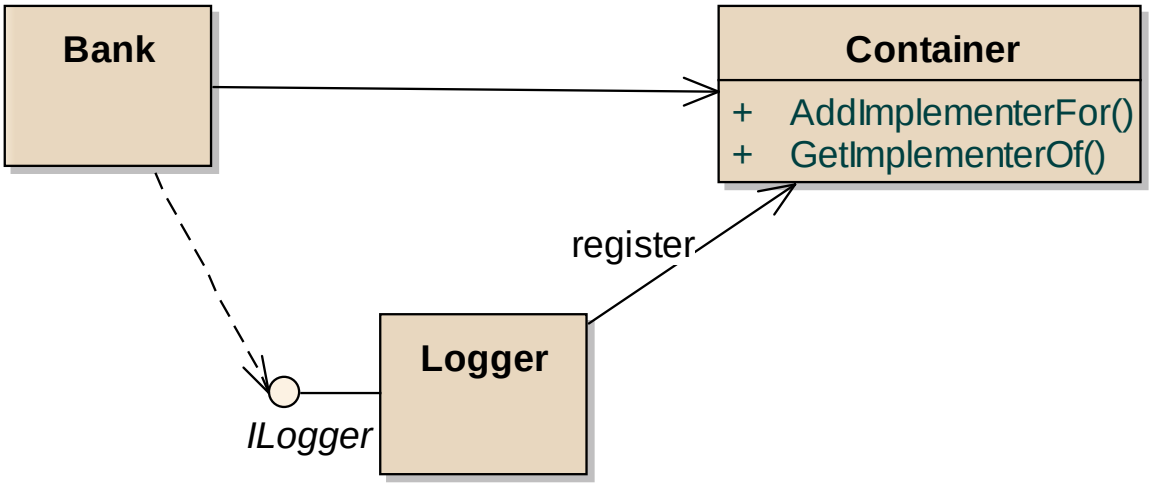

Bild 6: Nutzung eines IoC-Containers

Fazit

Es gibt immer eine Ausrede, warum man gerade keinen guten Code schreiben kann. Es wird aber immer nur eine Ausrede bleiben. Einen triftigen Grund, schlechten Code zu schreiben, gibt es nicht.

Die hier gezeigten Prinzipien sind Hinweise, die es einem Entwickler erleichtern, im Alltag die Codequalität zu verbessern. Die (kleine) Mühe amortisiert sich sehr schnell. Änderungen werden einfacher, und auch Test und Fehlersuche werden beschleunigt.

Quellen

Robert C. Martin, „Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code“, mitp

Autor

Dipl.-Ing. Frank Listing ist seit 2002 Trainer und Projektcoach bei der MicroConsult GmbH mit dem Schwerpunkt Microsoft-Plattformen, objektorientierte Programmierung und Testen von Embedded-Systemen und u.a. fachlich für das Thema .NET verantwortlich. Sein Wissen gibt er immer wieder auch in Publikationen und Fachvorträgen weiter.