TIME-OUT

Warum nicht gleich so?

Nachhaltig coden mit einem Styleguide, der als lebendiges Dokument gepflegt wird.

Bestimmt kennen Sie das: Sie schauen über fremden Code und Ihnen fällt sofort das ein oder andere auf, was Sie anders machen würden. Was man vereinfachen könnte. Oder was unverständlich ist und eigentlich umgeschrieben gehört. Weil es sich nicht gut anfühlt.

Nachhaltiger allerdings, als solchen bestehenden Code zu optimieren, ist es, schon viel früher anzusetzen und es gar nicht erst so weit kommen zu lassen. Indem man Kriterien definiert, die festlegen, wie denn Code auszusehen habe, und sich daran hält. In der ganzen Abteilung. Doch das ist leichter gesagt als getan. Zum einen gibt es Leute, die nach einem *If* keine geschweiften Klammern setzen, wenn die folgende Anweisung nur aus einer Zeile besteht, und andere, die das auf keinen Fall ohne Klammern wollen. Es gibt diejenigen, die immer *var* schreiben, und welche, die es niemals oder nur sporadisch verwenden. Wie soll man hier einen Standard schaffen? Und ein noch schwerwiegenderes Argument: Ein Styleguide kann niemals umfassend sein. Zum einen gibt es immer Grenzbereiche, und zum anderen ist das ganze Unterfangen so umfangreich, dass man niemals damit fertig würde.

Hier kommt eine klassische Weisheit ins Spiel: Dass man eine Aufgabe nicht beginnt, bei der man kein Ende sieht. Man will ja nicht zum Sisyphos werden, der den Stein immer wieder hochschleppt und ihn dann runterrollen sieht.

Aber ist es mit Software nicht genau so? Dass sie niemals fertig ist? Und dennoch beginnen wir damit. Weil wir ein Tool haben, mit dem wir Zwischenschritte verwalten können: GIT oder SVN. Und weil wir agil unterwegs sind und schon bei kleinen Schritten Mehrwerte für den Kunden schaffen können. Und so könnte man es mit einem Styleguide auch machen. Man könnte ein GIT-Projekt anlegen, bestehend aus Beispielcode und einer Anleitung, die die Prinzipien für guten Code erläutert. Änderungen der Kollegen bekommt man per E-Mail-Benachrichtigung mit. Kann sie nachvollziehen. Nach einer kurzen Teambesprechung werden sie eingecheckt. Ein lebendes Dokument, an dem alle aktiv mitarbeiten. Das man immer wieder als Nachschlagewerk verwenden kann.

Aber damit haben wir das Problem der Widersprüche noch nicht aufgelöst. Schreiben wir *var* – oder eben nicht? Setzen wir Klammern?

Hier ist es sinnvoll, wenn man sich überlegt, warum man Code optimiert. Was für Zwecke sollen damit erreicht werden? Es geht ja nicht nur um Schönheit, von der kann man bekanntlich nicht leben. Es geht um wirtschaftliche Prinzipien, denen man folgt. Für mich sind das drei Elemente:

- Wiederverwertbarkeit,
- Fehlerfreiheit und
- Lesbarkeit.

Diese Ziele, die uns effektiver machen, sollen erreicht werden. Lassen Sie mich das kurz erläutern.

Beispiel Wiederverwertbarkeit: Es ist schneller und sicherer, existierenden Code zu verwenden, der getestet ist und funktioniert, als ihn neu zu schreiben. Als Entwickler neigt man dazu, seine eigenen Fähigkeiten zu überschätzen, den Zeitaufwand für die Neuentwicklung zu unterschätzen und einige Fehlermöglichkeiten, die im bestehenden Code bereits berücksichtigt sind, zu übersehen.

Daraus leiten sich einige weiterführende Prinzipien ab. Damit man Code in anderen Projekten wiederverwenden kann, müssen einige Kriterien erfüllt sein. Der Code darf keine Voraussetzungen oder Bedingungen enthalten, die in anderen Projekten nicht nötig sind. (Beispiel: Wenn ich einen Betrag von Euro in US-Dollar umrechnen möchte, soll das möglich sein, ohne dass ich vorher eine Datenbankverbindung angeben muss. Denn eine andere Komponente könnte die Kurse ja über ein API bekommen.)

