Andreas Schlapsi
(andreas@schlapsi.net)
ist ein Softwareentwickler aus Wien mit langjähriger Berufserfahrung. Seine Schwerpunkte sind agile Entwicklungsmethoden, Software-Craftsmanship, Softwarearchitektur sowie objektorientierte und funktionale Programmierung.

Technische Schulden

Das Thema „Technische Schulden" interessiert mich schon seit längerer Zeit. Viele interessante Blogartikel sind schon über dieses Thema erschienen. Dieser Artikel bietet eine kurze Einführung in diese Materie.

Was sind technische Schulden?

Ward Cunningham hat in den 90er-Jahren die Metapher „Technische Schulden" (engl. Technical Debt) geprägt. In einer Videobotschaft (vgl. [WEDP]) erklärt er den Ursprung dieser Metapher.

Um einen kurzfristigen Vorteil zu erlangen, z. B. um ein Produkt schneller auf den Markt zu bringen, muss manchmal eine Entscheidung getroffen werden, die für die langfristige Entwicklung der Software nicht optimal ist. Solche Entscheidungen haben Folgen, die Ward Cunningham mit den Konsequenzen verglich, die durch Schulden entstehen.

Wenn man Schulden hat, muss man Zinsen zahlen. In der Softwareentwicklung bedeutet das, dass durch ein suboptimales Design Änderungen schwieriger sind und somit länger dauern. Der Aufwand, um neue Features einzubauen, wächst.

Um die Zinslast zu senken, kann der Schuldner einen Teil seiner Schulden zurückzahlen. Der Softwareentwickler refaktorisiert und verbessert so das Design der Software. Änderungen sollten dann einfacher sein. Der Softwareentwickler kann neue Features wieder schneller einbauen.

James Shore erzählt von einem Projekt (vgl. [Sho06]), für das er und sein Kollege technische Schulden aufgenommen haben, um einen Demotermin auf einer Konferenz zu halten. James Shore schätzt, dass die technischen Schulden einen Zusatzaufwand von einem Monat (also „Zinsen") für einen relativ kurzen Zeitraum von wenigen Wochen gekostet haben. Es dürfte sich gelohnt haben. Sie konnten die Demo erfolgreich präsentieren und sie haben danach ihre „Schulden" zurückgezahlt.

Ausweitung der Metapher

Die Metapher wurde im Laufe der Jahre für alle Designprobleme angewandt, sogar für solche, die „unabsichtlich" in die Software gelangten. Die Auswirkungen und die Lösung sind ja gleich.

Robert C. Martin hat einen Blogpost (vgl. [Mara]) geschrieben, in dem er zwischen technischen Schulden (technical debt) und „Schlamassel" (mess) unterscheidet. Ein Schlamassel ist immer eine Folge von Faulheit und Unprofessionalität, während die Entscheidung, technische Schulden zu machen, immer **bewusst** und **nach Abwägung** der Vor- und Nachteile getroffen wird.

Das Vier-Quadranten-Modell

Martin Fowler meint, dass es nur um eine Metapher (vgl. [Fow09]) geht und stellt die Frage, ob diese Metapher auch hilfreich ist, wenn man allgemeine Designprobleme im Programmcode hat. Er teilt die Ursache technischer Schulden in vier Quadranten ein:

- umsichtig-überlegt (ursprüngliche Bedeutung des Begriffs „technische Schulden")
- umsichtig-versehentlich („Jetzt wissen wir, was wir hätten tun sollen.")
- waghalsig-überlegt („Wir haben keine Zeit für Design!")
- waghalsig-versehentlich („Was heißt Layering?")

Robert C. Martin stimmt dieser Systematik prinzipiell zu (vgl. [Marb]). Er gibt aber zu bedenken, dass einige, die sich eigentlich im Quadranten „waghalsig-überlegt" befinden, sich im Quadranten „umsichtig-überlegt" wähnen könnten.

Das Vier-Quadranten-Modell zeigt uns also, wie gut wir unsere technischen Schulden kennen und ob die Gründe dafür gut gewählt sind.

Designen oder nicht designen? Das ist hier die Frage.

Wie viel technische Schulden kann man aufnehmen? Martin Fowler hat dazu die

Hypothese der Designwiderstandskraft (Design Stamina Hypothesis) aufgestellt (vgl. [Fow07]). Ohne Designaktivitäten kann man zu Beginn eines Projekts tatsächlich Zeit sparen. Aber je mehr Features implementiert werden, desto langsamer lässt sich die Software weiterentwickeln.

Mit gutem Design sorgt man dafür, dass die Entwicklungsgeschwindigkeit annähernd konstant bleibt. Stellt man diesen Zusammenhang in einem Diagramm dar, ergibt sich ein Punkt, an dem sich die zwei Linien treffen. Die Anzahl der im Code akkumulierten Features an diesem Punkt kann man als horizontale Linie darstellen, der sogenannten „Designpayofflinie".

Bis zur „Designpayofflinie" können technische Schulden problemlos aufgenommen werden, darüber ist man mit dem sauberen Design besser dran. Leider lässt sich diese Hypothese nicht beweisen, aber sie erklärt, was viele Softwareentwickler beobachten.

Wie findet man heraus, wo die „Designpayofflinie" ist? Die Produktivität eines Softwareentwicklers lässt sich leider nicht messen und daher gibt es dazu verschiedene Ansichten (vgl. [Fow03]). Ich teile übrigens die Meinung von Martin Fowler, dass die Linie viel niedriger ist, als viele annehmen.

