www.clean-coding-cosmos.de

Clean Coding Cosmos:

Teil 2: Kosmologische Suche nach Softwareentwicklungsschmutz

Analysiert man modernen Quellcode, findet ein kundiger Leser dort reichlich Schmutz. Die unterschiedlichen Arten von Schmutz im Code lassen die Urheber (uns eingeschlossen) unfähig, faul oder feige erscheinen. Werden Softwareautoren direkt befragt, was ihnen in ihrem Alltag Probleme bereitet, nennen sie eine Reihe von Schwierigkeiten, die ihre Arbeit wesentlich beeinträchtigen. Ob Codeanalyse oder Programmierer-Befragung, die Gründe sind sehr unterschiedlich und betreffen alle Dimensionen des Entwicklerkosmos, die wir in Teil 1 dieses Artikels in der letzten Ausgabe von OBJEKTspektrum vorgestellt haben. Wir wollen Licht auf diese „dunkle Materie" in unserem Entwicklerkosmos werfen, um zu lernen, woran man Schmutz erkennt und wie man ihn beseitigt oder gar vermeidet.

Schmutz will keiner haben, „clean" soll alles sein. Aber was bedeutet diese Metapher in der *Softwareentwicklung (SE)*? Software hat zwei sehr unterschiedliche Aspekte, die beide gleich wichtig sind (**siehe Kasten 1**, „Innere und äußere Softwarequalität"). Für beide Aspekte gilt: Schmutz ist das, was Ineffizienz verursacht (**siehe Kasten 2**).

Schmutz ist die dunkle Materie im Entwicklerkosmos und, ebenso wie die physikalische Materie durch Massenanziehung von selbst akkumuliert, reichert sich in der SE Schmutz anscheinend von allein immer mehr an. Der Schmutz verzerrt die Raumzeit und wirkt auf alle Dimensionen. Die Akteure der SE müssen regelmäßig Energie aufbringen, um sich vor einem Übermaß an Schmutz zu schützen und ein gesundes Maß an Sauberkeit zu bewahren (**siehe Abbildung 1**).

Um Schmutz zu erkennen oder ihn gar zu vermeiden, stellen wir in diesem Artikel zum einen typische technologie- und sprachunabhängige Anti-Patterns vor (Schmutz im Code bzw. im Produkt). Zum anderen präsentieren wir die Ergebnisse einer Umfrage, was Softwareentwickler bei ihrer Arbeit am meisten behindert (Schmutz im SE-Alltag bzw. im Prozess).

Dirty-Code

Erschrocken betrachtet auch der erfahrene Entwickler Programm-Code, den er vor Monaten selbst geschrieben hat, und fragt sich: „War ich damals eigentlich *unfähig*, oder einfach nur zu *faul* oder gar zu *feige*, das sauber zu machen?"

Dirty-Code äußert sich durch Schmerzen bei der Wartung (**siehe Kasten 3**). Typische Symptome sind langwierige Fehleranalysen, aufwändige Umbaumaßnahmen, ungewollte Seiteneffekte bei Codeänderungen und Mängel in den nicht-funktionalen Qualitätsmerkmalen, wie Stabilität, Änderbarkeit und Testbarkeit. Konkrete *Indizien* im Code sind unter anderem:

- Codeverdopplungen
- Unverständlicher Code (unpassende Bezeichner, große Klassen, zu lange Methoden, große Verschachtelungstiefe)
- Unvollständige Implementierung (bezüglich Fehlerbehandlung, Refactoring, Logging usw.)
- Unnötig komplexe Implementierung
- Toter Code
- (Zu) starke Kopplungen zwischen Klassen und Modulen
- Vermischung von Belangen und Verantwortlichkeiten
- Fehlende Tests, mangelnde Testabdeckung
- Mangelnde Automatisierung (Build, Auslieferung usw.)

Schmutz ist alles, was die SE verzögert und ihre Akteure behindert, z. B. unverständliche Dokumente oder unklare Prozesse. Schmutz findet sich in allen vier Dimensionen des Entwicklerkosmos wieder (vgl. [Obe13]). Schmutz äußert sich darin, dass er einen Akteur bei seiner Arbeit stört. Aber nicht alles Störende ist Schmutz. Neben Schmutz stört auch, was den persönlichen Vorlieben der Akteure widerspricht. Dazwischen zu unterscheiden, ist oft sehr schwer. Schmutz ist nach Abwägen aller Vor- und Nachteile dennoch einigermaßen objektiv begründbar.

Kasten 2: Was ist Schmutz und was keiner?

Softwareentwicklungsprozess (SEP)
Der SEP schließt die Implementierung sowie die anderen Aufgaben im Entwickleralltag mit ein, z. B. Anforderungsanalyse, Qualitätssicherung und Inbetriebnahme (**vgl. auch Kasten 1** in [Obe13]).

Innere und äußere Softwarequalität
Äußere Qualität ist vom Anwender sichtbar (z. B. Funktionalität), innere Qualität ist nur für den Insider ersichtlich (Qualität des SEP, Codequalität). Innere und äußere Qualität müssen ausgewogen, d. h. aufeinander abgestimmt sein (vgl. Eisberg-Metapher in **Abbildung 1** in [Obe13]).

Dimension
Die Dimension ist ein Typ von Eigenschaften, welche die Softwareentwicklung prägt:

- **Prozess:** Zeitliche Dimension, die Abläufe und Aktionsketten thematisiert.
- **Handwerk:** Praktische Dimension, die Fertigkeiten in der Anwendung von Tools und Softwareentwicklungstechniken thematisiert.
- **Wissen:** Technische Dimension, welche die SE-Theorie thematisiert. Dazu zählen z. B. Kenntnisse in Programmiersprachen, Modellierung, Datenbankdesign und Softwarearchitektur.
- **Motivation:** Menschliche Dimension, die das Wollen der Akteure thematisiert.

Kasten 1: In Teil 1 von Clean Coding Cosmos ([Obe13]) eingeführte Begriffe.

