Technische Schulden auf den Kopf gestellt

Von technischen Schulden zu Investitionen in Softwarequalität

von Wolfgang Werner

Am 8. Juli 2015 musste der Handel an der New Yorker Börse für einen halben Tag aufgrund technischer Probleme eingestellt werden[1] – die längste ungeplante Unterbrechung in der Geschichte der NYSE. Am gleichen Tag war die Website des Wall Street Journals zeitweise nicht erreichbar.[2] Und ebenfalls am 8. Juli musste United Airlines wegen Problemen in der Netzwerkkommunikation 3500 Flüge streichen.[3]

Diese Vorfälle wurden nicht durch Angriffe von außen ausgelöst. In allen drei Fällen waren die Ursache Fehler in Softwaresystemen oder deren Konfiguration. Die NYSE wurde durch ein Konfigurationsproblem nach einem Update lahmgelegt,[4] bei United Airlines löste ein defekter Router eine Kettenreaktion aus, die mehrere Systeme (darunter das Reservierungssystem) ausfallen ließ,[5] und die Server des Wall Street Journals waren schlicht und einfach überlastet – die Probleme der New Yorker Börse haben wohl zu einer deutlich höheren Anzahl von Anfragen geführt.

Bemerkenswert ist vor allem das zeitliche Zusammenfallen der Ausfälle – dabei sind Störungen von etwas kleinerem Ausmaß längst keine Seltenheit mehr. Ein Grund für die Häufung solcher und ähnlicher Probleme ist, dass große Systeme, die den Betriebsablauf von Konzernen steuern, inzwischen ein relativ hohes Alter erreicht

1 http://www.zeit.de/wirtschaft/boerse/2015-07/nyse-handel-unterbrechung-technische-probleme

2 http://www.ibtimes.com/wall-street-journal-homepage-wsjcom-down-nyse-stops-trading-computer-glitch-1999756

3 http://www.nbcnews.com/business/travel/united-airlines-passengers-say-flights-grounded-nationwide-n388536

4 http://www.theguardian.com/business/live/2015/jul/08/new-york-stock-exchange-wall-street

5 http://money.cnn.com/2015/07/08/news/companies/united-flights-grounded-computer/?iid=EL

haben. Diese Systeme werden nur selten in Gänze ausgetauscht oder rundum erneuert. Neue Funktionalität wird in zusätzlichen Schichten über bestehenden Systemen abgebildet, Fehler werden oft umgangen statt behoben, und geänderte Anforderungen werden unter hohem Zeitdruck umgesetzt. Dies sind nur einige der Gründe, die zur Erosion von Softwaresystemen führen.

Höchste Zeit also, sich strukturiert mit Softwarequalität auseinanderzusetzen. Der vorliegende Artikel vermittelt einige Grundgedanken zur Qualität von Software und bietet einige Metaphern, die Überlegungen zu diesem und die Kommunikation über dieses Thema vereinfachen.

Interne und externe Qualität

In diesem Zusammenhang ist es wichtig, zwischen interner und externer Qualität von Software zu unterscheiden. Die externe Qualität ist für den Nutzer der Software sichtbar, die interne Qualität nur für Entwickler und – wenn auch in etwas geringerem Maße – für Betreiber des Systems.

Externe Qualität beschreibt das Maß, in dem Software in der Lage ist, die ihr zugedachten Aufgaben effizient und fehlerfrei auszuführen. Folgende Eigenschaften sind externen Qualitätsanforderungen zugeordnet.[6, 7] Das System …

- erfüllt die Erwartungen des Nutzers.
- arbeitet zuverlässig und genau.
- ist komfortabel zu bedienen.
- arbeitet robust und ohne Ausfälle.
- ist einfach an geänderte Anforderungen anpassbar.

Externe Qualität kann durch Integrations-, User-Acceptance-, Regressions- und Performance-Tests, im Idealfall automatisiert, überwacht werden. Ein hoher Grad externer Qualität zeigt, dass das richtige System implementiert wurde. Investitionen in externe Qualität zahlen sich unmittelbar aus, da der unterstützte Geschäftsprozess direkt optimiert wird.