Eine Methode sollte generisch sein, das bedeutet, dass sie beispielsweise nicht *CalcEuro2Dollar* heißen darf, sondern *CalcWaehrung* mit den Parametern der Währungen. Die Methode sollte in einer eigenen Komponente gekapselt sein, damit man sie eben als NuGet oder DLL dazuladen kann. Sie soll ein Interface implementieren, das in einer eigenen DLL liegt. Damit kann zu Testzwecken mit einer Mockup-Methode gearbeitet werden. Und so weiter.

Oder Lesbarkeit. Man „liest" Code, um zu verstehen, was passiert, und um zu kontrollieren, ob man an alles gedacht hat. Die Lesbarkeit von Code ist ein erstrebenswertes Ziel. Dazu sind zwei Punkte nötig: Der Code sollte kompakt sein. Also nicht zwei Leerzeilen nach jeder Zeile aufweisen oder Zeilen haben, die über den Bildschirm hinausgehen, sodass man beim Lesen nicht scrollen muss. Man sollte ihn schnell und einfach lesen können. Dazu ist es zum Beispiel hilfreich, wenn alle Variablenname an der gleichen Stelle beginnen, weil immer *var* davorsteht.

Und die Variablen und Methoden müssen selbsterklärend sein. Man sollte also Kommentar nur in den seltensten Fällen brauchen, da sich alles aus der richtigen Benennung der Variablen und Methoden ergibt.

Zuletzt geht es um Fehlerfreiheit. Fehler entstehen zum Beispiel, weil man falsche Annahmen macht. Beispielsweise geht man davon aus, dass die Variable einen Wert enthält oder das Datum gefüllt ist. Aber: Guter Code zweifelt (fast) alles an. Hat die Variable einen Wert, kann der ja auch außerhalb des Gültigkeitsbereichs liegen. Eine gute Anwendung funktioniert auch bei Falscheingaben. Daraus leiten sich wieder weitere Prinzipien ab.

Sie sehen, es ist gar nicht so schwer, erst einmal anzufangen. Wenn man dann noch die Last auf mehrere Entwickler verteilt, die aktiv mitschreiben, geht es schneller, macht mehr Spaß und ist umfassender. Wie echte Software eben auch.

In diesem Sinne: Fangen Sie an! Schreiben Sie einen Styleguide. Und lassen Sie ihn zu einem lebendigen Dokument werden! Happy Coding! ■

Time-out End

Dipl.-Theol. Bernhard Pichler
war 20 Jahre lang Geschäftsführer der Softwarefirma informare, bevor er 2020 zum Leiter Entwicklung und Produktmanagement der DPS BS ernannt wurde. Die DPS BS ist Deutschlands größter Softwarepartner für betriebswirtschaftliche Software für den Mittelstand.

dnpCode A2311Timeout

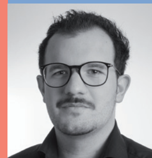
DEV ACADEMY

HANDS-ON-WORKSHOPS UND WEITERBILDUNG FÜR SOFTWARE-ENTWICKLER UND -ARCHITEKTEN

TRAININGS

Refactoring und Unit-Testing
Trainer: David Tielke

Async & Await
Trainer: Christian Giesswein



Kritische Backend-Architekturen
Trainer: Patrick Schnell

Technische Schulden
Trainer: Stefan Mintert

Azure DevOps – CI/CD mit Azure Pipelines und Git
Trainer: Thomas Tomow

Gruppen zu Teams machen
Trainer: Janosch Felde

UX- und UI-Design für Entwickler
Trainer: Sebastian Seidel

REMOTE TRAININGS MÖGLICH!

developer-academy.de

Ihre Ansprechpartnerin: **Susanne Herl**
+49 731 88005-8835 • susanne.herl@ebnermedia.de