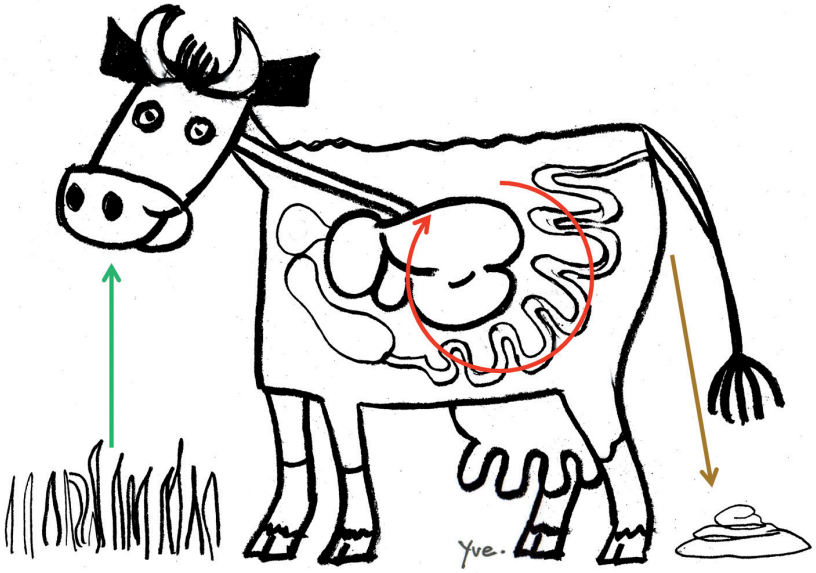


Abb. 1: Lebende Organismen wie Kühe und Software zeichnen sich durch Stoffwechsel aus: Stoffaufnahme, Verarbeitung und Ausscheidung. Funktioniert die Ausscheidung schlecht, reichern sich Schadstoffe an. Dauert diese Anreicherung an, kommt es schließlich zu Schmerzen, Organausfall oder gar zum Tod des Organismus. Eine gute Ausscheidung erreicht keine perfekte Sauberkeit (Sterilität), sondern stellt ein Fließgleichgewicht ein, bei dem die Verschmutzung auf einem maßvollen, gesunden Niveau gehalten wird.

Wenig wird über die *Ursachen* von Dirty-Code diskutiert und noch weniger über *Prävention*. Dieser Abschnitt widmet sich vorrangig diesen Themen. **Abbildung 2** zeigt die Übersicht über die im Folgenden angesprochenen Ursachen.

Fallbeispiel

Ein Mitarbeiter kommt in ein neues Projekt. Er erhält zum Einstieg eine kleine Aufgabe als Möglichkeit, Software, Tools und Infrastruktur kennenzulernen.

Wie wird der Entwickler vorgehen? Untersucht er bestehenden Code dahingehend, ob ähnliche Aufgaben bereits gelöst wurden und Lösungsstrategien übernommen werden können? Sucht er nach Tests? Arbeitet er Test-First? Vermutlich nicht.

Ein Großteil der Entwickler wird wahrscheinlich versuchen, diese Aufgabe möglichst schnell zu erledigen, um das eigene Können und seinen Wert für die Firma unter Beweis zu stellen. Dann kann er mit Stolz verkünden: „Fertig!“ Aber was bedeutet es genau, „fertig“ zu sein? Wie sieht die DoD denn aus?

1. Definition of Done (DoD)

Die so genannte „Definition of Done“ ist ein wesentlicher Bestandteil, Softwarequalität sicherzustellen. Was zur DoD gehören kann, zeigt **Abbildung 3**. Das Fehlen der DoD kann Konflikte innerhalb des Teams nach sich ziehen, denn Entwickler haben unterschiedliche Vorstellungen von „fertig“. Gegenmaßnahmen sind:

- DoD möglichst frühzeitig klären.
- DoD allen Entwicklern deutlich machen.
- Einhaltung prüfen.

Sowohl die Einigung auf eine DoD als auch ihre Einhaltung stellen ein wesentliches Problem im SEP dar. In Teil 3 dieses Artikels in OBJEKTspektrum stellen wir das *Team Clean Coding (TCC)* vor. Diese Methode soll Einigung und Einhaltung gewährleisten.

2. Die Gas-Fabrik

Inspiriert von der Komplexität einer echten industriellen Gasfabrik steht der Begriff in der SE für eine übertrieben komplexe Lösung. Ein Motiv ist typischerweise ungezügelte Neugierde. Nicht selten wird eine kleine Erweiterung zum Anlass genommen, ein neues Framework oder Pattern auszuprobieren.

Ein anderes Motiv ist die zu hohe Priorisierung von Konfigurierbarkeit und Erweiterbarkeit: Eine generische und an jede Situation anpassbare Lösung erfordert in der Regel eine unnötig komplexe Implementierung. Gegenmaßnahmen sind:

Code ist wartbar, wenn er folgende Eigenschaften besitzt:

- gut verständlich
- leicht änderbar
- einfach testbar

Verständlicher Code macht es möglich, schnell und präzise die Stellen zu finden, die für eine Implementierungsaufgabe geändert werden müssen. Leicht änderbarer Code erlaubt es, Erweiterungen ohne großen Umbau der bestehenden Implementierung realisieren zu können. Einfach testbarer Code ermöglicht es, mit wenig Aufwand sicherzustellen, dass keine unerwünschten Seiteneffekte auftreten. Jede Verletzung dieser drei Aspekte behindert die Wartung und lähmt die SE.

Kasten 3: Was ist wartbarer Code?

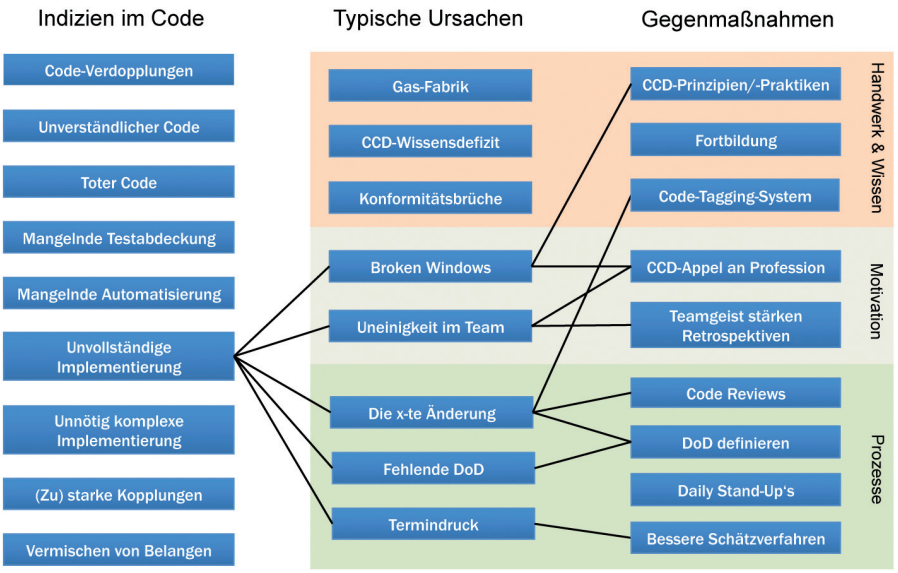


Abb. 2: Diagnose und Therapie von sprach- und technologie-unabhängigem Dirty-Code: Die Zusammenhänge zwischen den Indizien, den Ursachen und Gegenmaßnahmen sind vielfältig. Aus Gründen der Übersichtlichkeit sind die Beziehungen exemplarisch nur für ein Indiz eingezeichnet.