Interne Qualität ist für einen Nutzer oder externen Beobachter des Systems nicht wahrnehmbar; sie ist ausschließlich für Entwickler und Architekten sichtbar, die an dem System arbeiten. Hier bietet sich der bekannte Vergleich mit einem Eisberg an: er liegt größtenteils unter Wasser (interne Qualität), während seine sichtbare Spitze nur ein Achtel der Gesamtmasse ausmacht (externe Qualität).

Interne Qualität umfasst alle wünschenswerten Eigenschaften, die die effiziente Implementierung, Wartung und Weiterentwicklung eines Systems unterstützen.

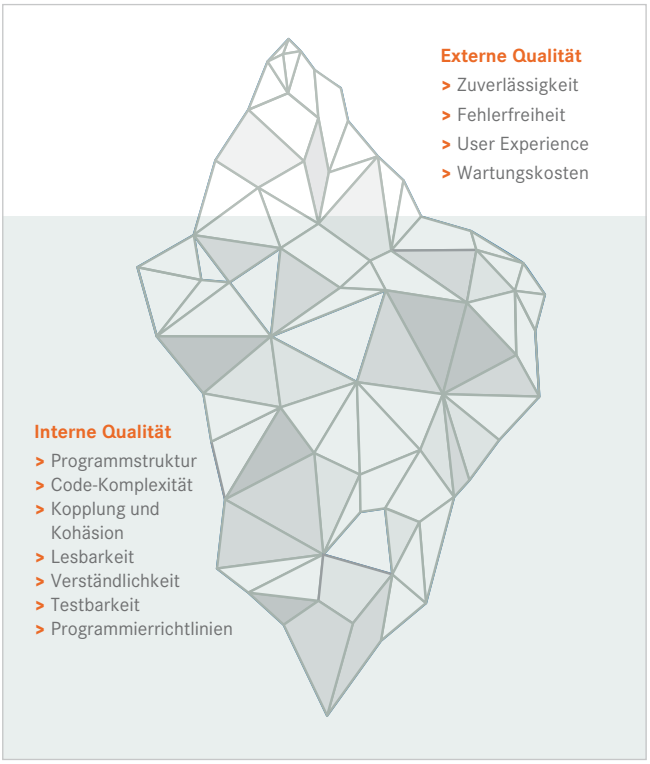

Abbildung 1: Interne Softwarequalität – für Nutzer nicht wahrnehmbar

6 http://c2.com/cgi/wiki?InternalAndExternalQuality

7 McConnell, Steve (1993), Code Complete (First ed.), Microsoft Press.

- **Wartbarkeit** – Auftretende Fehler sind einfach zu beheben und benötigte Anpassungen einfach und risikoarm durchzuführen.
- **Verständlichkeit** – Die Funktion der Anwendung ist aus dem Source Code klar erkennbar; neue Teammitglieder werden schnell produktiv.
- **Testbarkeit** – Module der Anwendung können einfach automatisiert getestet werden.
- **Flexibilität** – Das System lässt sich aufwands- und risikoarm auf geänderte Rahmenbedingungen anpassen.
- **Einfachheit** – Einzelne Module/Klassen/Methoden der Anwendung haben genau eine Aufgabe.
- **Analysierbarkeit** – Die Anwendung stellt die Informationen bereit, die es möglich machen, Probleme effizient zu analysieren.
- **Portabilität** – Die Anwendung ist in unterschiedlichen Umgebungen lauffähig und trifft nur minimale Annahmen über externe Abhängigkeiten

Für die Messung einzelner Gesichtspunkte interner Qualität von Software steht eine Vielzahl von Metriken zur Verfügung: Testabdeckung (zum Beispiel Line and Branch Coverage, Code to Test Ratio), Komplexität (zum Beispiel McCabe Cyclomatic Complexity), Kohäsion und Kopplung (zum Beispiel LCOM4 und RFC), um nur einige zu nennen. Diese Metriken können mit Werkzeugen zur statischen Code-Analyse, wie zum Beispiel SonarQube[8] oder Coverity Scan[9], gemessen werden und geben Entwicklern und Architekten wichtige Hinweise auf mögliche Schwachstellen im Code.