Für Startups kann es besser sein, zu Beginn weniger auf das Design zu achten und technische Schulden aufzunehmen. Wichtig ist dabei, rechtzeitig die Schulden zurückzuzahlen. Kent Beck hat die Phasen eines Startups mit den Phasen beim Starten eines Flugzeugs verglichen (vgl. [TRIa]). Er erklärt auch den Zusammenhang zwischen diesen Phasen und dem Softwaredesign (vgl. [TRIb]).

Was also tun? Die Schulden zurückzahlen oder mit den Zinsen leben?

Es gibt zwei Möglichkeiten, mit technischen Schulden umzugehen. Man kann durch Refaktorisieren Schulden zurückzahlen oder mit den erhöhten Kosten in der Softwareentwicklung leben. Wenn sich die technischen Schulden in Code befinden, der selten geändert wird, zahlt es sich wahrscheinlich nicht aus, viel Zeit ins Refaktorisieren zu investieren.

Wenn die höheren Kosten ein Problem sind, kann es manchmal notwendig sein, diese Kosten sichtbar zu machen. Johanna Rothman hat User Stories verwendet, bei denen der Ertrag der einzelnen Story in Form von langfristiger Zeitersparnis ausgedrückt wurde, um die Reduktion technischer Schulden in der Iterationsplanung berücksichtigen zu können (vgl. [Rot09]).

Was können Softwareentwickler davon lernen, wie Unternehmen mit ihren finanziellen Schulden umgehen? Steve McConell schlägt in seinem Artikel (vgl. [McD]) vor, technische Schulden in Kategorien, wie z. B. Überziehungskredite oder Festzinskredite, einzuteilen und erläutert dementsprechende Strategien für deren Umgang. Die Art der technischen Schulden entscheidet darüber, wie man damit umgehen sollte.

Aaron Erickson (vgl. [Eri09]) verglich den Umgang der Softwareindustrie mit technischen Schulden mit den Praktiken, die die Firma Enron (vgl. [Enr]) für ihre Bilanzfälschungen verwendet hat.

Kritik an der Metapher

Manche sehen technische Schulden als einen negativ besetzten Begriff und lenken die Aufmerksamkeit auf die positiven Aspekte des Programmcodes. Eine interessante Idee von Michael Feathers ist, den Code als Kapital bzw. Vermögen zu sehen (vgl. [Fea09]). Eine interessante Idee ist dabei, den Code als Kapital bzw. Vermögen zu betrachten, wie es Michael Feathers und Chris McMahon vorschlagen (vgl. [McM08]).

Chris McMahon hatte eine ähnliche Idee. Brian Marick baute diese Idee aus und verglich den Programmcode mit einem Garten, der gepflegt werden muss (vgl. [McM08]).

Interessant sind auch die Eigenschaften, die Teams mit niedrigen technischen Schulden auszeichnen, wie z. B. Mitglieder, die sich besonders um Codequalität kümmern, regelmäßige teamweite Gespräche darüber, wie der Code sich „anfühlt" und einige weitere (vgl. [ETE08]).

Fazit

Technische Schulden sind eine Metapher für den Umgang mit Designmängel in Software. Sie zeigen, wie sich Designprobleme auswirken.

Das Vier-Quadranten-Modell kategorisiert die Ursachen. Anhand dieses Modells lassen sich Gründe für die Aufnahme technischer Schulden ableiten.

Es gibt Situationen, in denen die Aufnahme technischer Schulden die bessere Wahl ist. Aber man darf niemals vergessen, diese Schulden wieder zurückzuzahlen. Die Kosten der Softwareentwicklung steigen, wenn dies vergessen wird.

Um technische Schulden zurückzuzahlen gibt es vor allem eine Möglichkeit: Refaktorisierung. Manchmal ist es notwendig, die Kosten der technischen Schulden sichtbar zu machen, z. B. durch Gegenüberstellung der Kosten des Refaktorisierens und der Kosten bei Beibehaltung des Status quo.

Neuere Ideen sehen den Programmcode in einem positiveren Licht. Sie lenken die Aufmerksamkeit von den Problemen darauf, dass der Programmcode einen Wert darstellt, auf den geachtet und der gepflegt werden muss. ■

Literatur

[WEDP] http://c2.com/cgi/wiki?WardExplainsDebtMetaphor

[Sho06] http://www.jamesshore.com/Blog/CardMeeting/Voluntary-Technical-Debt.html

[Mara] https://web.archive.org/web/20121025014559/http://blog.objectmentor.com/articles/2009/09/22/a-mess-is-not-a-technical-debtl

[Marb] https://web.archive.org/web/20121020042752/http://blog.objectmentor.com/articles/2009/10/15/we-must-ship-now-and-deal-with-consequences

[Fow07] http://martinfowler.com/bliki/DesignStaminaHypothesis.html

[Fow03] http://martinfowler.com/bliki/CannotMeasureProductivity.html

[TRIa] http://www.threeriversinstitute.org/blog/?p=251

[TRIb] http://www.threeriversinstitute.org/blog/?p=338

[Rot09] http://www.jrothman.com/blog/mpd/2009/10/expressing-technical-debt-as-user-stories-helps-with-roi.html

[McD] http://www.construx.com/10x_Software_Development/Technical_Debt/

[Eri09] http://www.informit.com/articles/article.aspx?p=1401640

[Enr] http://de.wikipedia.org/wiki/Enron

[Fea08] http://michaelfeathers.typepad.com/michael_feathers_blog/2008/08/beyond-technica.html

[McM08] http://chrismcmahonsblog.blogspot.de/2008/06/technical-investment.html

[ETE08] http://www.exampler.com/blog/2008/08/15/whats-special-about-teams-with-low-technical-debt-2/