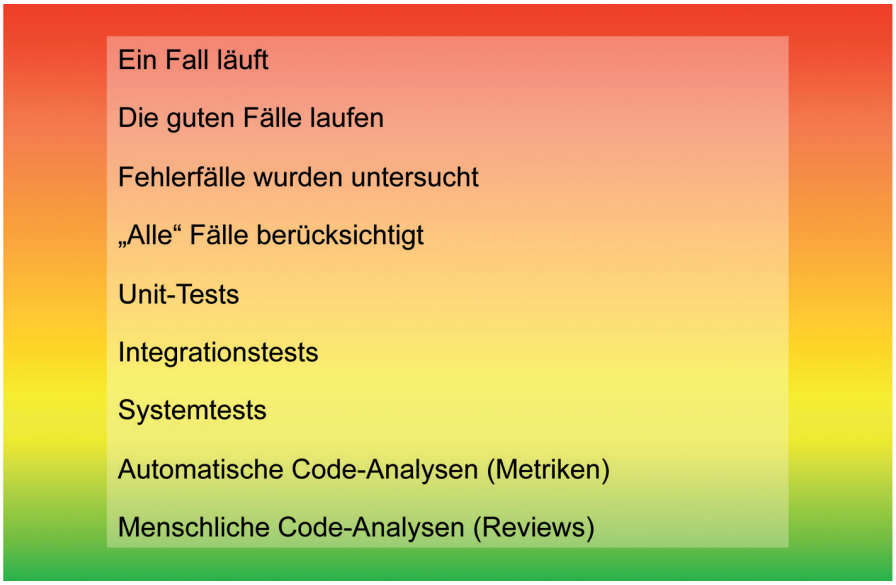


Abb. 3: Was kann und was soll die DoD beinhalten? Die Meinungen gehen dabei oft auseinander. Reichen Unit-Tests? Müssen Code-Analysen sein?

- Folgende Clean-Code-Developer-Regeln etablieren: KISS (vgl. [CCD1]), Vorsicht vor Optimierungen (vgl. [CCD2]) und YAGNI (vgl. [CCD3]).
- Einhaltung der Regeln prüfen: Code-Reviews durchführen (vgl. [CCD4])

3. Die x-te Änderung/schnell wechselnde Anforderungen

Der Projektalltag ist geprägt von ständig wechselnden Anforderungen. Folglich sind Situationen, in denen Codeteile stark umgebaut oder gar verworfen werden, nicht ausgeschlossen. Das drückt die Motivation, vollständig sauber zu arbeiten, weil ungewiss ist, ob die aktuelle Version nicht abermals überarbeitet oder verworfen werden muss. Siehe dazu auch unten im Abschnitt „Rahmenbedingungen“. Gegenmaßnahmen sind:

- Betreffende Codestellen mit einem speziellen Kommentar markieren und konsequent diese Markierungen prüfen (siehe „Code-Tagging-System“ in der nächsten Ausgabe von OBJEKTspektrum).
- Agile SE-Methoden anwenden (**aber siehe Kasten 4**).
- Frusttoleranz stärken (z.B. mit einem guten Stück Schokolade).

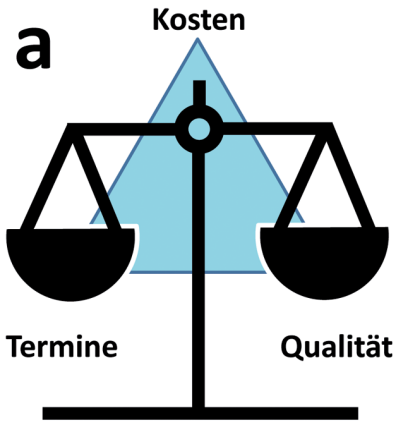
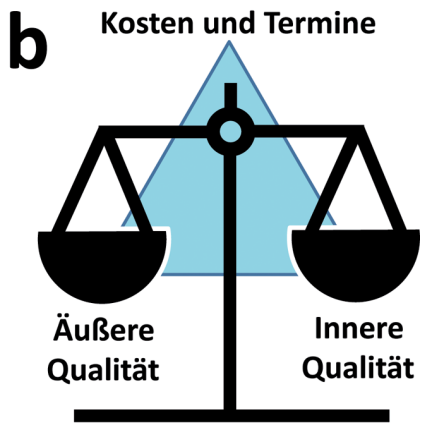


Abb. 4a: Kräftedreick aus Kosten, Terminen und Qualität. Sollten die Termine bei gleichen Kosten schwerer wiegen, würde die Qualität leiden.



4b: Innere und äußere Qualität sind gleich wichtig (vgl. Kasten 1). Allerdings wird in der Regel die äußere Qualität (z.B. Funktionalität) bei gleichen Terminen und Kosten schwerer gewichtet als die innere (z.B. Wartbarkeit), worunter letztere und damit die SE-Effizienz der folgenden Entwicklungsphasen leidet (siehe Abb. 5).

Als Antwort auf schnell wechselnde Anforderungen wurden die agilen Methoden entwickelt (vgl. [Bec01], [Coh04]). Doch auch die agilen Methoden stehen im Verdacht, zum Softwareschmutz beizutragen (Kapitel 3 in [Kor13]). Beispielsweise kann der Zwang, bei jedem Sprint auslieferungsfähige Software zu erzeugen, Termindruck und damit Softwareschmutz verursachen (siehe „7. Termindruck“). Doch die Kombination von agilen Methoden und Clean-Code-Prinzipien verspricht, diese negativen Aspekte auszugleichen (vgl. [Ble13], [Wes12]).

Kasten 4: Agile SE-Methoden.