Interne Qualität zielt also darauf ab, das System richtig zu implementieren. Investitionen in interne Qualität zahlen sich mittel- bis langfristig aus, da die Wartungs- und Weiterentwicklungskosten gesenkt und die Flexibilität gegenüber neuen Anforderungen erhöht wird.

Diskussionen über interne Qualität sind inhärent hoch technisch, was die Kommunikation zwischen Fach- und IT-Bereich oft erschwert. Um dem entgegenzuwirken, entwickelte Ward Cunningham[10] die „Debt"-Metapher[11, 12].

Die „Technical-Debt"-Metapher

Angenommen sei beispielhaft der Fall, dass eine neue Funktionalität zu einem bestehenden System hinzugefügt werden soll: Eine Möglichkeit, dies zu realisieren, kann zwar kurzfristig umgesetzt werden, ist aber technisch oder architektonisch nicht sauber und wird dazu führen, dass zukünftige Änderungen aufwendiger umzusetzen sind. Ein anderer Weg führt zwar zu sauberem Design, erfordert aber initial höheren Aufwand. Hier bietet die Debt-Metapher eine Analogie an, um über diese Abwägung nachzudenken.

Die Metapher beschreibt technische Schulden analog zu finanziellen Schulden. Die Aufnahme eines Kredits erlaubt es uns, Investitionen zu tätigen, die ohne Kredit erst später oder gar nicht möglich wären. Im Gegenzug dazu verpflichten wir uns, zukünftig Zins- und Tilgungszahlungen zu leisten. Während finanzielle Zinsen monetär beziffert werden können, schlagen sich technische Schulden in verlangsamter Entwicklungsgeschwindigkeit, höheren Entwicklungsaufwänden und größeren Regressionsrisiken nieder. Je länger die Schulden bestehen, umso höher ist der kumulierte zu begleichende Zins. Sind und bleiben die anfallenden Zinsen niedrig, so kann es der sinnvollste Weg sein, diese Zinsen weiterhin zu begleichen. Ebenso kommt Technical Debt nur dann zum Tragen, wenn Änderungen an dem mit Schulden behafteten System vorgenommen werden. Arbeitet das System fehlerfrei und wird nicht mehr angepasst, so entstehen keine Nachteile durch technische Schuld. Die Erfahrung zeigt jedoch, dass dies nur in

8 http://www.sonarqube.org/

9 https://scan.coverity.com/

10 Unter anderem bekannt für die Entwicklung des ersten Wikis.

11 https://www.youtube.com/watch?v=pqeJFYwnkjE

12 http://martinfowler.com/bliki/TechnicalDebt.html

13 Leider endet die Analogie hier; das aktuelle Zinsniveau bleibt auf den Finanzmarkt beschränkt.

14 1http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

wenigen Fällen eintritt. Ist ein Anstieg der Zinsen durch einen weiteren Ausbau des Systems absehbar, so kann es sinnvoll sein, die Tilgung in Angriff zu nehmen.[13]

Es besteht also die Möglichkeit, durch die Aufnahme technischer Schulden die neue Funktionalität schneller an den Markt zu bringen, etwa um das neue Feature vor Wettbewerbern anbieten zu können. Ist absehbar, dass dadurch neue Kunden gewonnen werden können oder eine neue Einnahmequelle erschlossen wird, so ist das eine wertvolle Option. Ist jedoch kein unmittelbarer ROI sichtbar und existieren keine stichhaltigen Gründe für eine übereilte Implementierung, so sollte die interne Qualität der Software nicht belastet werden. Eine konkrete Abwägung stellt Schätzungen der zu erwarteten Mehreinnahmen den erwarteten Zusatzaufwänden möglicher zukünftiger Implementierungen gegenüber.

Ursachen für Technical Debt

Technische Schulden haben ihren Ursprung in verschiedenen Sachverhalten. Sie können bewusst eingegangen werden, um eine kürzere Time-to-Market zu erreichen, oder versehentlich entstehen. Letzteres birgt die Gefahr, dass fällig werdende Zinsen nicht in der Planung berücksichtigt sind und nicht oder nur unter deutlich erschwerten Bedingungen beglichen werden können. Typischerweise wächst während der Implementierungsphase das Wissen um das zu erstellende System, die dazugehörige Geschäftsdomäne und die abgebildeten Prozesse rasant an.