4. Broken Windows

Dieser Begriff steht ursprünglich für ein in der Kriminalistik beschriebenes Phänomen und besagt, dass ein kleiner unbehandelter Schaden (z.B. eine zerschlagene Scheibe eines Gebäudes) zur weiteren Zerstörung einlädt und letztlich zu völliger Verwahrlosung führt (vgl. [Wiki1]). In der IT ist diese Erscheinung auch bekannt unter dem Namen Software-Entropie (vgl. [Wiki2]): Je weiter die Verrottung (z.B. einer Klasse) fortgeschritten ist, desto anstrengender wird es, sich dort hineinzudenken, und desto mehr läuft man Gefahr, Workarounds zu programmieren, anstatt einmal gründlich aufzuräumen. Damit aber wird das ganze Gebilde noch unverständlicher. Gegenmaßnahmen sind:

- Konsequent Pfadfinder-Regel anwenden (vgl. [CCD5]).
- Regelmäßig Refactoring-Maßnahmen durchführen (vgl. [CCD6], [CCD7]).

5. Konformitätsbrüche

Ein Konformitätsbruch liegt dann vor, wenn der Code plötzlich von einem regelmäßigen Implementierungsschema abweicht. Beispiel: Für einen Teil der Businessobjekte wurde zur Datenspeicherung Library A verwendet, für den anderen Library B. Die Folge ist, dass ein weiterer Entwickler später einen unnötigen Teil seiner Zeit damit verbringt, nach dem Grund der Abweichung zu suchen, und sich in zwei Lösungswege für das gleiche Problem einarbeiten muss.

Eine typische Ursache dafür ist ein unvollständiges Refactoring, das – eventuell aus Zeitnot – vorzeitig beendet wurde. Eine andere Ursache liegt darin, dass ein Entwick-

Schätzfehler in der Softwareentwicklung sind eine häufige Ursache für falsche Projektplanung (vgl. „Estimation“ in [Mar11]) und für unnötigen Zeitdruck (vgl. „Time Management“ in [Mar11]):

1. Komplexe (Software-)Systeme sind unberechenbar. Stellenweise wird die Materie umso komplizierter, je tiefer man in sie eindringt, und die Wechselwirkungen ihrer Bausteine entziehen sich einer präzisen Vorhersagbarkeit (vgl. [Roy04]). Ohne Berücksichtigung dieser Zusatzaufwände (siehe Punkt 2) fallen regelmäßig deutliche ungeplante Mehraufwände an.
2. Der Mensch schätzt psychologisch eher den für ihn *wahrscheinlichsten* als den (höheren) *durchschnittlich* zu erwartenden Aufwand (**siehe Abbildung 6**). Das heißt, die in Punkt 1 genannten Ausnahmefälle, die sehr viel länger benötigen als erwartet, fließen in die persönliche Schätzung nicht mit dem richtigen Gewicht ein.
3. Projektmanager tragen die Verantwortung für kurzfristige Ziele. Sie werden an der zeitlichen Einhaltung von *Terminen* gemessen. Folglich werden Entwickler (seitens des Managements) für Geschwindigkeit und weniger für Sorgfalt belohnt.
4. Entwickler neigen dazu, schon während der Schätzung mit kurzen Fertigstellungszeiten Konflikten mit dem PM ausweichen oder damit beeindrucken zu wollen.
5. Einige Ursachen sind organisatorischer oder politischer Art. Ungünstige Projektfaktoren (siehe unten „Rahmenbedingungen”) verzögern häufig unerwartet die technische Umsetzung wegen benötigter, aber nicht rechtzeitig zur Verfügung gestellter Ressourcen: Hardware, Software, Manpower.

Kasten 5: Schätzfehler bei Aufwandsabschätzungen.

ler eine neue Aufgabe mit einer bereits im Code umgesetzten Lösung realisieren könnte, diese aber gar nicht kennt (mangelnde Kommunikation) oder sogar glaubt, sie anders besser lösen zu können (Profilierungsneigung). Gegenmaßnahmen sind:

- Code-Review oder Pair-Programming: Gemeinsam entscheiden, ob eine neue Lösung etabliert werden soll.
- Vollständiges Refactoring: Alle Stellen umbauen, wenn eine neue Lösung etabliert wird.

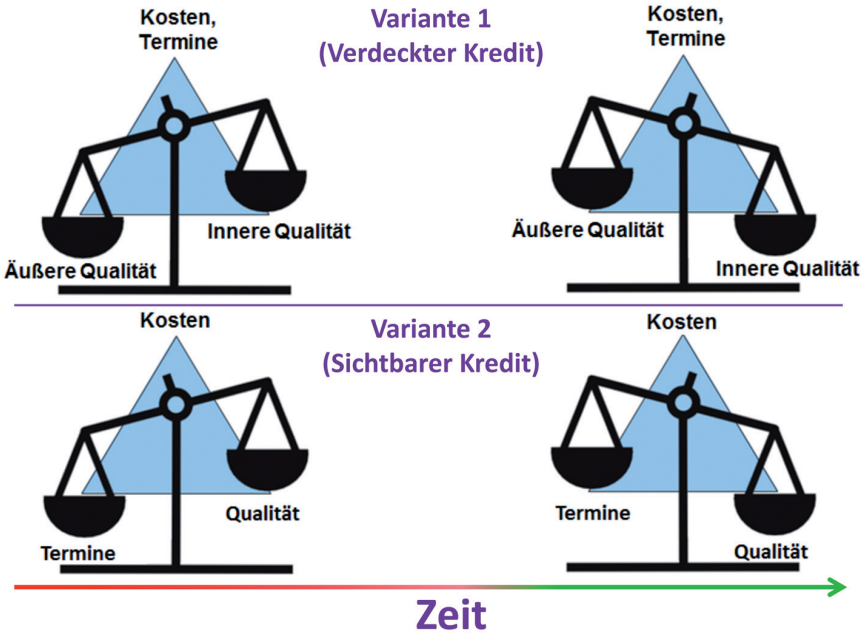


Abb. 5: Technischer Kredit: Technische Schulden (vgl. [Wiki4]) können in einer heißen Phase zu Gunsten der Kosten und Termine aber zu Lasten der Qualität bewusst eingegangen werden. Das kann verdeckt geschehen (nur Schulden an die innere Qualität) oder sichtbar für den Kunden und die Anwender (Schulden an die äußere Qualität). Alle Schulden müssen aber in der Folgephase bewusst wieder beglichen werden. Ansonsten droht Organausfall (siehe Abbildung 1) bzw. der Untergang des Eisbergs „Software“ (vgl. Abbildung 1 in [Obe13]).

6. Uneinigkeit im Team

Es gibt viel Literatur zu den so genannten *Soft Skills* (vgl. [Sch10]). Soft Skills haben einen großen Einfluss auf die Effizienz des SEP, denn persönliche Eigenheiten von Teammitgliedern können die Stimmung im Team belasten, die Kommunikation einschränken und für Uneinigkeit sorgen. In solchen Situation arbeitet jeder für sich allein, sodass gemeinsame Ideen von Lösungen nicht existieren und Wildwuchs im Code entsteht. Gegenmaßnahmen sind:

- Mehr Mut, sich ehrlich selbst zu reflektieren.
- Mehr Mut, mit anderen in Kommunikation zu treten.
- Mehr Motivation, Verantwortung zu übernehmen (sich zu kümmern).
- Mehr Motivation, Neues zu lernen und auszuprobieren.

7. Termindruck

Als Hauptgrund von Dirty-Code wird von Entwicklern eigentlich immer als erstes Termindruck genannt. In der Tat kann dieser große Probleme aufwerfen. Das Kräftedreieck in **Abbildung 4** zeigt, dass die Faktoren Kosten, Termine und (äußere und innere) Qualität miteinander in einem konkurrierenden Zusammenhang stehen. Bei der dort abgebildeten Waage hat ein „Druck auf den Termin“ bei gleichbleibenden Kosten stets einen negativen Einfluss auf die Softwarequalität – und zwar in der Regel auf die *innere* Qualität (**siehe Kasten 1**). Mangelnde innere Qualität beeinträchtigt Merkmale wie Analysierbarkeit, Änderbarkeit und Testbarkeit und lähmt deshalb langfristig die Effizienz der SE (siehe auch unten, Abschnitt „Projekt-Management“). Gegenmaßnamen sind:

- Verantwortung für die innere Softwarequalität übernehmen und seinem Berufsethos (den Clean-Coding-Regeln) treu bleiben.
- Termindruck durch realistische Aufwandsschätzungen (**siehe Kasten 5**) reduzieren.
- In Ausnahmefällen kann ein technischer Kredit aufgenommen werden (**siehe Abbildung 5**).
- Frusttoleranz stärken (diesmal vielleicht mit frischer Luft bei einem kleinen Spaziergang)

Schmutz im SE-Alltag

Die Ergebnisse zur Umfrage, was Softwareentwickler in ihrem Alltag am meisten

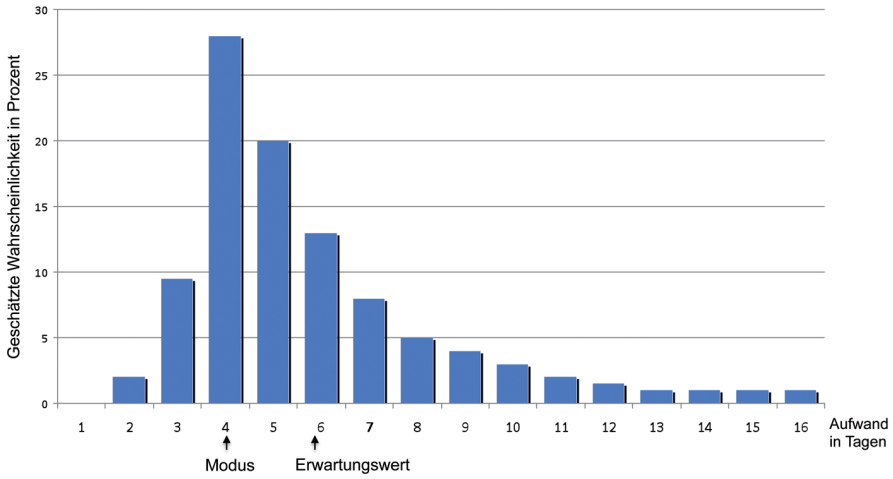


Abb. 6: Inhärenter Fehler bei der Aufwandsschätzung: Typischerweise schätzen Entwickler den für sie wahrscheinlichsten Aufwand, den so genannten Modus (vgl. [Wiki5]). Für die Projektplanung ist aber ein anderer Schätzwert von größerer Bedeutung, der die weniger wahrscheinlichen hohen Werte auch berücksichtigt, der Erwartungswert (vgl. [Wiki6]).

behindert, zeigt **Tabelle 1**. Die Antworten beziehen sich nicht nur auf den Code, sondern auf den gesamten SEP. Sie lassen sich in sechs Problemtypen einteilen:

1. Requirements Engineering (RE)

RE steht nicht zu Unrecht auf Platz 1 von **Tabelle 1** und auf Platz 2 von **Tabelle 2**, denn falls Entwickler nicht ausreichend mit richtigen Anforderungen versorgt werden, führt dies zu Schmutz, der die Softwareentwicklung lahmlegen kann. Aber wie ist der große Abstand zu den anderen Punkten zu erklären? Ursachen für RE-Probleme können in allen Dimensionen liegen:

- *Prozess:* Gibt es Akteure im Projekt mit der alleinigen Aufgabe, Entwickler mit Anforderungen zu versorgen? Wir meinen: Nein, in vielen Projekten nicht.
- *Wissen:* Sind diese Akteure RE-Spezialisten? Gibt es Best Practices und werden diese angewendet? Wir meinen: Nein, in vielen Projekten nicht.
- *Handwerk:* Sind die verwendeten Tools leistungsstark genug und werden sie sinnvoll eingesetzt? Wir meinen: Nein, in vielen Projekten werden nur Office-Programme verwendet. Versionierung, Strukturierung, gute Suchfunktionen, Traceability und die Verwaltung von Metainformationen fehlen.
- *Motivation:* Ist bei allen Akteuren wirklich der Wille da, sauber zu arbeiten? Stimmt die Kommunikation mit der