Diese Erkenntnisse müssen wiederum in die Entwicklung einfließen, um die interne Struktur des Systems optimal auf die Anforderungen abgestimmt zu halten. Ein Versäumnis, dieses zu tun, erhöht wiederum die technische Schuld. Veränderungen in der Technologielandschaft können ebenfalls zu technischen Schulden führen. Neue Möglichkeiten, Probleme eleganter und effizienter zu lösen, können individuell entwickelte Systemteile obsolet machen. So altern Softwaresysteme auch ohne direkte Weiterentwicklung. Um in der Analogie zu bleiben, könnte man in diesem Fall von technischer Inflation sprechen.

Weitere Gründe für Technical Debt können folgende sein:

- zu hohe Kopplung einzelner Komponenten,
- zu geringes Prozess- oder Domänen-Verständnis,
- mangelnde Zusammenarbeit zwischen Fach- und IT-Seite,
- zu hohe Parallelisierung der Entwicklung,
- mangelnde Standardisierung (intern wie hinsichtlich Industriestandards),
- mangelnde (automatisierte) Tests,
- mangelndes Wissen bei Entwicklern.

Grundsätzlich ist davon auszugehen, dass Technical Debt in Entwicklungsprojekten entstehen wird – eine vollständige Vermeidung ist nicht realistisch. Um die Entwicklung nicht zu gefährden, muss technische Schuld aktiv, bewusst und professionell adressiert werden.

Sowohl bei der Einschätzung, ob die technische Schuld getilgt oder akzeptiert werden sollte, als auch bei der Risikobewertung bietet es sich an, die oben genannten Gründe zu klassifizieren. Martin Fowler hat dazu 2009 den „Technical Debt Quadrant“ entwickelt, der die Achsen „leichtsinnig-vernünftig“ und „absichtlich-versehentlich“ abbildet.[14]

Abbildung 2: Technical Debt Quadrant

Technical Debt messen

Leider lassen sich technische Schulden nicht so klar quantifizieren wie finanzielle Schulden, da immer nur Schätzungen der erwarteten Entwicklungsaufwände herangezogen werden können. Eine Möglichkeit ist, nach der Implementierung eines Features eine Schätzung vorzunehmen, wie lange die Implementierung in einem sauberen System – also in einem System frei von Technical Debt – gedauert hätte. Das Ergebnis entspricht den bezahlten Zinsen. Diese Methode hat jedoch den Nachteil, dass die Schätzung auf einem imaginären Stand des Systems basiert und daher wenig objektiv ist, wie folgendes Beispiel zeigt: „Wenn das Benutzerrollen-Modell feingranularer konfigurierbar wäre (das entspräche dem ‚sauberen Stand'), hätte auf die Einführung zwei weiterer Rollen verzichtet und so drei Tage Aufwand eingespart (was den gezahlten Zinsen entspricht) werden können." Dennoch ist das Ergebnis ein guter Indikator des Qualitätsstandes der Code Base, der für nichttechnische Mitarbeiter greifbar ist.

Um die Ergebnisse der Schätzungen zu verbessern, bietet es sich an, in der Retrospektive nach jeder Implementierungsiteration zu schätzen, wie viel debt-bedingter Zusatzaufwand während dieser Iteration angefallen ist und diesen anhand der bekannten Problemfelder zu klassifizieren. Dies hat den Vorteil, dass sich die Schätzung auf gerade fertiggestellte Aspekte bezieht und so genauer ist. Auch in nicht formal agil gesteuerten Projekten ist es oft sinnvoll, die Implementierung in funktional eigenständige Phasen aufzuteilen, diese iterativ durchzuführen und im Nachgang Retrospektiven abzuhalten.

Daneben kann auch die Behebung (Tilgung) bestimmter Problemfelder geschätzt und als Arbeitspaket eingeplant werden. Zusätzlich dazu sollte der wegfallende Mehraufwand geschätzt und der Behebung gegenübergestellt werden.