Problem	%	Problemtyp
Unklare Anforderungen	51,1	RE
Zu häufig & schnell wechselnde Anforderungen	38,0	RE
Schlechtes Projektmanagement	29,2	PM
Termindruck => Quick & Dirty	28,5	Rahmenbedingungen, PM
Geringe oder fehlende Testabdeckung	27,0	Technische SEP-Schwächen
Unverständlicher Code (z.b. weil historisch gewachsen)	27,0	Technische SEP-Schwächen
Unzureichende Kommunikation im Team	26,3	Kommunikation
Ineffektive (Entwicklungs-) Prozesse	20,4	PM
Unnötige Meetings & Diskussionen	16,1	Kommunikation
Nicht machbare Aufwandsschätzungen	14,6	Rahmenbedingungen, PM
Technologien unpassend | ineffizient eingesetzt	13,9	Technische SEP-Schwächen
Schlechte Teamstimmung	13,1	Teamstimmung
Unzureichende Ausbildung der Mitarbeiter	12,4	Rahmenbedingungen
Persönliche (eigene) Schwächen	10,2	Rahmenbedingungen
Management-Druck (Überstunden, schneller entwickeln!)	8,8	Rahmenbedingungen, PM
Starre Organisationsstrukturen	8,8	Rahmenbedingungen
Mangelnde Fortbildungsmöglichkeiten	8,0	Rahmenbedingungen
Mangelndes Verständnis für Softwareentwicklung im Management	8,0	PM
Nicht-kompilierbarer Code im (shared) Repository	5,8	Technische SEP-Schwächen
Schlechte Tools	5,8	Rahmenbedingungen, Technische SEP-Schwächen
Motivationsmängel: z.B. wenig Feedback, Kritik, Lob	5,8	Teamstimmung, PM
Keine Zeit für ausreichende Tests	5,1	Rahmenbedingungen, PM, Technische SEP-Schwächen
Mangelnde Rollenklärung / Rollenverständnis	4,4	PM
Menschliche Schwächen von Teammitgliedern	3,7	Rahmenbedingungen

Tabelle 1: Ergebnis der Umfrage unter http://umfrage.clean-coder.de (Stand vom 1.11.2013 mit 142 Teilnehmern). Jeder Teilnehmer konnte bis zu 5 Probleme ankreuzen, die ihn am meisten behindern. Die Prozentwerte geben den Anteil der Personen an, die dieses Problem angekreuzt haben. Probleme mit weniger als 3% sind hier nicht aufgeführt. Alle Probleme sind einem oder mehreren von sechs Problemtypen zugeordnet (siehe Tabelle 2).

Problemtyp	%
PM	26,8
RE	19,1
Rahmenbedingungen	22,7
Technische SEP-Schwächen	18,2
Kommunikation	9,1
Teamstimmung	4,1

Tabelle 2: Kategorisierung der Umfrageergebnisse von Tabelle 1, wobei pro Problemtyp die relative Anzahl der Kreuze in Prozent angegeben ist.

Fachseite und haben die Stakeholder dort genug Zeit, ausreichend fachlichen Input zu liefern? Wir meinen: Nein, in vielen Projekten hat die Fachseite nicht genügend Ressourcen zur Kommunikation mit der Entwicklung.

Alle Dimensionen müssen clean sein, damit das RE und damit der SEP effizient laufen. Aber in allen Dimensionen gibt es noch viel Verbesserungspotenzial. Aus diesem Grund ist der hohe Abstand der RE zu den anderen Problemen in **Tabelle 1** durchaus verständlich.

2. Projektmanagement (PM)

Ein Softwareentwickler braucht für seine Arbeit ein hohes Maß an Kreativität. Manche Entwickler betrachten sich deshalb eher als Künstler. Derjenige Akteur, der als Nicht-Entwickler am meisten mit ihm zusammenarbeitet, ist der Projektmanager bzw. das PM (**siehe Kasten 6**). Immer wieder schränkt das PM den Spielraum des Entwicklers ein und hat daher bei vielen Entwicklern ein schlechtes Image. Doch auch aus anderen Gründen finden sich vier PM-Probleme unter den Top 10 in **Tabelle 1**.

Ein typischer Grund für Einschränkungen ist Termindruck (siehe oben), der im Interessenskonflikt zwischen Entwicklung und Vertrieb begründet liegt. Zusammen mit den Entwicklern sollten der Projektmanager oder gegebenenfalls der Entwicklungsleiter gemeinsam dafür kämpfen, dass genügend Zeit zur Verfügung steht. Unrealistische Aufwandsabschätzungen sind hier der größte Feind (**siehe Kasten 5**). Als Entwickler scheut man davor, Aufwandszahlen zu nennen, die im PM unwillkommen sind. Das geschieht aus mangelnder Erfahrung, aus mangelnder Motivation zur Aufwandsanalyse oder aus mangelndem Mut, Zahlen zu nennen, die Stress verursachen (**siehe Kasten 7**). Entwickler sollten sich stets ehrlich bemühen, realistische Aufwände zu schätzen, und Projektmanager ein Einsehen haben, diese zu akzeptieren.

Gutes Projektmanagement äußert sich darin, dass nicht nur kurzfristige Ziele (**siehe Kasten 5**) sondern auch langfristige verfolgt werden. Dann können nämlich Entwickler und Projektmanager zusammen einen Weg suchen, einen Rahmen zu schaffen, in dem die Entwickler sauber arbeiten können: einen *Clean Coding Cosmos*. Aber es gibt auch schlechtes PM. Hier helfen kein Meckern und Klagen, sondern die Motivation, diplomatisch zu kommunizieren: Feedback geben und konstruktive Kritik äußern (**siehe Kasten 7**).

3. Technische SEP-Schwächen

Es gibt eine Reihe von technischen Problemen im SEP (**siehe Tabelle 1**). Zusammen können sie mit 18,2 Prozent aller Probleme in Zusammenhang gebracht werden (**siehe Tabelle 2**). Auch die Ursachen von technischen Problemen können in allen Dimensionen liegen:

- *Prozess:* Sind Qualitätssicherungsmaßnahmen gut im SEP integriert?
- *Wissen:* Ist das Know-how vorhanden, wie Architektur, Implementierung, Build, Release und Qualitätssicherung sauber realisiert werden?
- *Handwerk:* Sind die richtigen Tools im Einsatz? Werden sie effizient eingesetzt? Werden Softwareentwicklungstechniken (z.B. Domain Driven Design (DDD), Test Driven Development (TDD)) angewandt?
- *Motivation:* Sind alle Akteure bereit, Prozesse einzuhalten, sich neues Wissen anzueignen oder neue Tools und Techniken einzusetzen?

Mehr als jeder vierte Teilnehmer der Umfrage gab an, dass fehlende Tests ein Problem sind. Mehr als jeder fünfte Teilnehmer gab an, dass auch unwartbarer Code ein Problem ist. Deshalb geben wir folgende Empfehlungen:

- Mit den Regeln der Clean Code Developer entwickeln und die beiden Techniken TDD und TCC (siehe Teil 3 dieses Artikels) anwenden.

Ein SE-Projekt muss in mehrfacher Hinsicht betreut werden:

- Technisch: Architektur u. a.
- Fachlich: Anforderungen
- Finanziell: Budget, Zeit, Qualität
- Personell: Teamstimmung u.a.
- Organisatorisch: Prozesstreue, Infrastruktur u.a.

Es gibt verschiedene Strategien diese Verantwortlichkeiten zu verteilen. Im klassischen PM findet man die Strategie, die Verantwortlichkeiten in möglichst wenigen Personen zu vereinen (z.B. in der eines Projektleiters). Scrum mit seiner agilen Vorgehensweise beinhaltet die Strategie, diese Verantwortlichkeiten aufzuteilen. Hier liegt die technische Verantwortung beim Entwicklerteam, die fachliche und finanzielle beim Product Owner, die personelle und organisatorische Verantwortung beim Scrum Master. Unabhängig von der Strategie besteht die wesentliche Aufgabe des PM darin, den Rahmen des Projekts gegenüber der Entwicklung, den Geldgebern und der Fachseite zu definieren und zu gewährleisten.

Kasten 6: Aufgaben des PM.

Grundsätzlich können Konflikte auf zwei Arten gelöst werden: „Fight or Flight“, zwei Strategien, welche die Psychologie uns lehrt (vgl. [Wiki7]). „Fight“ bedeutet hier aber keine Aggression, sondern eine aktive, konstruktive Problemlösung durch Kommunikation oder – im Fall von Kommunikationsproblemen – durch Meta-Kommunikation (vgl. [Sch10]). „Flight“ bedeutet hier keine Kommunikationsverweigerung, sondern die dauerhafte Trennung der Konfliktpartner, zum Beispiel durch Projektwechsel eines Beteiligten.

Neben diesen beiden aktiven Reaktionsmöglichkeiten ist es auch möglich, nichts zu tun, abzuwarten, die Frusttoleranz zu stärken und durchzuhalten. Dauerhaft geht das aber nur, wenn der dabei empfundene Disstress nicht zu groß ist. Ansonsten besteht die große Gefahr zu resignieren (**siehe Abbildung 5** in [Obe13]).

Kasten 7: Möglichkeiten, mit Konflikten umzugehen.

- Die Dirty-Code-Anti-Patterns bewusst vermeiden
- Den Einsatz von CI-Tools (vgl. [Pal04]) im Prozess verankern.

4. Kommunikation

Kommunikationsprobleme sind häufig. Zwei haben es in die Top 10 in **Tabelle 1** geschafft und fast jeder zehnte Entwickler gab Kommunikationsprobleme an (**siehe Tabelle 2**). Dabei gibt es drei typische Fehler: Erstens zu wenig, zweitens zu viel unnötige, und drittens notwendige, aber ineffektive Kommunikation. Wie bei allen Konflikten gibt es verschiedene Möglichkeiten, damit umzugehen (**siehe Kasten 7**). Wie eine sinnvolle Kommunikation im Projektalltag gestaltet werden kann, stellen wir in Teil 3 dieses Artikels dar.

Das Kommunikationsvermögen eines Menschen hat mehrere Aspekte:

- Fehlende oder vorhandene natürliche Begabung.
- Erlernbares Kommunikationsgeschick.
- Die persönliche Beziehung zu dem Kommunikationspartner. Antipathie zwischen den Beteiligten kann die Kommunikation und damit die Softwareentwicklung drastisch lähmen.

Durch gezieltes Training und spezielle Kommunikationsprozesse (vgl. [Sch10]) können Kommunikationsprobleme abgeschwächt oder sogar gelöst werden.

5. Rahmenbedingungen

In jedem Projekt gibt es zwei Sorten von Rahmenbedingungen, welche die Arbeit der Akteure erschweren: Ungünstige *Naturkonstanten* und ungünstige *Projektfaktoren*.

Zu den Naturkonstanten zählen Termindruck und schnell wechselnde Anforderungen. Sie treten immer auf, auch wenn niemand etwas falsch macht (vgl. auch „The Uncertainty Principle“, [Mar11]). Nicht ohne Grund rangieren sie in den Umfrageergebnissen in den Top 5 (**siehe Tabelle 1**).

Ungünstige Projektfaktoren sind z. B. starre Organisationsstrukturen, Teamzusammensetzung, räumliche Aufteilung des Teams, dauerhafter Managementdruck für höhere Entwicklerleistung sowie mangelnde Fortbildungsmöglichkeiten. Sie treten nicht immer auf, aber wenn sie existieren, sind sie oft schmerzhaft und ebenfalls kaum änderbar – und wenn, dann nur mittels Kommunikation.

6. Teamstimmung

Zwischenmenschliche Probleme können ebenfalls Effizienzkiller darstellen. Allerdings ist das eher selten (**siehe Tabelle 2**). Uneinigkeit bei Entscheidungen, Schwächen einzelner Mitglieder, welche die Zusammenarbeit belasten, oder mangelndes Interesse an der eigenen Arbeit („… ist doch egal, wie ich das mache, schaut sich ja sowieso keiner an …“) drücken häufig die Motivation der Mitarbeiter und damit die Effizienz der Softwareentwicklung. Hinzu kommen vielleicht auch ungünstige Rahmenbedingungen.

Außerdem kann sich als Folge von Monotonie, Unterforderung oder mangelnder Wertschätzung eine Null-Bock-Stimmung („innere Kündigung“) einstellen – neuerdings auch „Boreout“ genannt (vgl. [Wiki3]).

Hinter schlechter Teamstimmung steht immer ein Konflikt, entweder zwischen Teammitgliedern (interner Konflikt) oder mit einem externen Konfliktpartner. In beiden Fällen gibt es die in **Kasten 7** aufgezeigten Möglichkeiten, damit umzugehen.