Werkzeuge zur statischen Code-Analyse bieten oft die Möglichkeit, Technical Debt zu errechnen, indem Verletzungen des Regelwerkes pro Regel mit einem ungefähren Behebungsaufwand hinterlegt werden. Die resultierenden Zahlen sind jedoch nicht genauer als die Resultate der oben genannten Schätzungen. Im Gegenteil, es fehlen die Risikobewertung und die Einschätzung des Einflusses einzelner Regelverletzungen. Interessant ist aber zu beobachten, wie sich der errechnete Wert im Lauf der Zeit verändert – nimmt die interne Qualität zu oder ab? Welche der oben genannten Gründe könnte dies verursachen? Wie sind die Gründe im Technical-Debt-Quadranten zu verorten? Die Beantwortung dieser Fragen hilft, versehentliches Entstehen technischer Schuld frühzeitig erkennen zu können.

Die SQUALE-Methode[15, 16] ist eine Methode zur Evaluierung von Softwarequalität mit besonderem Fokus auf das aktive Management von Technical Debt. SQALE bietet ein umfangreiches Qualitäts- und Analysemodell an. Der Aspekt der Remediation-/Non-Remediation-Kosten aus SQALE entspricht den zuvor beschriebenen Kosten für Zinsen (Non-Remediation-Cost NRC), die bei Änderungen in einer mit technischer Schuld behafteten Code Base anfallen, und Tilgung (Remediation-Cost RC), also dem Aufwand, der notwendig ist, um die technische Schuld zu beseitigen.

Investition statt Schulden

Bisher standen Schulden und deren Tilgung im Fokus – eine hilfreiche Analogie, strukturiert über Defizite in interner Qualität nachzudenken und zu kommunizieren. Diese Metapher lässt sich jedoch auch invertieren: Statt über in der Vergangenheit entstandene Schulden kann man auch über Investitionen in die Qualität, die sich in der Zukunft auszahlen werden, sprechen.

Diese Interpretation der Debt-Metapher ist eine relativ neue Entwicklung, die maßgeblich von Eberhard Wolff und Felix Müller geprägt wurde. [17, 18, 19]

Hilfreich ist hier eine Definitionen aus SQALE: Der ROI unserer Investition in Qualität ist damit Non-Remediation-Costs (NRC) minus Remediation-Costs (RC). Eine Investition zahlt sich also nur aus, wenn die NRC über einen gewissen Zeitraum größer sind als die RC.

Dadurch erhält man – genauso wie oben beschrieben – eine Entscheidungsgrundlage für die Investition.

Die Unterschiede beider Metaphern sind inhaltlich minimal. Während jedoch das Management von Schulden einen reaktiven Charakter hat, stellt der Begriff der Quality Investments eine proaktive Herangehensweise in den Vordergrund. „Schulden" betont die Rückwärtsgewandtheit: „Wir haben uns Schulden aufgeladen, jetzt müssen wir damit fertig werden." Im Gegensatz dazu betont der Ausdruck „Investition in Qualität" die Zukunftsorientierung und Strategie: „Eine Investition in die Verbesserung der Qualität wird in der nächsten Iteration zu einem positiven ROI führen."

Das einfache „Auf-den-Kopf-Stellen" der Debt-Metapher betont den positiven Aspekt: die verbesserte Qualität und die zielgerichtete Investition. Man bewegt sich gedanklich weg von der Konzentration auf die Defizite und konzentriert sich stattdessen darauf, verfügbare Ressourcen zu stärken und auszubauen. Während die negativ konnotierte Schuldmetapher eher konservative Entscheidungen hervorruft, ermutigt die zukunftsorientierte Metapher der Investition dazu, die interne Qualität von Software als strategisches Asset zu betrachten.

15 http://www.sqale.org/

16 https://en.wikipedia.org/wiki/SQALE

17 http://www.se-radio.net/2015/04/episode-224-sven-johann-and-eberhard-wolff-on-technical-debt/

18 http://www.infoq.com/articles/no-more-technical-debt

19 http://www.sigs-datacom.de/fachzeitschriften/objektspektrum/online-themenspecials/artikelansicht.html?tx_mwjournals_pi1[pointer]=0&tx_mwjournals_pi1[mode]=1&tx_mwjournals_pi1[showUid]=7714

Autor

Wolfgang Werner
Lead IT Architect,
CoC Software Engineering
> +49 (0) 89 / 943011 - 1854
> wolfgang.werner@msg-gillardon.de