Gesamtbewertung der Umfrage

Tabelle 2 zeigt, dass über ein Viertel aller Probleme mit dem PM in Verbindung gebracht werden können. Dieser hohe Wert mag teilweise am schlechten PM-Image bei den Entwicklern liegen, ist aber bestimmt nicht dadurch allein erklärbar. Die PM-Probleme haben mit den RE-Problemen und den Rahmenbedingungsproblemen gemeinsam, dass sie von den Entwicklern nicht alleine gelöst werden können. Hier können nur Mut zur Kommunikation und Aufklärung weiterhelfen. Bei Problemen mit technischen Aspekten des SEP, mit (teaminterner) Kommunikation und mit der Teamstimmung ist das anders. Hier können die Entwickler vieles allein zum Besseren wenden. Voraussetzung hierfür ist: In allen vier Dimension Neuem gegenüber aufgeschlossen zu sein.

Fazit

SE-Schmutz ist die dunkle Materie im Entwicklerkosmos, die wie echte physikalische Materie dazu neigt, zu akkumulieren und sich zu verdichten. Ohne aktive Anstrengungen gegen Schmutz werden immer mehr Umwege und Workarounds nötig, um die erkrankte Software am Leben zu halten (Verzerrung der SE-Raumzeit). SE-Schmutz findet sich in allen Dimensionen des Entwicklerkosmos und manifestiert sich vor allem in schlecht wartbarem Code, suboptimalen Prozessen und verbesserungswürdiger Kommunikation zwischen den Akteuren. Zur Prävention von SE-Schmutz halten wir folgende Kernkompetenzen für wichtig:

- Mehr Motivation, sich neues Wissen und handwerkliche Fähigkeiten anzueignen.
- Mehr Motivation, konstruktive Kommunikation zu suchen.
- Mehr Mut, seinem Berufsethos treu zu bleiben.

Alle in diesem Artikel angesprochenen Maßnahmen kann jeder Entwickler für sich allein verfolgen. Im dritten Teil dieses Artikels stellen wir vor, wie diese Maßnahmen effektiv und systematisch im Entwicklerteam realisiert werden können. ||

Die Autoren

|| Dr. Reik Oberrath
(R.Oberrath@iks-gmbh.com)
ist als IT-Berater bei der iks Gesellschaft für Informations- und Kommunikationssysteme tätig. Er beschäftigt sich seit vielen Jahren mit der Entwicklung von individuellen Geschäftsanwendungen und mit der Architektur komponentenbasierter Systeme.

|| Jörg Vollmer
(info@informatikbuero.com)
ist freiberuflicher IT-Berater und verfügt über langjährige Erfahrung als Entwickler, Softwarearchitekt und Trainer im Java-Umfeld, speziell Java EE und Spring. Seine Leidenschaft gehört dem Vermitteln von Software-Qualitätsbewusstsein.

Literatur & Links

[Bec01] K. Beck, et al., Manifesto for Agile Software Development, 2001, siehe: http://agilemanifesto.org/

[Ble13] M. Bless, T. Bührer, Scrum & Clean Code Development: Ein perfektes Paar? in: OBJEKTspektrum 5/2013

[CCD1] Clean Code Developer, Roter Grad, siehe: http://www.clean-code-developer.de/Roter-Grad.ashx#Keep_it_simple_stupid_KISS_1

[CCD2] Roter Grad – Clean Code Developer, siehe: http://www.clean-code-developer.de/Roter-Grad.ashx#Vorsicht_vor_Optimierungen!_2

[CCD3] Blauer Grad – Clean Code Developer, siehe: Phttp://www.clean-code-developer.de/Blauer-Grad.ashx#You_Ain%C2%B4t_Gonna_Need_It_YAGNI_2

[CCD4] Oranger Grad – Clean Code Developer, siehe: http://www.clean-code-developer.de/Oranger-Grad.ashx#Reviews_7

[CCD5] Roter Grad – Clean Code Developer, siehe: http://www.clean-code-developer.de/Roter-Grad.ashx#Die_Pfadfinderregel_beachten_4

[CCD6] Roter Grad – Clean Code Developer, siehe: http://www.clean-code-developer.de/Roter-Grad.ashx#Einfache_Refaktorisierungsmuster_anwenden_7

[CCD7] Gelber Grad – Clean Code Developer, siehe: http://www.clean-code-developer.de/Gelber-Grad.ashx#Komplexe_Refaktorisierungen_9

[Coh04] M. Cohn, User Stories Applied: For Agile Software Development, Addison-Wesley Professional 2004

[Kor13] H.-P. Korn, Das „agile" Vorgehen: Neuer Wein in alte Schläuche – oder ein „Déjà-vu"?, in: GI Edition Proceedings Band 224 – Vorgehensmodelle 2013, siehe: http://www.korn.ch/archiv/publikationen/Neuer-Wein-in-alte-Schlaeuche-oder-Deja-vu.pdf

[Mar11] R.C. Martin, The Clean Coder: A Code of Conduct for Professional Programmers, Prentice Hall 2011

[Obe13] R. Oberrath, J. Vollmer, Clean Coding Cosmos: Teil 1: Kosmologie für Softwareentwickler, in: OBJEKTspektrum 6/2013

[Pal04] M. Paluch, Testautomatisierung mit Continuous Integration: Ist manuelles Testen noch sinnvoll? in: OBJEKTspektrum 4/2013

[Roy04] Royal Academy of Engineering, British Computer Society, The Challenges of Complex IT Projects, 2004

[Sch10] B. Schneider, U. Vigenschow, Soft Skills für Software-Entwickler: Fragetechniken, Konfliktmanagement, Kommunikationstypen und -modelle, dpunkt.verlag 2010

[Wes12] R. Westphal, „Wir glauben nicht an schöne Worte" – Interview mit CCD-Mitbegründer Ralf Westphal, in: OBJEKTspektrum 4/2012

[Wiki1] Wikipedia, Broken-Windows-Theorie, siehe: http://de.wikipedia.org/wiki/Broken-Windows-Theorie

[Wiki2] Wikipedia, Software entropy, siehe: http://en.wikipedia.org/wiki/Software_entropy

[Wiki3] Wikipedia, Diagnose Boreout, siehe: http://de.wikipedia.org/wiki/Diagnose_Boreout

[Wiki4] Wikipedia, Technische Schuld, siehe: http://de.wikipedia.org/wiki/Technische_Schuld

[Wiki5] Wikipedia, Modus, siehe: http://de.wikipedia.org/wiki/Modus_(Statistik)

[Wiki6] Wikipedia, Erwartungswert, siehe: http://de.wikipedia.org/wiki/Erwartungswert

[Wiki7] Wikipedia, Fight-or-flight, siehe: http://de.wikipedia.org/wiki/Fight-or-flight